CargoAPI

Nuno Filipe Couto Pereira

Dissertação para obtenção do Grau de Mestre em

Engenharia e Informática de Computadores

Orientador(es)/Supervisor(es): Prof. António Rito Silva

Júri

Presidente: Prof./Dr. Lorem Ipsum
Orientador: Prof. António Rito Silva
Vogais: Prof./Dr. Lorem Ipsum
Prof./Dr. Lorem Ipsum

Maio 2017

Abstract

This thesis describes the modernization of a legacy system called Transporter, which is a transport and logistics Enterprise Resource Planning (ERP) implemented by MAEIL. Transporter is a client/server application which has business logic in both parts. MAEIL intents that Transporter becomes more user friendly and easier to develop for it. In order to achieve this goal, it is needed to create the CargoAPI, an Application Programming Interface (API) to expose services. These services will allow the communication between Transporter's functions and a web browser user interface. Along this thesis, we describe three possible solutions to implement this API as well the pros and cons of each solution. Transporter is an application with a client-server architecture with business logic present on both sides. In this thesis it is also described how the integration is made between Transporter and CargoAPI, how the API integrates with the database, and how requests are made in order to communicate with CargoAPI. We also propose a Representional State Transfer (REST)-like query language as a template to show the different combinations of functionalities provided by the Transporter system.

Keywords: Transporter, Legacy Systems Modernization, API, Integration.

Resumo

Este documento descreve a modernização de um sistema legado chamado Transporter. O Transporter é um ERP na área de transportes e logística desenvolvido pela empresa MAEIL. Esta empresa pretende que o Transporter se torne mais apelativo para o utilizador comum e que acompanhe a evolução da tecnologia. Esta modernização consiste na criação da CargoAPI, ou seja, criar uma API para expor funcionalidades do Transporter de modo que seja possível ter uma interface *web browser* que substitua a interface atual deste. São descritas três possíveis soluções já existentes para implementar esta API tendo em conta as vantagens e desvantagens de cada uma delas. O Transporter é uma aplicação com uma arquitetura cliente-servidor com lógica de negócio presente em ambas as partes. É descrito como é que a integração é realizada entre o Transporter e a CargoAPI, ou seja: a integração desta API com a base de dados, os pedidos realizados na comunicação com a CargoAPI e a linguagem de REST para expressar as diferentes combinações de funcionalidades fornecidas pelo sistema do Transporter.

Palavras-Chave: Transporter, Modernização, Sistemas Legados, API, Integração.

Índice

Lista de Figuras

Acrónimos

API Application Programming Interface. iii, v, xiii, 1, 14, 17, 18, 23, 26, 31–33, 37, 39, 41

AT Autoridade Tributária e Aduaneira. 3

BL Bill of Lading. xiii, 4–6, 32

CA Computer Associates. 1, 3, 7, 8, 17, 19, 39

COM Component Object Model. 7, 18, 39

CRM Customer Relationship Management. 3

CRUD Create, Read, Update and Delete. 14

EDI Electronic Data Interchange. 3, 32

ERP Enterprise Resource Planning. iii, v, 3, 41

HTTP Hypertext Transfer Protocol. 13

IIS Internet Information Services. 17–19, 21, 28

JSON Javascript Object Notation. 13, 14, 32

ODBC Open Database Connectivity. 7

OLE Object Linking and Embedding. 7

RAD Rapid Application Development. 3

REST Representional State Transfer. iii, v, 1, 13, 14, 17, 18, 31, 32

RPC Remote Procedure Call. 7

SGBD Sistema de Gestão de Base de Dados. 12

SOAP Simple Object Access Protocol. 1, 13

UML Unified Modeling Language. xiv, 31

URL Uniform Resource Locator. 32, 39

WCF Windows Communication Foundation. xiii, 17, 19–21, 23, 24, 28

XML Extensible Markup Language. 13

Capítulo 1

Introdução

Sistemas legados são sistemas desatualizados ou aplicações de *software* que são usadas sem possibilidades de terem novas versões. Por outras palavras, sistemas legados são antigos e usam *software* desatualizado, o que pode por vezes ser um problema para os utilizadores que precisam de os utilizar. Ocasionalmente, um sistema legado não preenche totalmente as necessidades de um cliente pela interface entrar em desuso. Como estes sistemas normalmente têm anos de desenvolvimento, ou seja, têm bastante lógica de negócio implementada, o que torna necessário muito tempo e recursos gastos para reescrever tudo do princípio. Como as empresas pretendem sempre reduzir custos, existe um mecanismo básico que possibilita modernizar um sistema em pouco tempo que é criar uma interface bem desenhada, conhecida por API [1]. Uma API é uma biblioteca de funções ou uma especificação de serviços REST ou Simple Object Access Protocol (SOAP) que são formas de aceder a *web services*, permitindo expor funcionalidades para outras aplicações que, por sua vez, poderão ser sistemas legados. Resumindo, tudo isto permite ter uma nova interface de utilizador que comunica com uma API que expõe funcionalidades de um sistema legado.

1.1 Descrição do Problema

O Transporter[1] é um sistema legado com 17 anos de desenvolvimento pela empresa MAEIL[2] implementado numa ferramenta chamada Computer Associates (CA) Plex. Esta ferramenta está a entrar em desuso, o que limita o Transporter a nível de acompanhamento de evolução de tecnologias e é difícil encontrar programadores que já saibam trabalhar com a mesma. Adicionalmente, a curva de aprendizagem para desenvolver em Plex é pouco acentuada. Uma outra limitação, é a arquitetura atual do Transporter que obriga a que este seja instalado localmente em cada computador. De modo que se consiga seguir as tecnologias modernas, a MAEIL pretende ter uma interface *web browser* para substituir a atual.

[1] www.transportersystems.com
[2] www.maeil.pt

Capítulo 2

Transporter

O Transporter é um ERP focado na área de transportes e logística, certificado pela Autoridade Tributária e Aduaneira (AT) que permite fazer a gestão de processos logísticos. Atualmente tem em registo mais de 50 instalações em Portugal, Angola e Reino Unido. Este *software* contém centenas de entidades no seu modelo relacional e milhares de funcionalidades em cada um dos seus módulos. É desenvolvido em CA Plex com uma arquitetura cliente-servidora pela MAEIL.

O CA Plex é uma plataforma Rapid Application Development (RAD) que suporta modelação, criação de relatórios e geração de código num único ambiente tendo também o seu próprio repositório que permite gerir toda a estrutura de código e da base de dados. Esta plataforma permite a criação de aplicações cliente-servidor sem ter a necessidade de preocupar com as questões que possa haver com a comunicação. Neste caso, todo o código do Transporter está configurado para ser gerado em linguagem C++ e posteriormente compilado em ficheiros DLL.

O modelo de domínio do Transporter é dividido em sete módulos: Contabilidade, Equipamento, Logística, Operação, Vendas, Embarque e Electronic Data Interchange (EDI). O módulo de Contabilidade é responsável pela gestão da faturação. O módulo de Equipamento é onde é gerido todo o tipo de equipamento, como por exemplo, gestão de sobre-estadias que se referem ao tempo excedido de um equipamento num determinado porto. O módulo de Logística permite a gestão de cargas. O módulo de Operação é responsável pelas viagens e embarcações. O módulo de Vendas permite a criação de cotações, gestão de contas e atividades Customer Relationship Management (CRM). O módulo de Embarcação é responsável pela gestão de transportes, processos de reserva e embarcações. Por fim, o módulo de EDI é transversal a todos os módulos referidos anteriormente onde é definida toda a documentação eletrónica que possa ser importada ou exportada. Existe um outro módulo que não é destinado para o utilizador, mas sim apenas para os administradores poderem fazer a gestão de utilizadores e definição de directórios para a geração de relatórios.

2.1 Casos de Uso

Nesta secção irei descrever três casos de uso com diferentes complexidades, por forma a se poder vir a avaliar diferentes alternativas de modernização do sistema.

2.1.1 Criar Conversão

A operação de criar uma conversão que tem como parâmetros de entrada os identificadores das seguintes entidades: Conta, Cliente, Embarcação e Viagem. Também como parâmetros de entrada temos uma data, a moeda e os valores de compra e venda desta moeda, sendo que apenas os últimos

três parâmetros (moeda, valor de compra e valor de venda) são obrigatórios para a criação de uma conversão (ver os campos a vermelho na figura 2.1). As conversões podem ser criadas para serem aplicadas em diferentes contextos, tendo apenas os campos obrigatórios preenchidos. Quando é dado um código de cliente como entrada, a conversão que é criada só irá ser aplicada para o cliente que corresponde a esse código. O mesmo é aplicado quando são preenchidos os outros dados de entrada para especificar em que contexto esta conversão deve ocorrer. Um exemplo de contexto é ter uma conversão aplicada só para um cliente ”X” em que apenas se preenche o campo ”Client Code” com o código respetivo desse cliente ”X” além dos campos obrigatórios. Assim, esta conversão só será aplicada em faturas que estejam associadas ao cliente ”X”. As entidades envolvidas nesta operação são: Conversão, Moeda, Conta, Viagem e Embarcação (ver figura 2.2).

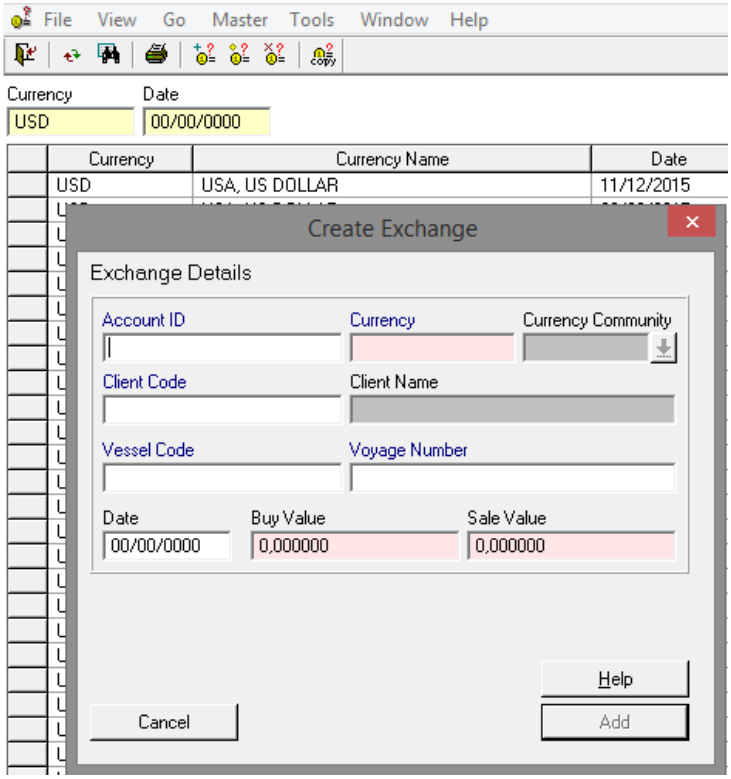


Figure 2.1: Painel de criação de uma nova conversão.

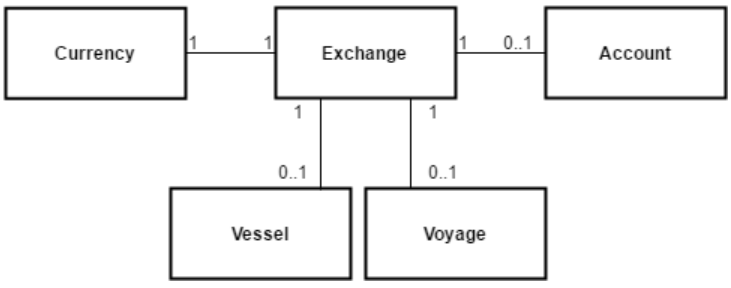


Figure 2.2: Relação das entidades envolvidas na criação de uma conversão.

2.1.2 Gerar Conhecimento de Embarque

Um conhecimento de embarque, normalmente tratado em inglês por BL, é um documento que descreve uma remessa de mercadoria. No Transporter, este documento é gerado automaticamente quando um

processo de reserva é criado ou quando o botão ”Regenerate” é pressionado, como se demonstra na figura 2.3, para que os dados do BL sejam atualizados. Este caso de uso tem alguma complexidade no sentido em que para que este documento seja gerado, é necessário obter informação por parte de várias entidades. As entidades envolvidas nesta operação são: Processo, BL, Embarque, Detalhe de Embarque, Cláusula de Transporte, Detalhe de Cláusula de Transporte, Viagem, Embarcação e Equipamento (ver figura 2.4).

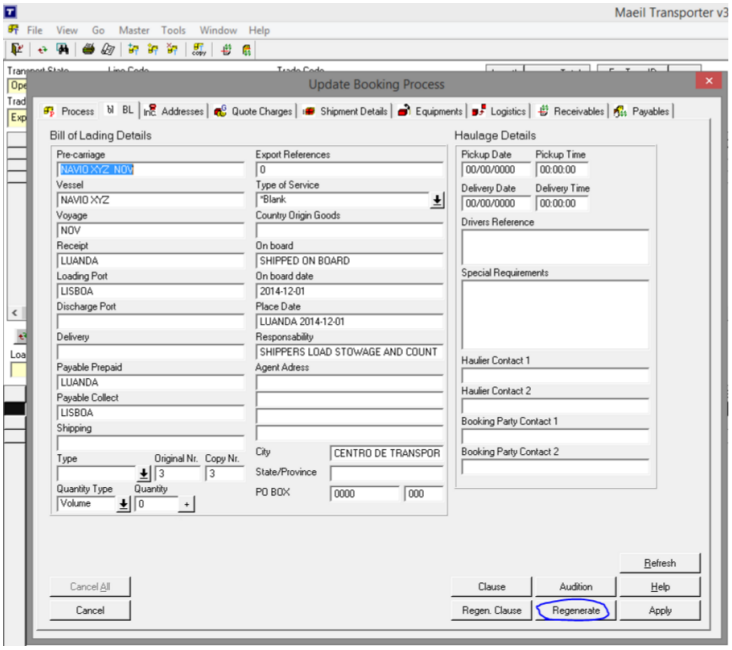


Figure 2.3: Separador BL de um processo de reserva

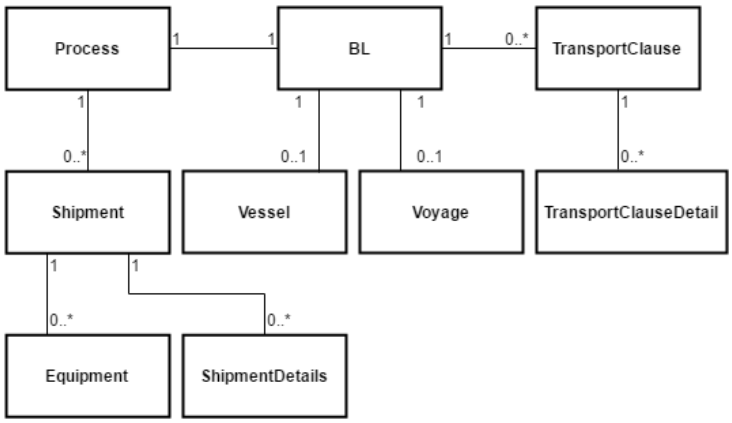


Figure 2.4: Relação das entidades envolvidas na geração de um BL.

2.1.3 Mover Processo de Reserva

Esta operação permite mover processos de reserva entre viagens ou períodos de tempo. Uma vez que o botão ”Move” é pressionado (ver figura 2.5), são realizadas certas validações no lado do cliente e posteriormente é invocada uma função servidora que é responsáveis pelo movimento de processos. Este função realiza outras validações, como por exemplo, ver se o processo cujo número de BL já existe na viagem para onde se pretende mover e se existem faturas ou serviços já realizados nesse mesmo

processo. Uma vez que as validações sejam bem sucedidas, o processo é inserido na nova viagem e eliminado da anterior. As entidades associadas são: Processo, Morada, Recebimento, Detalhe de Recebimento, Pagamento, Detalhe de Pagamento, BL, Equipamento, Movimento, Cotações, Embarque e Detalhe de Embarque (ver figura 2.6).

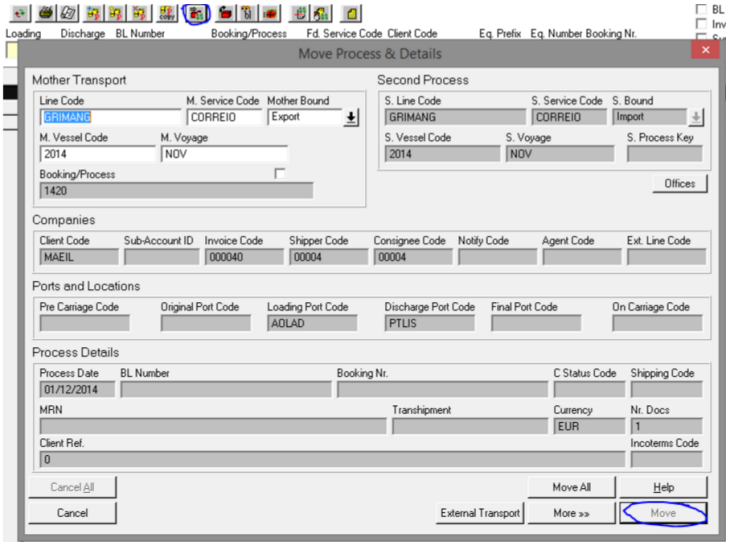


Figure 2.5: Janela de Mover Processo.

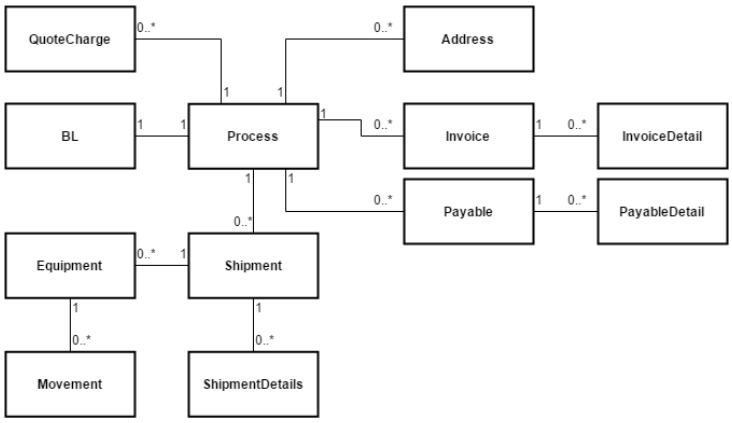


Figure 2.6: Relação das entidades envolvidas para mover um processo.

2.2 Arquitetura Atual

Nesta secção vou explicar qual a arquitetura atual do Transporter.

A atual vista de modelo de decomposição (figura 2.7) mostra como o Transporter é dividido em três módulos: Cliente, Servidor e Base de Dados. No lado do cliente, temos os ficheiros PNL que são os painéis de interface de utilizador e o código que suporta esses painéis em ficheiros cliente DLL. Os ficheiros locais (.INI), existentes nos módulos cliente e servidor, são usados para configuração de diretórios. No lado servidor é onde se encontra a maior parte da lógica de negócio nos ficheiros servidor DLL. O serviço de *Dispatch* gere aplicações em *run-time*, ou seja, é responsável por toda a comunicação que é realizada entre o lado cliente e servidor. No 3º módulo, de Base de Dados, encontram-se as tabelas, vistas, que servem de suporte aos dados que são registados e as *Table-*

Valued Functions que servem para realizar procuras de informação às tabelas/vistas existentes no Transporter.

A atual vista componente-conetor (figura 2.8), mostra como o Transporter coneta os seus módulos. A comunicação entre o cliente e servidor é tratada por Remote Procedure Call (RPC) que, por sua vez, é fornecida juntamente com o gerador servidor de *Windows C++*. RPC é uma interface de programação que permite realizar invocações de procedimentos remotos. Toda esta comunicação é invisível para o programador devido à forma como a aplicação é desenvolvida no CA Plex, tanto as funções cliente ou servidor são desenhadas automaticamente como RPCs. Desta forma, todas invocações remotas vão parecer como se fossem invocações locais. A comunicação com a base de dados *SQL Server* que é realizada com o lado servidor é através de um controlador Open Database Connectivity (ODBC).

Todas as funções geradas pelo CA Plex, à exceção de painéis, podem ser chamadas e controladas através de clientes controladores utilizando o protocolo *OLE Automation*. Este protocolo foi o primeiro mecanismo de integração Component Object Model (COM) fornecido para o CA Plex. Object Linking and Embedding (OLE) é o nome antigo utilizado pela Microsoft que foi substituído por COM. Este mecanismo permite que outros ambientes de desenvolvimento e outras linguagens de programação possam invocar funções CA Plex, havendo trocas de parâmetros *input* e *output*.

Uma possível integração com o Transporter, é através de dois ficheiros .EXE, *Ob600RS.exe* e *Ob600RC.exe*, que permitem a comunicação com o lado servidor e cliente respetivamente.

Na figura 2.9 temos a atual vista de alocação do Transporter. Toda esta aplicação é instalada numa máquina, neste caso, numa máquina virtual chamada *PAVA* em que é possível executar o Transporter nesta. Existe a possibilidade de ter mais computadores, existentes na mesma rede local, executar o Transporter sobre a mesma base de dados. Para tal, é necessário ter uma instalação local em cada computador tendo os ficheiros locais de configuração a apontar para os ficheiros DLL da máquina onde se encontra o *PAVA*, tendo as localizações desses ficheiros partilhadas pela rede.

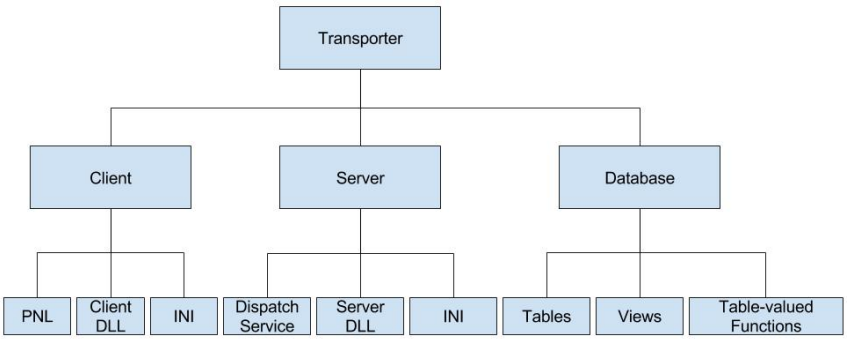


Figure 2.7: Atual vista Módulo de Decomposição do Transporter.

Seguem-se alguns exemplos dos diferentes ficheiros que existem nesta arquitetura. Na figura 2.10 temos um exemplo de painel que é guardado como ficheiro PNL. Este exemplo mostra como é que a interface dos detalhes de processo é desenhada no CA Plex. Na janela da esquerda, no *Panel Pallete*, podemos ver todas as variáveis existentes naquele painel (botões,campos,eventos e outras variáveis). Na janela do lado direito, encontramos as propriedades de um elemento específico da interface quando este é selecionado no próprio painel ou no *Panel Pallete*.

Na figura 2.11 temos um ficheiro local parcialmente. Este exemplo mostra a configuração para o lado cliente do Transporter saber onde se encontra o lado servidor, tendo lá definido, o nome da máquina onde este se encontra (*"System = PAVA"*), o protocolo (*"Protocol = NTRPC"*), o nome do programa (*"Program=PLEX_NT_DISPATCHER"*) e o nome do ambiente (*"Environment = Mtrans"*).

Na figura 2.12 temos um exemplo de código em CA Plex que é gerado em ficheiros C++ e compilado

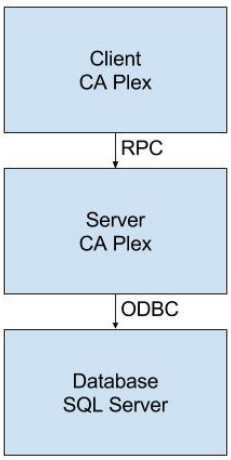


Figure 2.8: Atual vista Componente-conetor do Transporter.

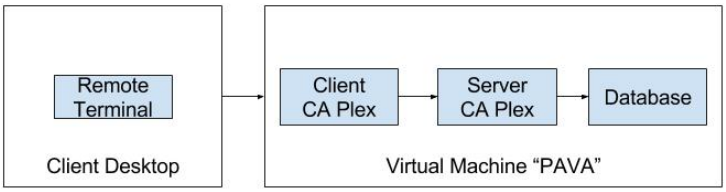


Figure 2.9: Atual vista de Alocação do Transporter.

para ficheiros DLL. Este código, presente na janela *Action Diagram*, é uma função cliente que cria um processo de reserva ao chamar a função servidora *”Create Instance”*. Se esta invocação for bem sucedida, a variável de *output* irá ter o valor que a função servidora devolve, senão irá ter como valor zero.

2.3 Como definir uma função em CA Plex

Nesta secção vou explicar como criar uma função em CA Plex.

Primeiro, precisamos de abrir o *Model Editor* que permite definir entidades, funções com os respetivos parâmetros de entrada, saída e variáveis locais. Na figura 2.13 podemos ver o modelo da função *API_CreateProcess*. Esta é uma função cliente, portanto precisamos de definir como *”Uibasic/-ClientExternal”* para que esta função seja considerada como cliente. O *input* é definido por uma vista da entidade de processo de reserva, portanto os parâmetros de entrada vai ser igual às colunas existentes na vista. Para implementar a lógica da função, é abrir a janela *Action Diagram* e usar o campo de entrada existente no topo (ver novamente a figura 2.12). Na janela *Diagram Pallete* vemos os todos os parâmetros de entrada (pasta *Input*), de saída (pasta *output*) e variáveis locais (pasta *Local*) que são usadas na função. Também se pode ver a lista de funções que são chamadas na pasta *Calls* e permite ver o *output* que cada uma dessas funções devolve. O tempo para implementar esta função foi cerca de cinco minutos tendo em conta que a vista usada como *input* já estava criada.

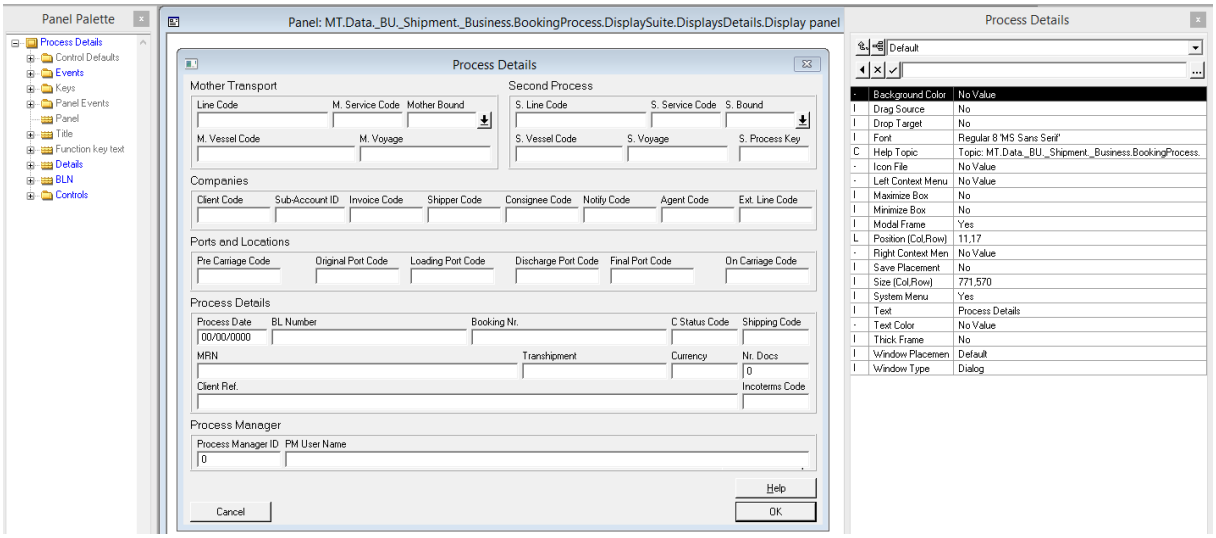


Figure 2.10: Exemplo de um painel

```
[RemoteWinNT]
System=PAVA
OpSys=WINNT
Protocol=NTRPC
Program=PLEX_NT_DISPATCHER
Environment=Mtrans
```

Figure 2.11: Exemplo de um ficheiro local.

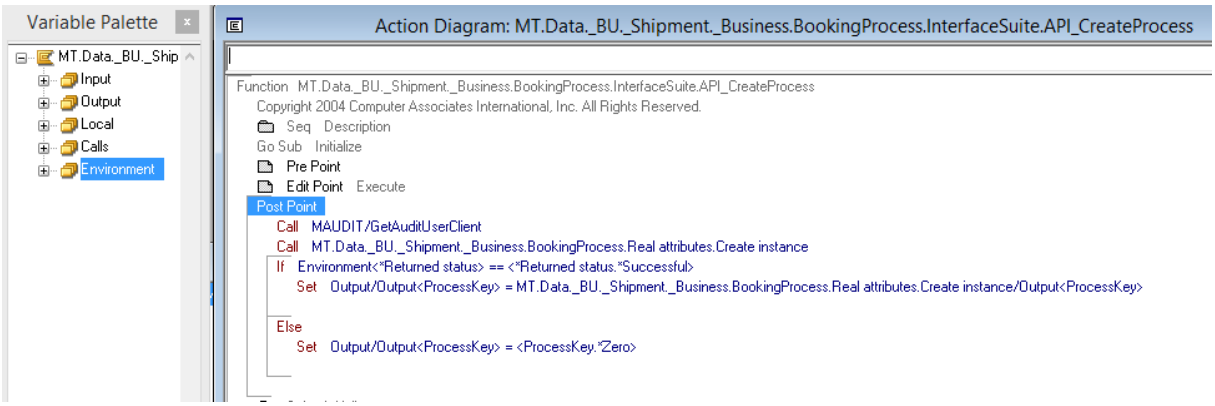


Figure 2.12: Exemplo de código Plex.

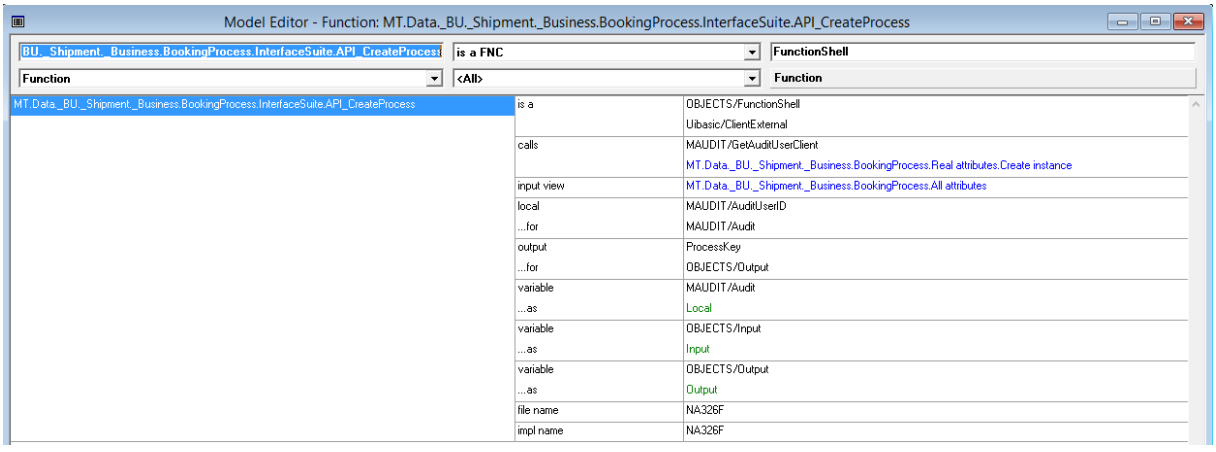


Figure 2.13: Modelo de criar processo.

Capítulo 3

Trabalho Relacionado

Nesta secção descrevo sobre sistemas legados[2], evolução de *software*, arquiteturas de *software* para a *web* e comunicação *web service*[3].

3.1 Evolução de Software

Nos dias de hoje a utilização de *software* é indispensável nos sistemas utilizados pela sociedade. Deste modo, existe a necessidade de manter estes sistemas atualizados e adaptáveis às mudanças constantes ao longo do tempo. Para tal, após o desenvolvimento de um *software*, este entra em fase de manutenção também conhecida pelo termo de Evolução de *software*.

Esta fase é bastante importante pois os requisitos por parte dos utilizadores tendem a evoluir e podem existir leis em termos legais que obrigam a alterações que tenham que ser realizadas para manterem certificações. Por exemplo, considerando um sistema de faturação, este está dependente de qualquer alteração que haja a nível de regras fiscais. Ou seja, se porventura alguma regra a nível fiscal for alterada, o *software* deverá ser atualizado de modo a acompanhar essa alteração.

Foram realizados estudos ao longo de trinta anos por Lehman juntamente com outras pessoas que deram em resultado oito leis que caracterizam a evolução de *software*[4][5]:

Lei da Modificação Contínua (1974) O *software* deve ser atualizado de acordo com novas necessidades que possam surgir para que mantenham a sua utilidade no mercado. Caso contrário, entrará em desuso e facilmente será substituído por outras soluções.

Lei da Complexidade Crescente (1974) Conforme são realizadas alterações a um *software*, este tende a adquirir uma maior complexidade o que irá aumentar os custos de manutenção. São necessárias alterações estruturais do código para que facilite manutenções posteriores.

Lei da Auto-Regulação (1974) Um *software* auto-regula-se quando não é preciso realizar muitas mudanças para que este se adapte a novas necessidades.

Lei da Conservação da Estabilidade Organizacional (1980) Ao longo do ciclo de vida de um *software*, o desenvolvimento deste deve ser independente dos recursos ou pessoas que estão envolvidas neste desenvolvimento.

Lei da Conservação da Familiaridade (1980) Um *software* e todos os outros que o integram, devem manter o domínio de modo que o comportamento entre eles seja igual.

Lei do Crescimento Contínuo (1980) Todo o *software* após fase de desenvolvimento, deve realizar alterações que não foram previstas nesta fase e de eventuais erros que possam comparecer para satisfazer os utilizadores.

Lei da Qualidade Declinante Modificação Contínua (1996) A qualidade do *software* tende a degradar-se à medida que são realizadas alterações fora da fase de desenvolvimento pelo que são necessárias práticas para ajudar a manter a qualidade.

Lei da Realimentação do Sistema (1996) Durante o ciclo de vida de um *software*, este é alimentado por *feedbacks*, positivos ou negativos, por parte dos utilizadores que facilita a o crescimento de um *software*.

Estas leis, se forem devidamente seguidas, permitem que um *software* tenha uma evolução saudável com o objetivo de satisfazer os utilizadores cumprindo todas as suas necessidades. Também é garantido que a qualidade deste seja mantida e que não fuja do seu domínio.

3.2 Sistemas Legados

Um *software* torna-se um sistema legado quando os custos de manutenção e alteração do sistema aumentam de forma não linear à complexidade e número de alterações. Sistemas legados que têm um elevado valor no mercado, devem ser modernizados ou substituídos. Há questões que devem ser respondidas para se poder tomar a decisão do que fazer a um sistema legado, por exemplo: Quanto custa substituir o *software*? E qual o custo para modernizar o mesmo? E qual a melhor forma de fazer essa substituição ou modernização? Responder a estas questões, ajudam no processo de decisão tendo em conta qual o rumo menos dispendioso quer a nível de tempo e/ou recursos para realizar a transformação. Normalmente a substituição só é realizada quando os custos de modernização não compensam. Existem diferentes abordagens para realizar uma modernização. É possível modernizar através de *screen scraping*, integração pela base de dados, integração por ficheiros ou criar *web services*.

Screen scraping é uma técnica que permite extrair informação da interface de uma aplicação. Existem ferramentas que implementam esta técnica que fazem leitura de dados numa interface de uma aplicação e permitem realizar ações como se fosse um utilizador. Estas ferramentas normalmente acarretam custos pelo que certas empresas preferem não optar por esta abordagem. A vantagem desta técnica é não haver a necessidade de realizar alterações à aplicação tendo como desvantagem o baixo desempenho e a dificuldade em simular um utilizador. O baixo desempenho é justificado por haver dependência do *software* legado onde são executadas operações.

A integração pela base de dados permite aceder à base de dados de um sistema legado para obter informação. Esta abordagem tem baixo custo visto que não existe a necessidade de reescrever aplicações e a maior parte dos Sistemas de Gestão de Base de Dados (SGBDs) fornecem controladores que permitem integrações. Por outro lado, existe a possibilidade de ter que transformar tipos de dados, não havendo as validações que são realizadas pela aplicação.

A integração por ficheiros é uma das formas mais utilizadas pois é universal para qualquer linguagem de programação e é fácil para os transferir com a utilização de protocolos. Um *software* que codifique dados para um ficheiro, por exemplo gerar uma fatura, permite que este possa ser descodificado por um outro *software*. A desvantagem desta abordagem é a limitação do desempenho e a complexidade de codificar e descodificar aumenta à medida que a informação se torna mais complexa.

Criar *web services* resume-se a encapsular funções existentes num sistema legado e expor as mesmas sob a forma de serviços. Esta abordagem permite a criação de uma nova interface evitando que a

lógica de negócio seja reescrita. Também cria a possibilidade de integração com sistemas terceiros. O desafio desta abordagem, é se existe ou não a possibilidade de expor as funcionalidades sob serviços e qual a melhor forma de o fazer.

3.3 Arquiteturas de Software para a Web

Um *software* com base numa arquitetura *web* é composto por dois principais componentes: cliente e servidor. Do lado cliente são realizados pedidos para o lado servidor, onde este realiza processamentos e enviam a resposta para o cliente. Para que haja uma interação entre o utilizador e o *software web* é necessário o uso de uma plataforma do lado do cliente que permita a realização dos pedidos. Um exemplo de uma plataforma é um *browser* onde tipicamente existe uma interface por onde o utilizador pode realizar os pedidos.

Arquiteturas *web* podem ter diferentes *tiers* (ver figura 3.1). Com duas *tiers*, podemos ter duas situações diferentes. Podemos ter uma aplicação onde temos uma interface, lógica de negócio e armazenamento de dados tudo junto . Ou podemos ter a camada de apresentação no cliente e a camada de lógica de negócio e a camada de gestão de recursos encontram-se na *tier* de servidor. Com três

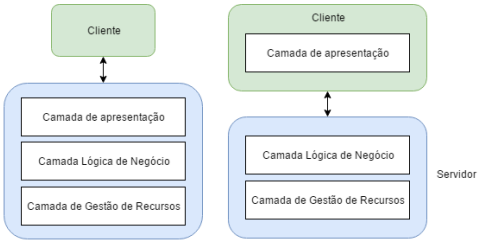


Figure 3.1: Duas situações de uma arquitetura de dois *tiers*.

tiers(ver figura 3.2), podemos ter também duas situações diferentes. Uma *tier* para a camada de armazenamento, outra para a camada de lógica de negócio e a camada de apresentação pode estar no cliente ou na mesma *tier* que a camada de lógica de negócio.

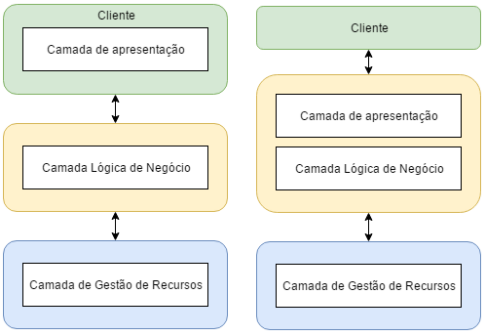


Figure 3.2: Duas situações de uma arquitetura de três *tiers*.

Hoje em dia, a utilização de arquiteturas *web* são uma forma de modernizar sistemas legados devido à funcionalidade orientada e à utilização de tecnologias padrão como Javascript Object Notation (JSON) sobre Hypertext Transfer Protocol (HTTP). Isto permite com que haja mais do que um cliente.

REST[6] é um estilo de arquitetura que consiste num conjunto coordenado de restrições aplicadas a componentes, conetores e dados tendo como propriedades, o desempenho e escalabilidade que são muito importantes para um *web software*. Por outro lado, temos SOAP que é um protocolo que permite a troca de informação estruturada por *web services* utilizando Extensible Markup Language (XML).

3.4 REST API

Os conceitos fundamentais de REST são referidos como recursos. Um recurso é um tipo de objeto com dados associados que pode ser relacionado com outros recursos e que é operado sob métodos padrão (*GET*, *POST*, *PUT* e *DELETE*) e representado como objeto JSON. Quando se tem vários recursos que podem ser agrupados, são tratados como uma coleção de recursos. Caso tenhamos um recurso fora de uma coleção, este é considerado como um recurso *singleton*. Cada coleção é homogénea e só pode ter um tipo de recurso e é modelada como um vetor de objetos. Os dados associados com os recursos são modelados como pares chave/valor. Quando uma API REST é desenhada, os pares chave/valor, são importantes para definir os recursos certos e para definir a granularidade certa para que se possa ter um comportamento correto e de fácil manutenção.

Definir a granularidade é um tópico muito discutido. Se tivermos operações Create, Read, Update and Delete (CRUD) numa API, irá tornar o uso e manutenção desta mais fáceis mas irá criar necessidade de executar várias invocações e os clientes têm que entender a lógica de negócio. Nestas operações CRUD, os clientes precisam de saber em como invocar os serviços da API, senão irá haver dados inconsistentes e a criação de problemas de manutenção. Por outro lado, tendo serviços expostos complexos na API, o número de invocações será menor e os clientes não precisam de entender a lógica de negócio. A dificuldade será maior no seu consumo por haver um maior número de dados a serem passados em cada serviço invocado.

Capítulo 4

Soluções Existentes

Nesta secção vou descrever sobre diferentes soluções para a modernização do Transporter. Existem três soluções possíveis para implementar a API: Serviços WCF gerados pelo CA Plex em que é descrito como gerar serviços WCF através do CA Plex e posteriormente compilados com a utilização de Visual Studio; Uma solução em que toda a lógica de negócio é reescrita para *stored procedures* e/ou *table-valued functions* com uma camada de serviços WCF ou REST implementada em Visual Studio; Uma solução API REST implementada, também em Visual Studio, integrada com o Transporter através de um ficheiro .exe fornecido pelo CA Plex que permite a comunicação com as funções existentes do Transporter. É importante referir que qualquer uma das soluções enumeradas acima, permitem com que outras interfaces de utilizador possam comunicar com o Transporter desde que tenham suporte a *web services*.

4.1 Arquiteturas das soluções

Nesta secção irei fazer uma descrição das arquiteturas de cada uma das soluções possíveis.

4.1.1 Serviços WCF gerados pelo CA Plex

A diferença arquitetural desta solução com a arquitetura atual descrita na secção 2.2 é que as funções do Transporter passam estar sob a forma de *packages* gerados pelo Plex. A estrutura da base de dados atual mantém-se e são adicionados dois novos módulos: Uma interface *web browser* e serviços WCF, também gerados pelo Plex (ver figura 4.1). A nova interface *web browser* irá comunicar com a API WCF que é gerada pelo Plex que contém os serviços para comunicar com as funções do Transporter (ver figura 4.2). A nível de alocação (ver figura 4.3), toda a parte servidora (API em Internet Information Services (IIS), funções em *packages* e base de dados) podem estar dentro da mesma máquina e um utilizador poderá aceder à aplicação tendo um *browser* para aceder à nova interface em qualquer máquina exterior.

4.1.2 Solução Centrada em Dados: Portal Transporter

Esta solução Portal Transporter descarta qualquer função gerada pelo Plex, apenas acedendo à base de dados Transporter diretamente e realizar operações sobre esta (ver figura 4.4). Este projeto feito em *ASP.NET* tem dois módulos: Cliente e Servidor. Do lado cliente, temos uma interface *web browser* e uma base de dados de utilizadores desta mesma interface. Do lado servidor, temos as *queries* que são realizadas sobre a base de dados do Transporter.

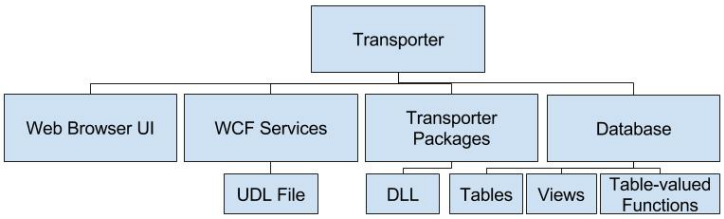


Figure 4.1: Vista de Decomposição da solução de serviços WCF.

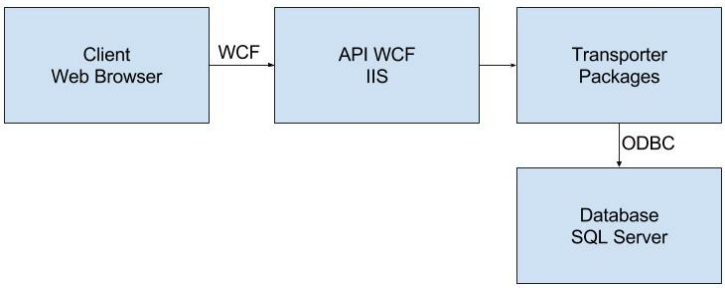


Figure 4.2: Vista Componente-Conetor da solução de serviços WCF.

A nível da vista componente-conetor (ver figura 4.5) e vista de alocação (ver figura 4.6), é semelhante comparativamente à solução referida na secção 4.1.1, em que os utilizadores terão uma interface web browser que comunica com a parte servidora. A diferença é por haver uma outra base de dados, onde se encontram os utilizadores da interface.

4.1.3 REST API integrado com o Transporter

Nesta secção vou explicar como é a arquitetura do Transporter integrado com uma API. Como é pretendido substituir a atual interface de utilizador, a diferença entre a arquitetura atual e esta, é que os ficheiros PNL são substituídos por uma REST API integrada com o Transporter. Esta integração permite ter uma interface *web browser* moderna que comunique com o Transporter através da API.

As diferenças entre o atual módulo de decomposição e o desta solução (ver figura 4.7), é a adição de dois módulos: a interface *web browser* e a REST API. No módulo ”Client”, os ficheiros PNL foram removidos. No módulo REST API temos controladores, modelos e um conjunto de classes a qual podemos chamar de ”Plex Engine”.

Na vista componente-conetor desta solução (ver figura 4.8), temos o ”Client Web Browser” a comunicar com a REST API utilizando o protocolo REST. Esta API está alojada num servidor IIS e comunica com a parte cliente e servidora do Transporter através uma aplicação COM e pode utilizar *TCP/IP* para comunicar com a base de dados *SQL Server*.

Na vista de alocação (ver figura 4.9), a API pode ser instalada na mesma máquina que o Transporter. Para se poder executar o Transporter de outras máquinas da mesma rede, só será necessário ter acesso a um *browser* para aceder à aplicação.

4.2 Descrição das soluções

Nesta secção irei fazer uma descrição da implementação de cada uma das soluções possíveis.

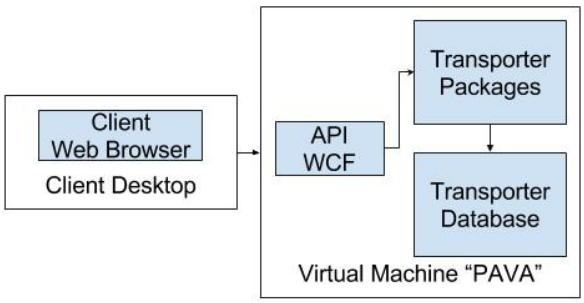


Figure 4.3: Vista de alocação da solução de serviços WCF.

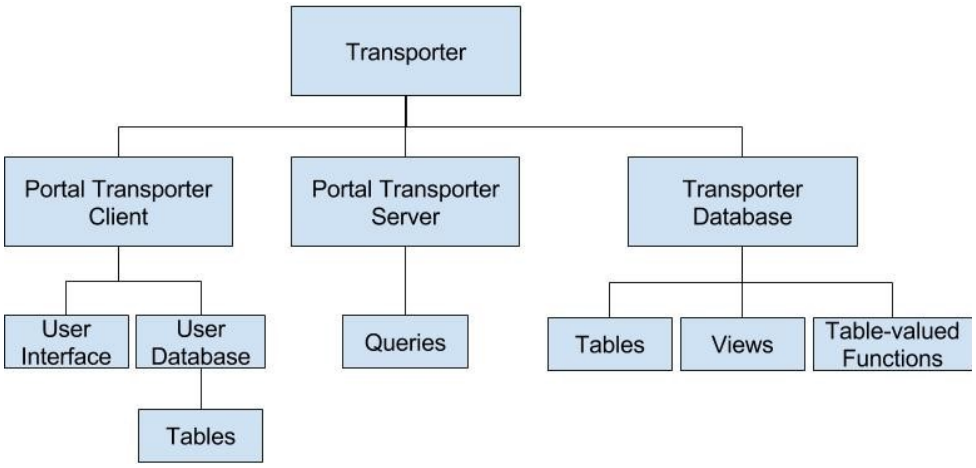


Figure 4.4: Vista de Decomposição Portal Transporter

4.2.1 Serviços WCF gerados pelo CA Plex

A ferramenta CA Plex permite a criação de serviços WCF que se resume uma a *framework* para construir aplicações orientada a serviços da Microsoft.

Como o Transporter é inteiramente desenvolvido em CA Plex, adotar esta solução não vai requerer com que código seja reescrito. A existência de um gerador de WCF, abre a possibilidade de expor funcionalidades do Transporter ao gerar serviços WCF em linguagem C#. Uma vez que o código é gerado, é necessário posteriormente realizar o *deploy* para um servidor IIS. Como o Transporter atualmente é gerado em C++ e os serviços WCF gerados em C#, temos a limitação em que as funções Transporter precisam de ser convertidas em *packages* para serem configuradas e geradas em C# (ver como implementar serviços na subsecção seguinte). A arquitetura desta solução é semelhante à da atual descrita na secção 2.2 tendo apenas a diferença de que temos um servidor IIS para disponibilizar os serviços que são gerados.

Como implementar serviços WCF em CA Plex

Os passos seguintes vão explicar como expor a função de obter um processo de reserva aplicando esta solução.

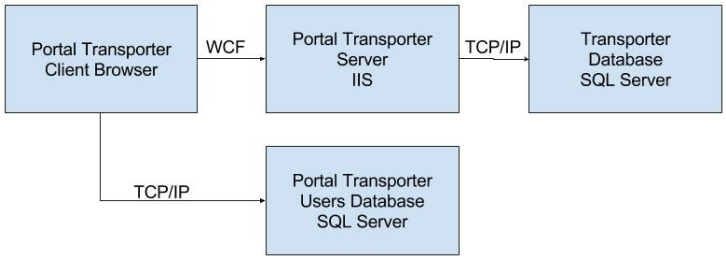


Figure 4.5: Vista Componente-Conetor Portal Transporter.

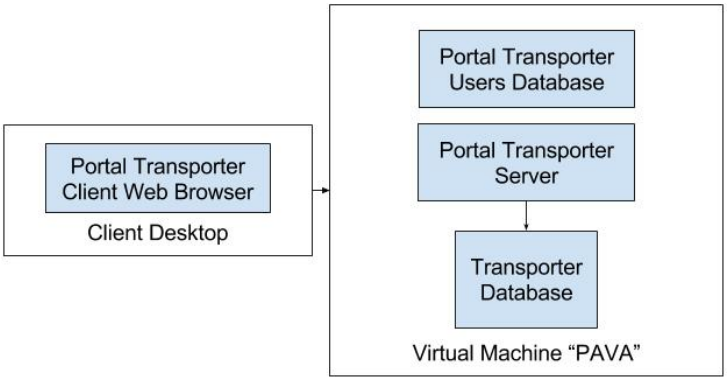


Figure 4.6: Vista de alocação Portal Transporter.

1 - Configuração Primeiro, precisamos de ter o *Storage Model* definido como ”.NET Server” (anteriormente estava ”NT ODBC Server”) no *Model Configuration* (ver figura 4.10). Isto irá permitir com que o CA Plex possa gerar qualquer função em C#.

2 - Criar um WCF *Package* Neste passo, criamos um *Package* (ver figura 4.11) com as seguintes caraterísticas:

1. *Service host* como IIS
2. *Type* como ”ServiceConnector”
3. *Language* como ”C#”
4. Adicionar um *Component* e uma *Interface* que passo a explicar nos próximos dois passos.

Este WCF *Package* vai ser responsável por gerar e compilar os serviços WCF em ficheiros C#.

3 - Criar *Component* Neste passo, definir um *Component*, de acordo como consta na figura 4.12, ter uma *Interface* como valor no ”implements”.

4 - Criar *Interface* Na *Interface* é onde se consegue adicionar os métodos que se pretendem que sejam expostos como serviços WCF (ver figura 4.13).

No *Object Browser*, após a criação do *Package*, do *Component* e da *Interface*, estes terão que estar de acordo com a figura 4.14.

Estes passos descritos até agora (do passo 1 ao 4), só é necessário serem executados uma vez.

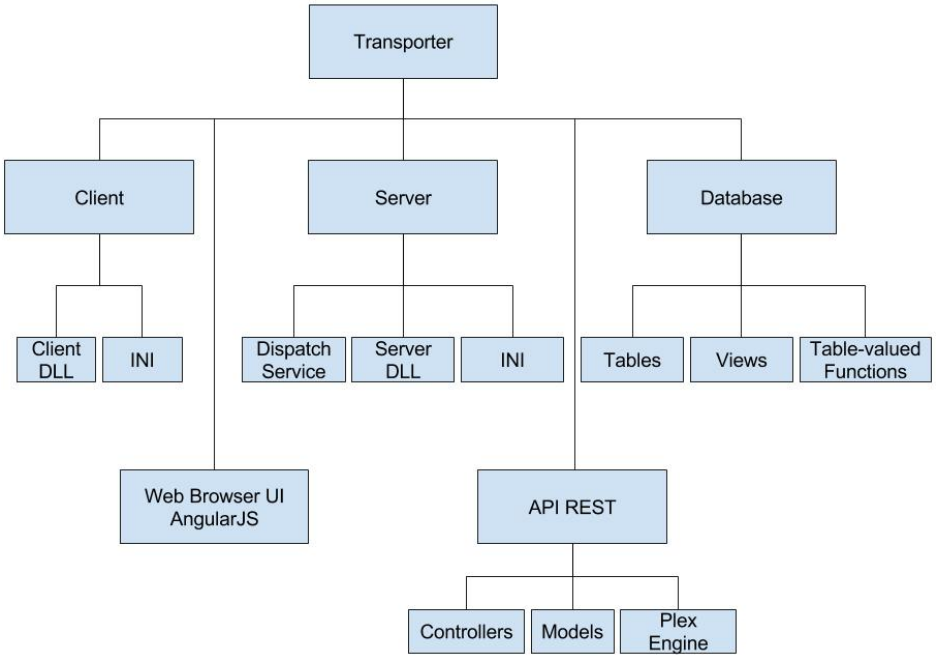


Figure 4.7: Vista de módulo decomposição.

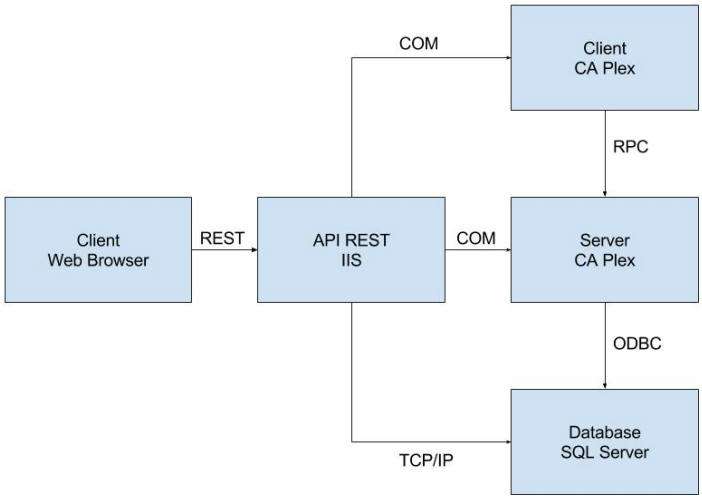


Figure 4.8: Vista componente-conetor.

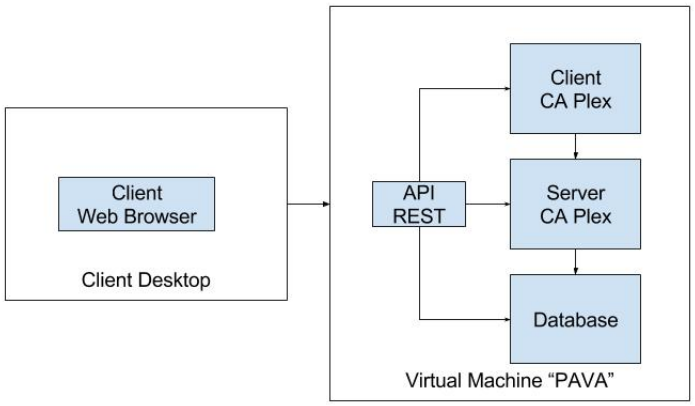


Figure 4.9: Vista de alocação.

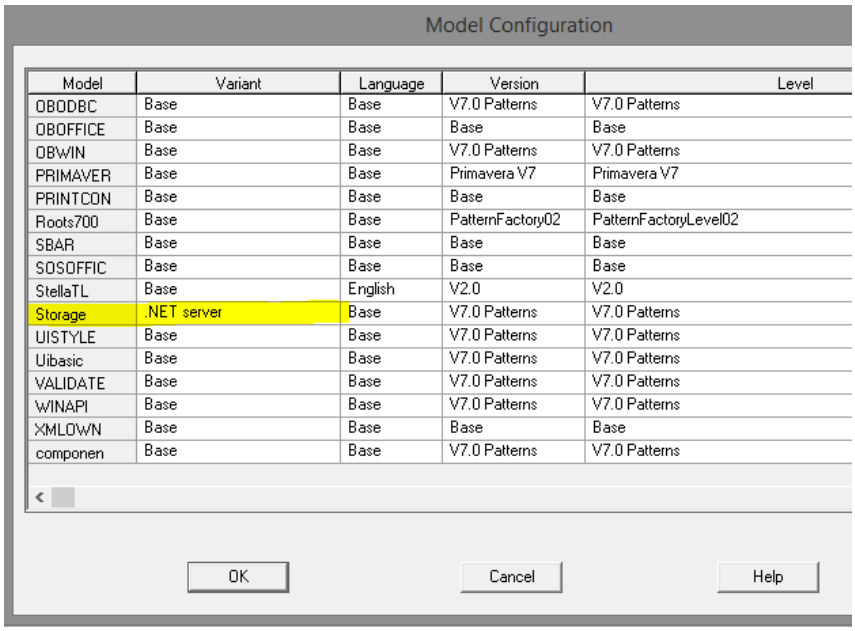


Figure 4.10: *Model Configuration*.

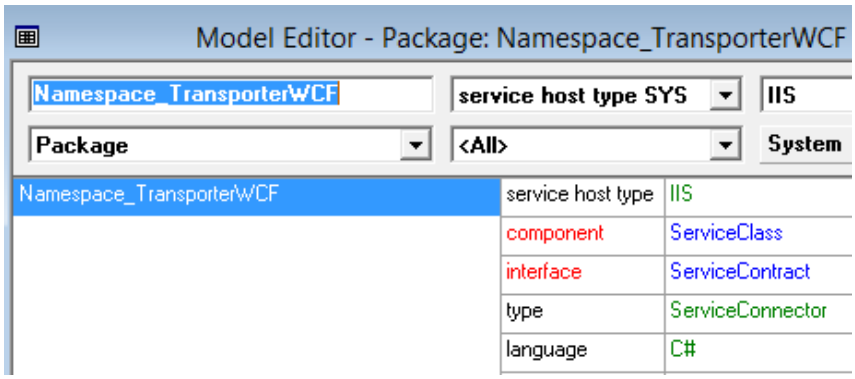


Figure 4.11: WCF *Package*.

5 - Criar Método Conforme a figura 4.13, criar o método e adicioná-lo à *Interface* criada anteriormente. Por forma que queiramos adicionar novos serviços para serem expostos, só é necessário um método por cada função que se queira expor. As funções que se encontram nos métodos, são funções encapsuladoras que invocam funções Transporter.

Segue abaixo como criar uma função encapsuladora.

Como implementar funções de encapsulamento para os serviços WCF em CA Plex.

Nesta secção descrevo como criar uma função encapsuladora para os serviços WCF.

Funções encapsuladoras são necessárias para que se possa definir os parâmetros de entrada e

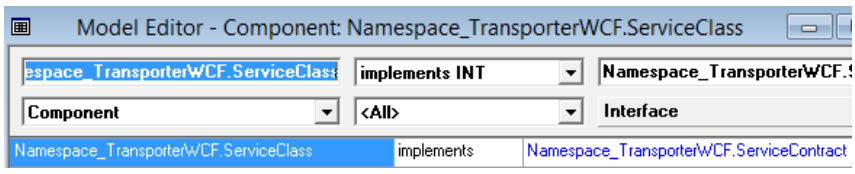


Figure 4.12: *Component*.

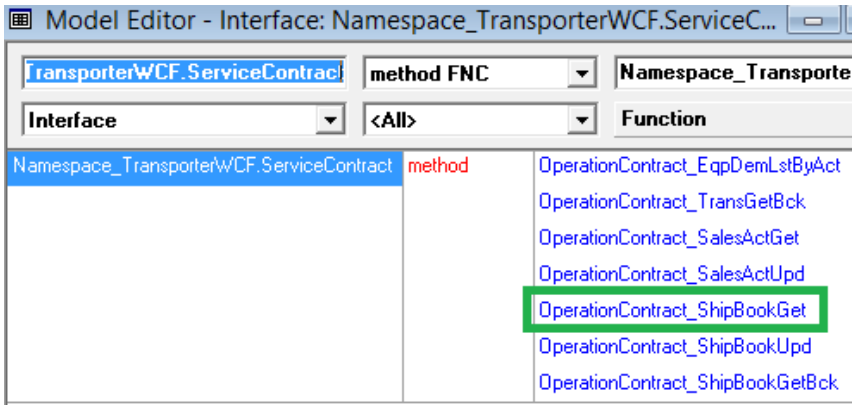


Figure 4.13: *Interface*.

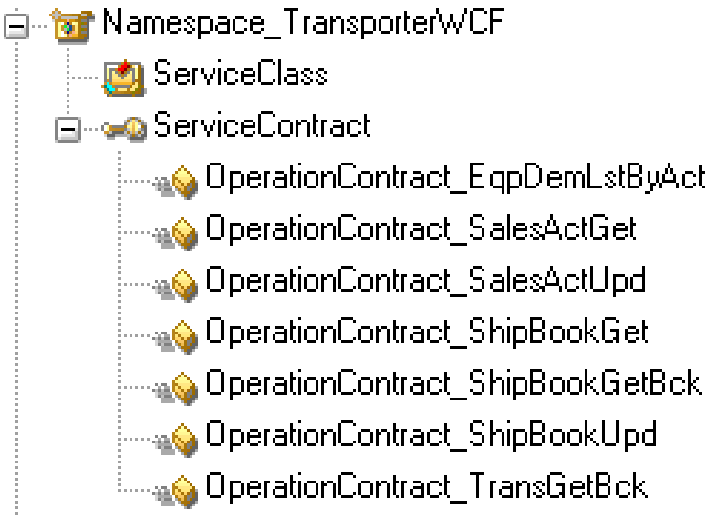


Figure 4.14: Vista *Object Browser* do *Package* WCF.

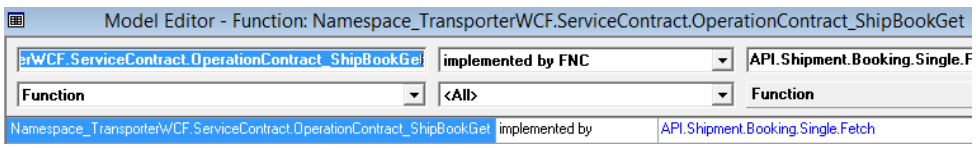


Figure 4.15: Método de obter um processo de reserva.

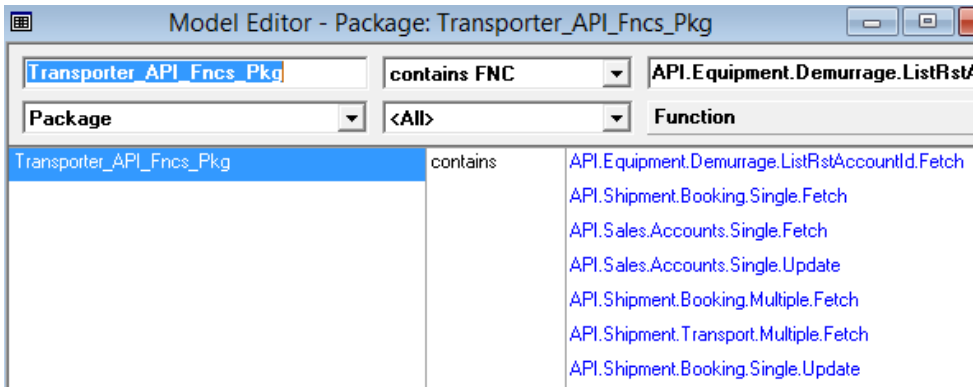


Figure 4.16: Exemplo do *Package*.

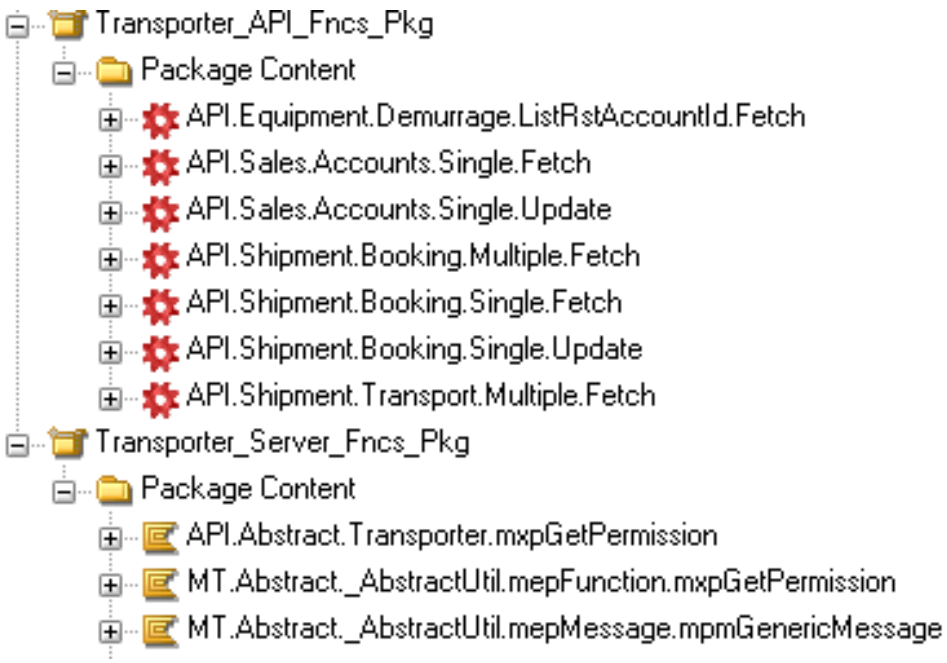


Figure 4.17: Vista dos dois *Packages* criados.

saída dos serviços e poder realizar validações antes de se invocar funções Transporter. A implementação das funções pode-se dividir nos seguintes passos:

1 - Criar *Packages* Criar dois *Packages*. Um para as funções encapsuladoras e o outro para as funções Transporter que são invocadas. Isto irá permitir gerar e compilar em ficheiros C# todas as funções de ambos os *Packages*. Estes *Packages* irão se apresentar no *Model Editor* como na figura 4.16 e no *Object Browser* como na figura 4.17.

2 - Criar *Code Library* De seguida, criar uma *Code Library* para que quando o código é gerado e compilado, uma solução de projeto Visual Studio seja criado. Isto irá permitir para realizar o *deploy* dos serviços WCF para um IIS.

A *Code Library* precisar de ter propriedades definidas como constam na figura 4.18 em que temos os *Packages* anteriomente criados como valores no ”comprises” e tem o *Package* WCF como valor de ”service”.

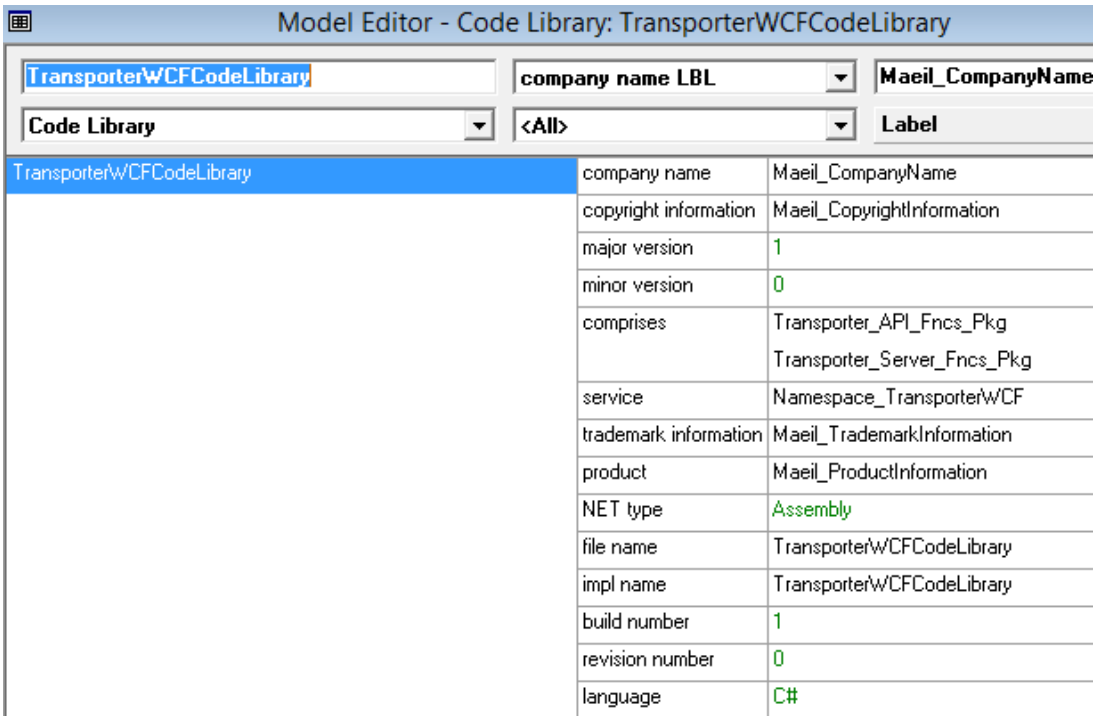


Figure 4.18: *Code Library*.

Figure 4.19: Vista da API *Entity* no Object Browser.

3 - Criar *Entity* Também é necessário criar uma *Entity* onde ficam localizadas as funções encapsuladoras mantendo a organização das entidades que existem no Transporter (ver figura 4.19).

Existe uma *Entity* "Abstract" onde irão ficar as funções que servem de auxílio para implementar as várias funções de encapsulamento.

Para manter a organização atual do Transporter, deve-se criar uma *Entity* (ver exemplo na figura 4.20) por cada entidade já existente no Transporter de modo que a API *Entity* fique similar com as entidades Transporter.

Um processo de reserva é uma entidade que pertence à entidade de embarcação portanto, neste caso, temos que criar uma sub-entidade Reserva dentro da entidade Embarcação.

Cada entidade tem uma interface onde podemos definir os parâmetros de entrada e de saída para serem usados quando os serviços são chamados.

4 - Criar função de encapsulamento Os três passos anteriormente descritos apenas necessitam de se realizar uma vez. Este passo serve para cada serviço que é criado para a API *Entity*.

Primeiro, cria-se uma função tendo no *Model Editor* como "calls" todas as funções Transporter que vão ser chamadas (ver exemplo na figura 4.21). Nesta função, podemos adicionar validações dos parâmetros de entrada, como a segurança, para que as funções Transporter só sejam invocadas

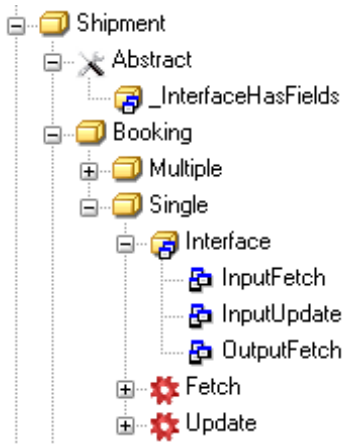


Figure 4.20: Vista da ”Shipment” *Entity* dentro da API *Entity* no Object Browser.

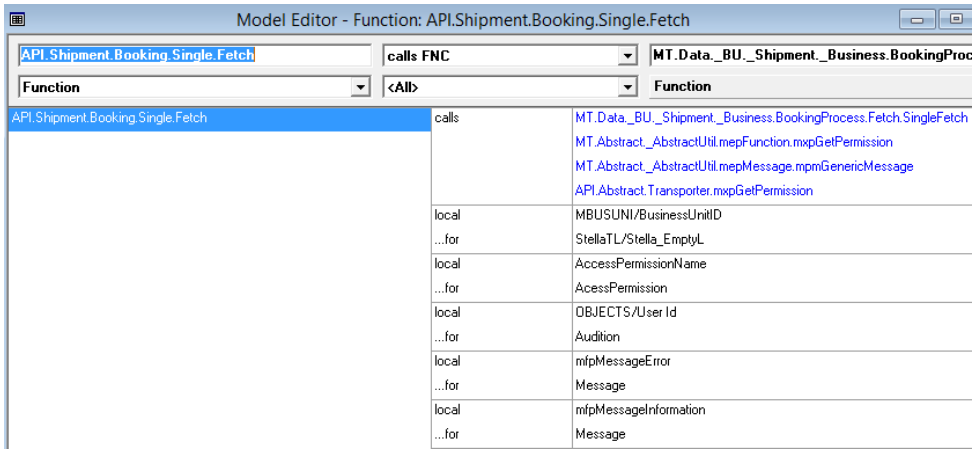


Figure 4.21: Função ”Booking Single Fetch” no *Model Editor*.

por utilizadores com permissões e/ou com os parâmetros corretos. De seguida, podemos definir os parâmetros de entrada e de saída do serviço (ver figuras 4.22 e 4.23). Por fim, passar os parâmetros de entrada na chamada à função Transporter e definir os parâmetros de saída com o resultado que se obtém desta chamada.

Segurança

O Transporter tem funções de validação para validar se os utilizadores têm permissão para aceder às funcionalidades. Por omissão, um utilizador tem acesso a todas as funcionalidades do Transporter exceto quando é criada uma regra para bloquear o acesso a uma funcionalidade específica pelo Trans-

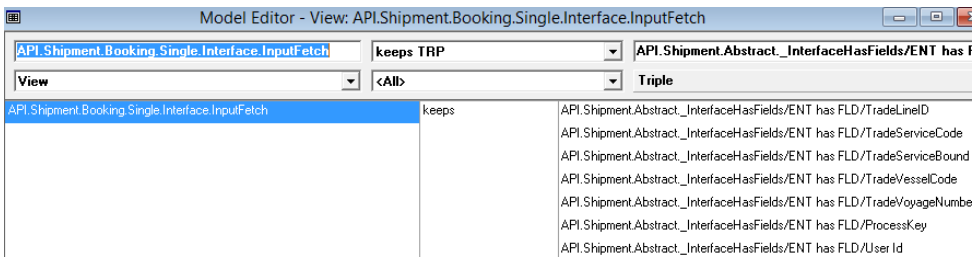


Figure 4.22: Parâmetros de entrada do serviço.

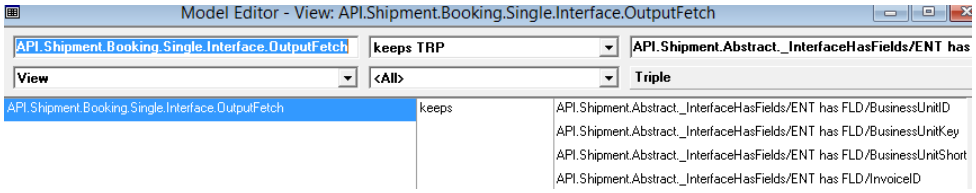


Figure 4.23: Parâmetros de saída do serviço.

porter AD.

Estas mesmas funções podem ser usadas para validar o controlo de acesso da API. É necessário criar uma validação nos serviços de modo que cada invocação realizada a estes, o utilizador que a realiza, é passado como um dos parâmetros de entrada obrigatoriamente. Caso este parâmetro não seja preenchido numa invocação, é retornada uma resposta a informar que não existe permissão para a mesma invocação. Desta forma consegue-se evitar que utilizadores sem permissões tenham acesso.

Um outro tipo de validação que pode ser implementado, por exemplo, é ter uma reserva que só pode ser alterada/atualizada pelo o utilizador que a criou. O código *Plex* presente na figura 4.24 mostra como é a implementação da validação quando se tenta realizar uma modificação numa reserva. Neste caso, se o parâmetro de utilizador estiver vazio ou se conter um utilizador que não foi o criador da reserva que se pretende alterar, irá ser retornada uma mensagem a informar de que não há acesso para realizar esta operação.

```
When  Stella_InputForBS<StellaTools.BusinessServices.Fields.InsertUpdateDelete> == <StellaTools.BusinessServices.Fields.InsertUpdateDelete.Update>
   Get booking process owner
   Call  MT.Data._BU._Shipment._Business.BookingProcess.Audition.Fetch.SingleFetch
   Check if user is the booking process owner
   If  Stella_InputForBS<AuditUserID> == MT.Data._BU._Shipment._Business.BookingProcess.Audition.Fetch.SingleFetch/FetchedData<AuditUserID>
      Get booking process
      Call  MT.Data._BU._Shipment._Business.BookingProcess.Fetch.SingleFetch
      Name  Function: MT.Data._BU._Sales._Business.Account.Fetch.SingleFetch, Environment<*Object>
      Go Sub  ST_CheckCall
      Set  Stella_UpdateDataL = MT.Data._BU._Shipment._Business.BookingProcess.Fetch.SingleFetch/FetchedData
      Set data to update
      Set  Stella_UpdateDataL = Stella_InputForBS
      Update
      Call  MT.Data._BU._Shipment._Business.BookingProcess.Update.UpdateRow
      Name  Function: MT.Data._BU._Shipment._Business.BookingProcess.Update.UpdateRow, Environment<*Object>
      Go Sub  ST_CheckCall
```

Figure 4.24: Validação de atualização de uma reserva.

4.2.2 Solução Centrada em Dados: Portal Transporter

Nesta secção é descrita como é implementada a solução que tem como base a plataforma Portal Transporter. Esta plataforma apenas comunica diretamente com a base de dados do Transporter para obter informação relacionada com cotações, reservas, faturas e rastreamento. Tem um lado servidor implementado em C# como serviço WCF alojado num servidor IIS. Este lado servidor é onde foram implementadas *queries* para colecionar os dados. Existe um lado cliente em *ASP.NET*, também alojado em IIS, onde tem uma interface de utilizador moderna implementada. Esta interface comunica com o lado servidor através dos seus *endpoints*.

Adotar esta abordagem, requer que toda a lógica de negócio do Transporter seja reescrita para *stored procedures*, o que não é viável a curto prazo, pois o tempo necessário de concretização é demorado devido à quantidade de lógica desenvolvida ao longo destes 17 anos. Mesmo não adotando esta solução, nova interface de utilizador pode-se basear na do Portal Transporter.

Exemplo de implementação

Nesta secção vou explicar como é implementado o obter um processo de reserva no Portal Transporter. No lado servidor, precisamos de definir dois *data contracts*, um para o processo de reserva e outro para os campos de filtração que são usados na pesquisa na interface (ver a listagem de código 4.1). Há dois *operations contracts*, um para obter os nomes das colunas da tabela de reserva da base de dados para se poder criar a tabela no lado da interface para quando é realizada uma pesquisa e o outro para obter os dados de procesos de reserva para preencher a mesma tabela.

A listagem de código 4.2 é a *query* que é executada quando o *operation contract* de obter os dados de processo de reserva é invocado. Nesta *query*, as tabelas de processo de reserva, embarcação, detalhes de embarcação e equipamentos de reserva são *joined* de modo que não só se obtém dados do processo de reserva em si mas também dados de outras entidades que estão relacionadas.

De seguida é só compilar o lado servidor e alojar num IIS.

A figura 4.25 mostra como é interface de pesquisa de processos de reserva no lado cliente. Existe um formulário com quatros campos obrigatórios para se poder fazer a pesquisa com os critérios que são passados (”Bill of Lading”, ”Container”, ”Start Date” e ”End Date”). Estes campos são passados depois para a *query* mencionada anteriormente.

No código é necessário adicionar a referência do serviço para se poder comunicar com o lado servidor (ver figura 4.26). O código na listagem 4.3 é executado quando o botão ”Search” presente na interface de utilizador é pressionado e invoca a função do lado servidor e gera a tabela na interface.

Listing 4.1: Serviço e *Data Contracts* de Processo de Reserva

```
namespace PortalTransporterWS {

  [ServiceContract]
  public interface IBookingSearch {
    [OperationContract]
    List<string> GetBookingFields();

    [OperationContract]
    List<BookingData> GetBookings(BookingSearchData searchData);
  }

  [DataContract]
  public class BookingSearchData {
    public BookingSearchData() {
    }

    [DataMember]
    public string ProcessBookingNumber { get; set; }

    [DataMember]
    public string ProcessBLNumber { get; set; }

    [DataMember]
    public DateTime BookingFromDate { get; set; }

    [DataMember]
    public DateTime BookingToDate { get; set; }

    [DataMember]
    public string Container { get; set; }

    [DataMember]
    public string ClientId { get; set; }
  }

  [DataContract]
  public class BookingData {
    public BookingData() {
      Data = new Dictionary<string, string>();
```

```
    }

    [DataMember]
    public Dictionary<string, string> Data { get; set; }
    }
  }
}
```

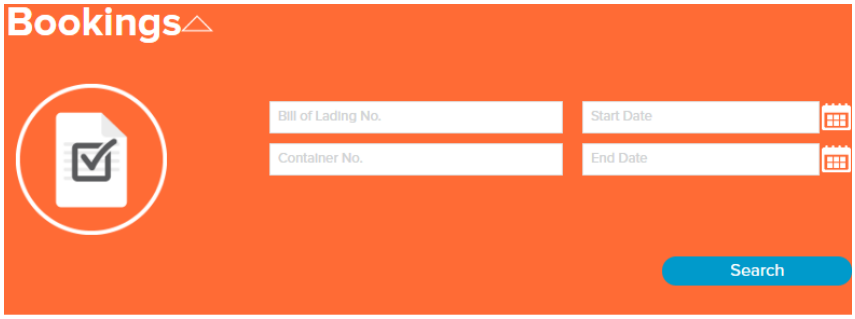
Listing 4.2: *Query* do Processo de Reserva na *Operation Contract*

```
select * from SHPROCSS as p
  inner join SHSHPMNT as s on p.TRDLNCOD = s.TRDLNCOD and p.TRDMTVCD = s.TRDMTVCD
  and p.TRDMTSBD = s.TRDMTSBD and p.TRDMTSCD = s.TRDMTSCD
  and s.TRDMTVNR = p.TRDMTVNR and s.PRCKEY = p.PRCKEY

  inner join SHSHPDTL as d on p.TRDLNCOD = d.TRDLNCOD and p.TRDMTVCD = d.TRDMTVCD
  and p.TRDMTSBD = d.TRDMTSBD and p.TRDMTSCD = d.TRDMTSCD
  and d.TRDMTVNR = p.TRDMTVNR and s.SHPKEY = d.SHPKEY
  and s.PRCKEY = d.PRCKEY and d.PRCKEY = p.PRCKEY

  left join SHSHPBKG as b on p.TRDLNCOD = b.TRDLNCOD
  and p.TRDMTVCD = b.TRDMTVCD and p.TRDMTSBD = b.TRDMTSBD
  and p.TRDMTSCD = b.TRDMTSCD and b.TRDMTVNR = p.TRDMTVNR
  and d.SHBKEY = b.SHBKEY and b.SHBPCKEY = d.PRCKEY

where p.DBSTATUS = 'E' and s.DBSTATUS = 'E' and d.DBSTATUS = 'E'
  and (b.DBSTATUS = 'E' or b.DBSTATUS is null)
  and (@client is not null and @client = p.ACCCLICD)
  and (p.PRCBLNBR = @bl or @bl is null)
  and (p.PRCDATE >= @sdate or @sdate is null)
  and (p.PRCDATE < @edate or @edate is null)
  and (p.PRCKEY in
    (select b.SHBPCKEY from SHSHPBKG b
    where ((b.EQPPRFIX+b.EQPNUMBR+b.EQPCHKDG) = @container or @container is null)))
```



TRDLNCOD	TRDMTSCD	TRDMTSBD	TRDMTVCD	TRDMTVNR	PRCKEY	LOCRCPCD	LOCLODC
ETE	MAR	E	2013	NOV	LISANR1301122		PTLIS
ETE	MAR	E	2013	NOV	LISANR1301122		PTLIS
ETE	MAR	E	2013	NOV	LISANR1301122		PTLIS
ETE	TERR	E	2013	NOV	LISOPO1301124		PTLIS
ETE	TERR	E	2013	NOV	LISOPO1301124		PTLIS
ETE	TERR	E	2013	NOV	LISOPO1301124		PTLIS
ETE	TERR	E	2013	NOV	LISOPO1301124		PTLIS

Figure 4.25: Procura de Reserva no Portal Transporter.

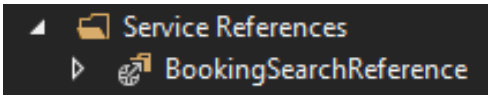


Figure 4.26: Referência do serviço da pesquisa de reserva adicionada ao lado cliente.

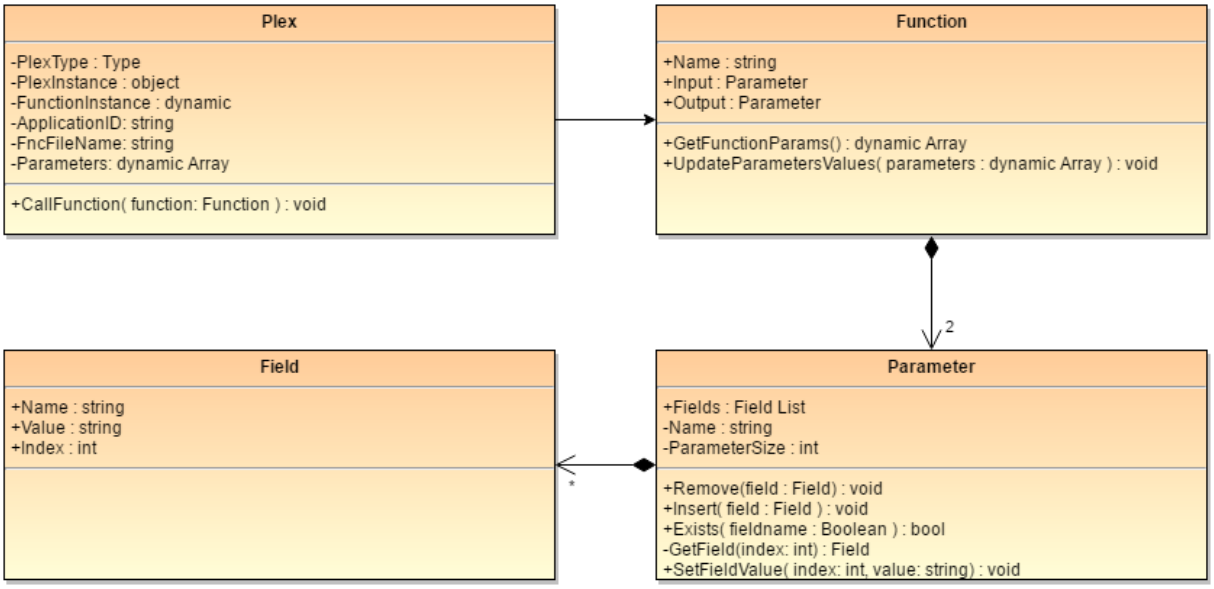


Figure 4.27: UML ”Plex Engine”.

Listing 4.3: Código do botão da pesquisa que é executado quando pressionado.

```
protected void SearchButton_Click(object sender, EventArgs e){

  BookingData[] response = new BookingData[0];
  BookingSearchClient searchClient = new BookingSearchClient("BasicHttpBinding_IBookingSearch");
  BookingSearchData searchData = generateSearchFilter();

  try {
    response = searchClient.GetBookings(searchData);
  } finally {
    searchClient.Close();
  }

  if (response.Length > 0) {
    BookingData booking = response[0];
    renderBookingFields(booking);
    renderBookings(response);

  } else {
    List<string> headerFields = new List<string>();
    searchClient = new BookingSearchClient("BasicHttpBinding_IBookingSearch");

    try {
      headerFields = searchClient.GetBookingFields().ToList();
    } finally {
      searchClient.Close();
    }
    renderBookingFields(headerFields);
  }
}
```

4.2.3 REST API integrado com o Transporter

Nesta secção irei descrever como é que a REST API pode ser implementada e integrada com o Transporter.

O módulo ”Plex Engine” é um conjunto de classes definidas na API que é responsável pela integração com o Transporter (ver o UML na figura 4.27).

Na classe ”Plex” é onde está definido a função(”CallFunction”) que chama os ficheiros .EXE, passando como argumento, um objeto do tipo da classe ”Function” que tem dois atributos do tipo ”Param-

eter" ("Input" e "Output") e cada um destes "Parameter" tem um ou mais objetos de "Field". Um objeto "Field" tem um nome, um valor e um índice que é a posição do "Field" nos parâmetros à chamada da função Transporter.

É possível invocar *stored procedures* ou executar *queries* diretamente na base de dados do Transporter através do *SQL Server*. Esta abordagem irá fazer com que a busca de informação seja mais eficiente do que chamar funções de busca do Transporter.

Esta solução é de baixa complexidade tendo uma integração fácil de realizar mas leva tempo para a implementar.

Estrutura de URL

Nesta solução, é possível mapear a estrutura funcional com uma estrutura desenhada de Uniform Resource Locator (URL) da API REST. Portanto, uma sugestão de estrutura de URL correspondendo aos módulos existentes no Transporter, seria:

Contabilidade ".../accounting"

Equipamento ".../equipment"

Logística ".../logistic"

Operações ".../operations"

Vendas ".../sales"

Embarcação ".../shipment"

EDI ".../edi"

Cada módulo do Transporter pode ser definido como coleção de recursos em que cada um destes irá ter sub-recursos. Esta estrutura é a menos específica que conseguimos ter. Para especificar um sub-recurso, por exemplo processos de reserva, ficaria definido como ".../shipment/bookingprocess" pois pertence à coleção da entidade de embarcação.

Existem relações entre sub-recursos de EDI e todos os outros sub-recursos da estrutura.

Dados os casos de uso referidos na secção 2.1, segue uma sugestão de URLs:

Criar conversão ".../accounting/exchange"

Gerar BL ".../shipment/bl/generate"

Mover processo ".../shipment/bookingprocess/move"

O primeiro URL é um recurso e os outros dois seguintes são processos de negócio. Como temos diferentes tipos de operações (recursos e processos de negócio), surge a questão de como é que podemos definir a estrutura de URLs se temos operações de diferentes granularidades. Uma possível maneira [7] é definir recursos como substantivos e os processos de negócio como verbos. Isto irá ajudar a identificar qual é o tipo de operação que cada URL.

Inserir um Processo de Reserva

Nesta secção vou explicar todo o procedimento que é realizado para inserir um processo de reserva. Na figura 4.28 é mostrado um formulário básico feito em *AngularJS* com *Bootstrap*. Este formulário permite realizar um pedido para inserir um processo de reserva.

Ao preencher os campos: ”Business Key”, ”User ID”, ”Trade Line ID”, ”Trade Service Code”, ”Trade Bound”, ”Vessel Code”, ”Voyage Number” e ”Process Date” e no fim clicar no botão ”Add Booking Process”, um pedido irá ser realizado. Por debaixo do formulário, temos uma janela de *debug* onde mostra o *input* (variável ”bookingProcess”) que é dado no formulário sob formato JSON e o *output* (variável ”result”) que se obtém da resposta do pedido. JSON é um formato de mensagem conhecido usado em comunicação cliente/servidora, em que aplicando a este caso, o cliente é o formulário no *browser* e o servidor é o Transporter conjuntamente com a API.

O código da listagem 4.4 é executado quando o botão do formulário é pressionado. Este código realiza um pedido POST para o URL ”.../api/bookingprocess” passando a variável ”bookingProcess” como ”data” e tendo definido no cabeçalho do pedido o ”Content-Type” como ”application/json”. Este pedido irá invocar a função presente na API que cria um processo de reserva no Transporter, tendo o *input* (variável ”bookingProcess”) mapeada para o modelo ”BookingProcess”(ver o código 4.5).

De seguida, este modelo de processo de reserva é passado como parâmetro para o controlador (ver o código 4.6).

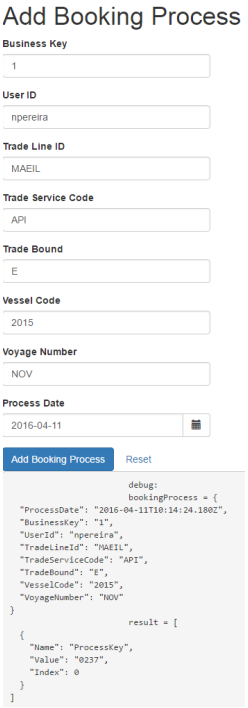


Figure 4.28: Formulário para adicionar um processo de reserva.

Listing 4.4: Pedido POST de um processo de reserva

```
$http({
  method: 'POST',
  url: 'http://localhost:60690/api/bookingprocess',
  data: $scope.bookingProcess,
  headers: {'Content-Type': 'application/json'}
})
```

Listing 4.5: Modelo do processo de reserva

```
public class BookingProcess {
```

```
  public string BusinessKey { get; set; }
  public string UserId { get; set; }
  public string TradeLineId { get; set; }
  public string TradeServiceCode { get; set; }
  public string TradeBound { get; set; }
  public string VesselCode { get; set; }
  public string VoyageNumber { get; set; }
  public string ProcessDate { get; set; }
}
```

Listing 4.6: Controlador do método POST de um processo de reserva

```
[ResponseType(typeof(List<Field>))]
public IHttpActionResult Post(BookingProcess bp){

  myLog.Source = "CargoAPI_InsertBookingProcess";
  Plex plex = null;

  try{
    plex = new Plex(); /* Initialize Plex COM */
  } catch (Exception e){
    myLog.WriteEntry("Falha ao chamar o ob600: " + e);
  }

  Function fnc = null;
  try{
    fnc = new Function("NA326F", 44, 1); /* Set Function Name, Input Size, Output Size */
  } catch (Exception e){
    myLog.WriteEntry("Falha ao encontrar a funcao " + e);
  }

  // Set Input Fields
  Field BusinessKey = new Field { Name = "BusinessKey", Index = 1, Value = bp.BusinessKey};
  Field UserId = new Field { Name = "AccessUserId", Index = 4, Value = bp.UserId};
  Field TradeLineId = new Field { Name = "TradeLineId", Index = 7, Value = bp.TradeLineId};
  Field TradeServiceCode = new Field { Name = "TradeServiceCode", Index = 8, Value = bp.TradeServiceCode};
  Field TradeBound = new Field { Name = "TradeBound", Index = 9, Value = bp.TradeBound};
  Field VesselCode = new Field { Name = "VesselCode", Index = 10, Value = bp.VesselCode};
  Field VoyageNumber = new Field { Name = "VoyageNumber", Index = 11, Value = bp.VoyageNumber};
  Field ProcessDate = new Field { Name = "ProcessDate", Index = 43, Value = bp.ProcessDate};

  // Insert Input Fields -----------------------
  fnc.Input.Insert(BusinessKey);
  fnc.Input.Insert(UserId);
  fnc.Input.Insert(TradeLineId);
  fnc.Input.Insert(TradeServiceCode);
  fnc.Input.Insert(TradeBound);
  fnc.Input.Insert(VesselCode);
  fnc.Input.Insert(VoyageNumber);
  fnc.Input.Insert(ProcessDate);

  // Set Output Fields -------------------
  Field ProcessKey = new Field { Name = "ProcessKey", Index = 0 };

  // Insert Output Fields ----------------------
  fnc.Output.Insert(ProcessKey);

  //Call Plex Function -------------------
  try{
    plex.CallFunction(fnc);
    return Ok(fnc.Output.Fields);
  } catch (Exception e){
       myLog.WriteEntry("Called function falhou " + e);
  }
}
```

Dentro do controlador, uma instância de Plex é criada (linha ”plex = new Plex()”). De seguida, definimos qual a função a que queremos chamar do Transporter que consta no DLL ”NA326F” que corresponde à função ”API_CreateProcess” referida como exemplo na figura 2.12 da secção 2.2. De

seguida, criamos e inserimos os parâmetros de entrada da função chamada dando os dados recebidos do cliente. Os índices de cada parâmetro é definido conforme a posição que este tem na função Transporter. O campo de *output* que é definido é o ”ProcessKey” que é o identificador do processo que é criado na inserção. Por fim, é chamada a função ”CallFunction”, definida na classe ”Plex” (ver código 4.7), passando o objeto ”Function” que foi criado para o DLL ”NA326F”.

Listing 4.7: Classe Plex

```
class Plex{
  private Type PlexType { get; set; }
  private object PlexInstance { get; set; }
  private dynamic FunctionInstance { get; set; }

  private string ApplicationID { get; set; }
  private string FncFileNme { get; set; }
  private dynamic[][] Parameters { get; set; }

  public Plex(){
    //Transporter's client functions
    this.ApplicationID = "ObRunOLE600RC.App";
    //Transporter's server functions
    //this.ApplicationID = "ObRunOLE600RS.App";

    this.PlexType = Type.GetTypeFromProgID(this.ApplicationID);
    this.PlexInstance = Activator.CreateInstance(this.PlexType);
  }

  public void CallFunction(Function function){
    string FncImplNme = function.Name;
    string FncFileNme = function.Name + ".dll";
    dynamic[] FuncParams = new dynamic[] { FncFileNme, FncImplNme };

    this.Parameters = function.GetFunctionParams();
    ParameterModifier pf = new ParameterModifier(2);
    pf[0] = true; pf[1] = true;
    ParameterModifier[] ModFParams = { pf };

    this.FunctionInstance = this.PlexType.InvokeMember("Function", BindingFlags.InvokeMethod |
    BindingFlags.Public | BindingFlags.NonPublic, null, this.PlexInstance, FuncParams, ModFParams, null, null);

    ParameterModifier pc = new ParameterModifier(2);
    pc[0] = true; pc[1] = true;
    ParameterModifier[] ModCParams = { pc };

    this.PlexType.InvokeMember("Call", BindingFlags.InvokeMethod | BindingFlags.Public |
    BindingFlags.NonPublic, null, this.FunctionInstance, this.Parameters, ModCParams, null, null);

    function.UpdateParametersValues(this.Parameters);
  }
}
```

A função ”Plex”, cria uma instância de ”ObRunOLE600RC.app” que corresponde ao ficheiro ”Ob600RC.exe”. A função ”CallFunction” recebe um objeto do tipo ”Function”, invocando a instância referida anteriormente passando esse objeto ”Function”. Uma vez que a instância é invocada, os parâmetros de *input* e *output* são atualizados ao chamar de seguida a função ”UpdateParametersValues”. O código 4.8 mostra como a classe ”Function” é implementada.

Listing 4.8: Classe Function

```
public class Function{

  public string Name { get; set; }
  public Parameter Input { get; set; }
  public Parameter Output { get; set; }

  public Function(string Name, int InputSize, int OutputSize){
    this.Name = Name;
```

```
    this.Input = new Parameter("Input", InputSize);
    this.Output = new Parameter("Output", OutputSize);
  }

  public dynamic[][] GetFunctionParams(){
    return new dynamic[][]{ this.Input.GetPlexParams(), this.Output.GetPlexParams()};
  }

  public void UpdateParametersValues(dynamic[][] parameters){
    for (int i = 0; i < parameters[0].Length; i++){
      this.Input.SetFieldValue(i, parameters[0][i]);
    }

    for (int i = 0; i < parameters[1].Length; i++){
      this.Output.SetFieldValue(i, parameters[1][i]);
    }
  }
}
```

A classe ”Function” é usada para definir as funções que queremos invocar do Transporter. Tem três atributos: um nome e dois ”Parameter” (*input* e *output*). O código 4.9 mostra como a classe ”Parameter” é implementada.

Listing 4.9: Classe Parameter

```
public class Parameter{

  public List<Field> Fields = new List<Field>();
  private string Name;
  private int ParameterSize = 0;

  public Parameter(string name, int size){

    this.ParameterSize = size;
    this.Name = name;

    for (int i = 0; i < ParameterSize; i++){
      Fields.Add(new Field { Value = "", Name = "", Index = i });
    }
  }

  public void Remove(Field field){
    Fields.Remove(field);
  }

  public void Insert(Field field){
    if (field.Index == -1) throw new System.FieldAccessException("Field␣index␣value␣must␣be␣set.");

    var fld = this.Fields.FirstOrDefault(i => i.Index == field.Index);

    if (fld != null){
      fld.Value = field.Value;
      fld.Name = field.Name;
    }else{
      throw new System.FieldAccessException("Field␣index␣value␣must␣be␣less␣or␣equal␣of␣Parameter␣type␣size.");
    }
  }

  public bool Exists(string fieldname){
    return this.Fields.Any(x=> x.Name == fieldname);
  }

  private Field GetField(int index){
    return this.Fields.FirstOrDefault(i => i.Index == index);
  }

  public void SetFieldValue(int index, string value){
    this.Fields.FirstOrDefault(i => i.Index == index).Value = value;
  }
```

```
  public dynamic[] GetPlexParams(){
    dynamic[] array = new dynamic[this.ParameterSize];

    for (int i = 0; i < this.ParameterSize; i++){
      array[i] = GetField(i).Value;
    }
    return array;
  }
}
```

Esta classe ”Parameter” serve para definir os ”Fields” de *input* e *output* das funções Transporter que queiramos invocar. No código 4.10 mostra a implementação de ”Field”.

Listing 4.10: Classe Field

```
public class Field{

  public string Name { get; set; }
  public string Value { get; set; }
  public int Index { get; set; }

  public Field(){

    this.Value = "";
    this.Name = "";
    this.Index = -1;
  }
}
```

Esta classe ”Field” serve para definir cada campo, de *input* ou *output*, dando o respetivo índice que corresponde à posição que aparece na função Transporter que é chamada.

Obter Processos de Reserva

Nesta secção vou explicar como obter todos os processos de reserva existentes no Transporter. O código 4.11 mostra como um pedido é feito para obter todos os dados de processos de reserva. O URL utilizado é o ”.../api/bookingprocess/” sob método GET em vez do POST que é utilizado na secção anterior, portanto, o controlador na API a ser executado vai ser diferente (ver código 4.12).

Para obter todos os dados, vamos diretamente à base de dados utilizando uma *framework* conhecida chamada *Entity Framework* da *Microsoft*. Esta *framework* permite realizar *queries* muito facilmente. Neste caso, realizamos uma *query* filtrando pela coluna ”DBSTATUS” igual a ”E” para que sejam retornados todos os processos de reserva que não tenham sido eliminados (ou seja, uma linha apagada no Transporter, fica com o ”DBSTATUS” a ”D”, de *disabled*).

Listing 4.11: Pedido GET do processo de reserva

```
$scope.getbookingprocesses = function(){
  $http.get('http://localhost:60690/api/bookingprocess/').success(function (data) {
  $scope.bookingProcessesCollection = data;
  });
}
```

Listing 4.12: Controlador GET do processo de reserva

```
public IQueryable<SHPROCSS> GetBookingProcesses(){
  return db.SHPROCSS.Where(p => p.DBSTATUS == "E");
}
```

Para que se possa comunicar com a base de dados do Transporter, é necessário definir também uma *connection string* no ficheiro ”Web.config” (ver código 4.13), onde se coloca o nome do servidor (”data source”) e a bases de dados a utilizar (”initial catalog”).

	Pros	Cons
Solution 1 (WCF Services by CA Plex)	- Fast functions exposure;	- Needs to convert the entire application to be generated in C#; - All Client DLL files must be rewritten/converted for Server DLL;
Solution 2 (Portal Transporter - data centric solution)	- Possibility to restructure/optimize Transporter;	- Needs to rewrite entire business logic;
Solution 3 (REST API integrated by Ob600RS.exe)	- Creation of a URL language which will identify each Transporter module; - Modern technology; - Easy functions exposure;	- All Client DLL files must be rewritten/converted for Server DLL;

Figure 4.29: Vantagens e desvantagens das soluções.

Listing 4.13: *Connection String*

```
<connectionStrings>
  <add name="BookingProcessDBModel"
  connectionString="data␣source=PAVA;initial␣catalog=Mtrans_GrimaldiAO;
␣␣integrated␣security=True;MultipleActiveResultSets=True;App=EntityFramework"
  providerName="System.Data.SqlClient"
  />
</connectionStrings>
```

4.3 Comparação das soluções

Apesar de ser a MAEIL que decide qual das abordagens optar, de acordo com as três soluções explicadas anteriormente, eu escolheria a solução 1 (ver figura 4.29) tendo em conta o tempo e recursos necessários para a implementação da API. A solução 2 não é viável a curto prazo para a empresa visto que é necessário reescrever toda a lógica de negócio desenvolvida durante 17 anos. Na outra solução, é necessário reescrever a lógica existente em funções clientes para funções servidoras. Adotando a solução 1, é necessário converter todo o Transporter para ser gerado em C# mas é possível expor funcionalidades mais rapidamente que as outras soluções.

Capítulo 5

Avaliação

Neste capítulo descreve-se uma avaliação de cada uma das abordagens e comparação entre estas.

5.1 Implementação

Para efeitos de avaliação, uma solução em *web browser* do módulo de CRM foi implementada adotando as abordagens descritas no capítulo 4. Como medição, teve-se em conta o tempo e o número de linhas necessárias para implementação em cada uma destas abordagens. A interface que foi implementada é igual para cada uma das abordagens apenas diferindo na forma como comunica com o Transporter.

5.1.1 Serviços WCF gerados pelo CA Plex

Nesta abordagem, foi implementada uma API que engloba serviços de CRM gerados pelo CA Plex. Também foi criada uma interface *web browser* em *.NET* que consome os serviços disponibilizados através da API.

Do lado da API, existem dois serviços, um para realizar uma operação *get* de uma atividade CRM e outro para realizar operações de *create, update e delete*. O tempo de criação do primeiro serviço, incluindo a estrutura mínima que forma a API, levou duas horas e 75 linhas de código *Plex*. O serviço de *create, update e delete*, levou três horas a implementar e 50 linhas de código *Plex*.

Do lado da interface, foi criada uma janela com uma tabela para disponibilizar os dados das atividades CRM e respetiva ligação à API para obter os dados. Esta implementação levou duas horas e 75 linhas de código *C#/.NET*. A operação de *update* de uma atividade levou quatro horas e 40 linhas de código. A operação de *delete* levou duas horas e 20 linhas de código. Por fim, a operação para inserir uma atividade CRM levou três horas e 40 linhas de código. No total, foi necessário 16 horas para implementar 300 linhas de código.

Para melhor interpretação dos valores medidos, os seguintes gráficos refletem as medições que foram tidas em conta durante a implementação. O gráfico 5.1 reflete o tempo que foi consumido na implementação e o gráfico 5.2 as linhas escritas.

Durante a implementação aplicando esta abordagem, a criação de serviços não é assim tão linear quanto se pensava inicialmente. Ou seja, há casos em que é preciso encapsular código existente em painéis em funções para que possam ser usadas através de serviços.

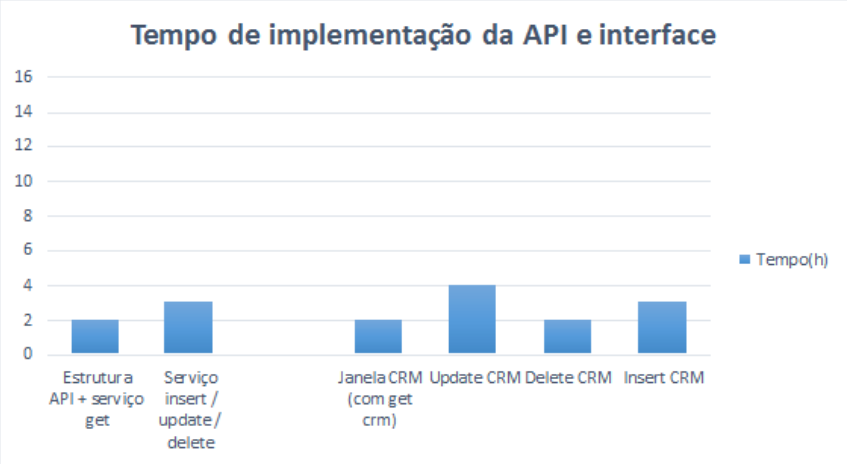


Figure 5.1: Tempo de implementação da API e interface.

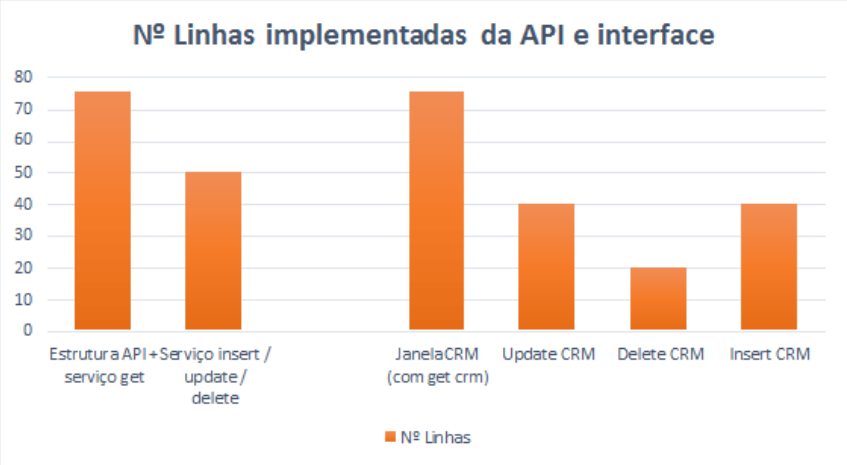


Figure 5.2: Número de linhas necessárias para construção da API e interface.

5.1.2 Solução centrada em dados

Nesta abordagem foi criada uma única solução em *.NET* com uma ligação a uma base de dados *SQL Server* contendo a interface *web browser* que opera diretamente sobre uma base de dados Transporter.

Esta solução demorou um total de duas horas com 35 linhas de código escritas. O gráfico 5.4 mostra o número de linhas que foram necessárias para a implementação de cada uma das tarefas e o gráfico 5.3 mostra o respetivo tempo necessário para as implementar.

O rápido desenvolvimento desta solução deve-se às ferramentas que o *.NET* dispõe. Foi utilizada uma *GridView* que é uma tabela que opera sobre uma tabela específica de uma base de dados e que gera automaticamente *queries* para que se possam realizar operações CRUD sobre a mesma tabela.

Pode-se assumir que esta abordagem, consegue ser bastante rápida em comparação com a anterior no que toca às operações CRUD e operações com alguma lógica de negócio simples. No caso de funções mais complexas, já poderá ser mais demorado pois é necessário a sua implementação de raiz e ter de conhecer o negócio.

5.1.3 REST API integrado com o Transporter

Nesta solução, foi necessário criar as funções clientes em *Plex* e a REST API em C# no lado servidor e a interface de utilizador em *.NET* no lado cliente.

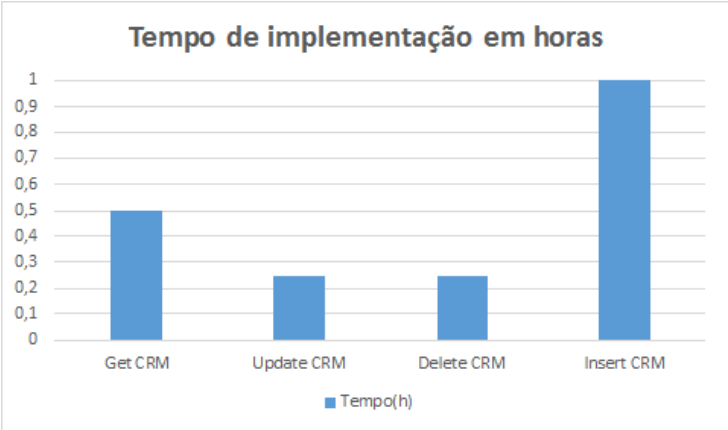


Figure 5.3: Tempo de implementação da API e interface.

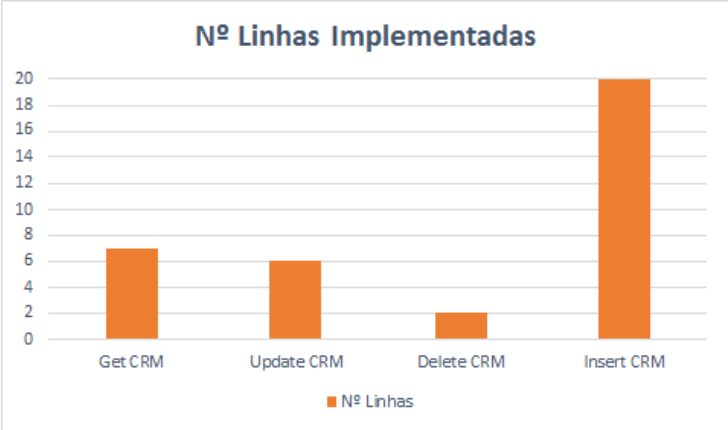


Figure 5.4: Número de linhas necessárias para construção da API e interface.

Esta abordagem levou quase 15 horas a implementar e 800 linhas de código foram escritas. Novamente, temos os gráficos que mostra o tempo e número de linhas escritas em cada um dos componentes desta solução. O gráfico 5.6 mostra o número de linhas que foram necessárias para a implementação de cada uma das tarefas e o gráfico 5.5 mostra o respetivo tempo necessário para as implementar.

Esta solução é a menos preferível, pois requer bastante código e torna difícil a sua manutenção.

5.2 Comparação de resultados

Nesta secção é realizada uma comparação dos resultados obtidos durante a implementação das 3 abordagens.

No gráfico 5.7 temos os tempos totais de implementação de cada uma das abordagens e no gráfico 5.8 temos o número de linhas escritas.

Com base nos gráficos apresentados, podemos concluir que a implementação da solução 1 foi a mais demorada e a solução 3 a menos demorada e com menos linhas de código escritas.

A solução 2 foi a que necessitou mais linhas de código em comparação com as outras soluções. Devido ao maior número de linhas, esta abordagem é a menos preferível pois é mais difícil de fazer a sua manutenção.

Os resultados da solução 3 justificam-se pelas operações CRUD que foram implementadas. Sendo

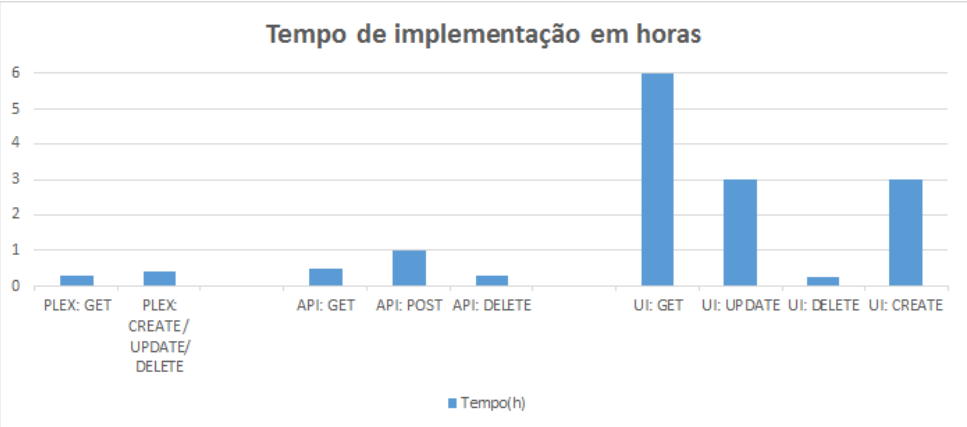


Figure 5.5: Tempo de implementação da API e interface.

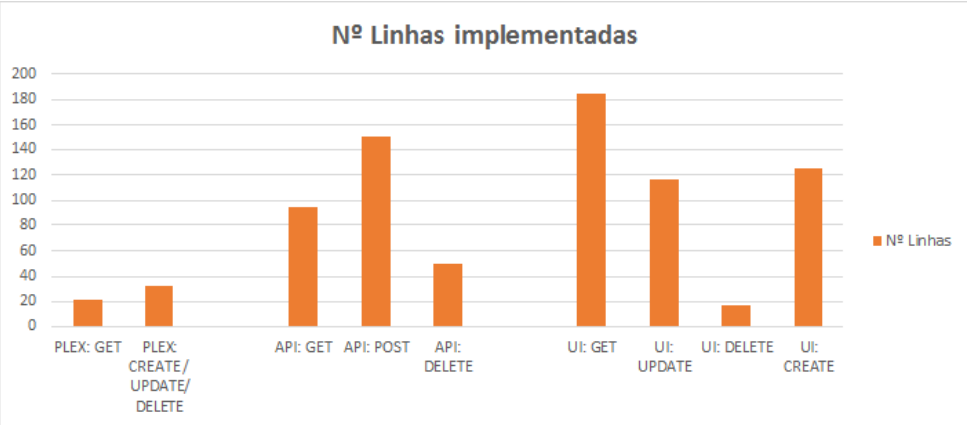


Figure 5.6: Número de linhas necessárias para construção da API e interface.

que outro tipo de operação que exija uma lógica de negócio mais complexa, esta abordagem não teria um desenvolvimento tão rápido pois exige com que seja reescrita lógica de negócio.

A solução 1, apesar de ter sido a mais demorada, uma vez que a estrutura base esteja implementada, o desenvolvimento desta abordagem torna-se mais rápido.

Na minha opinião, a curto prazo, seria optar por uma solução híbrida, que conjuga a solução 1 e 3. Nesta abordagem híbrida, irá permitir com que as operações CRUD sejam rapidamente implementadas e que não seja necessário reescrever lógica de negócio, utilizando a solução 1 para estes casos. A longo prazo, será implementar o Transporter do zero, utilizando uma tecnologia moderna para que esteja livre das limitações existentes no Plex.

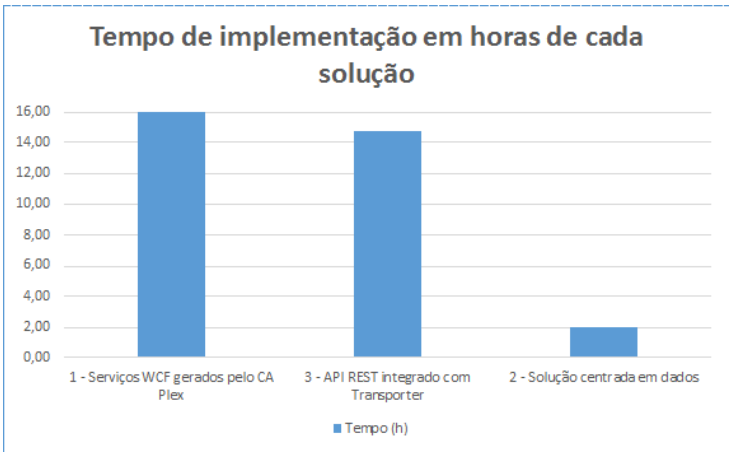


Figure 5.7: Comparação do tempo de implementação entre as 3 soluções.

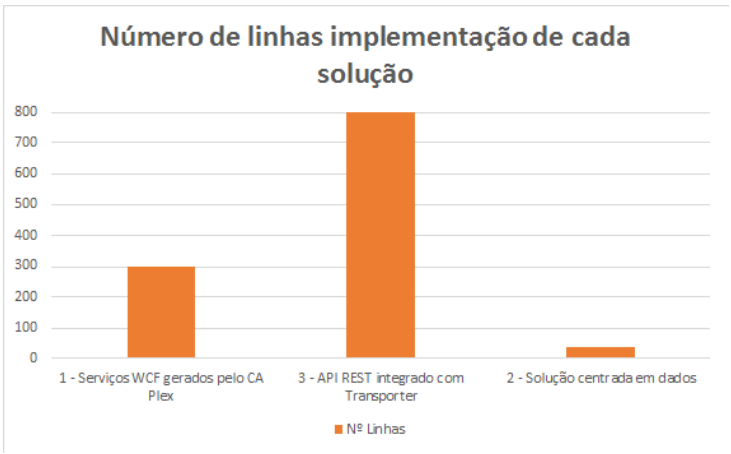


Figure 5.8: Comparação do número de linhas escritas entre as 3 soluções.

Capítulo 6

Conclusão

Neste documento é descrito o que é o Transporter que é um ERP focado na área de logística e transportes e é considerado um sistema legado. É um desafio para a MAEIL em modernizar o mesmo para que abra a possibilidade de desenhar e implementar uma nova interface *web browser*. A solução para se poder criar esta interface, é ter uma implementação de uma API. Esta abordagem permite aproveitar a funcionalidade desenvolvida ao longo de 17 anos e, desta forma, evitar que a lógica de negócio seja reescrita. Há três formas possíveis de construir a API que foram analisadas e comparadas entre si. É explicada a arquitetura atual e como é que cada solução integra com o Transporter.

Também é analisado como expor operações de diferentes granularidades que existem no Transporter em que operações simples são consideradas como substantivos e operações compostas são como verbos.

Será a MAEIL que terá a decisão final de qual das soluções optar de acordo com o tempo e recursos disponíveis. No entanto, a minha recomendação de qual das soluções optar, para longo prazo, será reimplementar o Transporter, de modo que o Plex deixe de limitar a evolução deste e permitir à MAEIL acompanhar o mercado a nível tecnológico. Adicionalmente, a utilização de novas tecnologias, irá permitir ser mais fácil encontrar pessoas especializadas.

A curto prazo, por exemplo para eventuais serviços sem a necessidade de uma interface, será utilizar a solução de serviços WCF gerados pelo Plex. Se houver a necessidade de ter uma interface a curto prazo, além optar pela solução dos serviços WCF que são gerados para as operações de maior complexidade, optar também pela construção de *stored procedures/queries* para operações mais simples, como CRUD. Adotar esta solução híbrida será a mais rápida e a própria aplicação terá um maior desempenho.

Para trabalho futuro, existe uma possível solução que possa ser explorada que é um gerador de REST por parte da CA. Este gerador faz parte do planeamento da CA como próximos desenvolvimentos. Adicionalmente, para total modernização do Transporter, será necessário definir quais os módulos deste a serem expostos e o desenho de interface de utilizador.

Bibliography

[1] D. R. L., "The encasement strategy: on legacy systems and the importance of apis." https://www.thoughtworks.com/insights/blog/rest-api-design-resource-modeling, 2014. [Acedido em: 2016-05-30].

[2] G. A. L. Robert C. Seacord, Daniel Plakosh, *Modernizing Legacy Systems: Software Technologies, Engineering Processes and Business Practices*. Addison-Wesley Professional, 2003.

[3] T. K. George Coulouris, Jean Dollimore, *DISTRIBUTED SYSTEMS, Concepts and Design, Fifth Edition*. Addison-Wesley, 2013.

[4] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proc. IEEE*, vol. 68, pp. 1060–1076, September 1980.

[5] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, "Metrics and laws of software evolution - the nineties view," in *Proceedings of the 4th International Symposium on Software Metrics*, METRICS '97, (Washington, DC, USA), pp. 20–, IEEE Computer Society, 1997.

[6] R. T. Fielding, "Representational state transfer (rest)." http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_2_1_1, 2000. [Acedido em: 2016-05-30].

[7] P. Subramaniam, "Rest api design - resource modeling." https://www.thoughtworks.com/insights/blog/rest-api-design-resource-modeling, 2014. [Acedido em: 2016-05-30].