UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Desenvolvimento de Solução Outsystems para Instituição Financeira

André Filipe de Jesus Corda

Mestrado em Engenharia Informática
Especialização em Engenharia de Software

Trabalho de Projeto orientado por:
Prof. Doutor Luís Antunes

2019

Agradecimentos

Quero agradecer em primeiro lugar à minha mãe que sempre lutou por mim e pela minha irmã para termos um futuro o melhor possível, muitas vezes tendo-se colocado em segundo plano para nosso prol. Seria impossível termos chegado onde chegámos sem o teu sacrifício. Obrigado.

À minha irmã Mariana que sempre me viu como um exemplo a seguir e que muitas vezes duvida de si mesma quando falha, mas lá no fundo tem capacidades para alcançar todos os seus sonhos e objetivos. És um verdadeiro exemplo a seguir, só tens de acreditar em ti mesma. Obrigado.

Aos meus restantes familiares e amigos de família de longa data, que tanto carinho têm por mim, e sempre nos suportaram, mesmo nos momentos mais difíceis. Obrigado.

Aos meus amigos que conheci na faculdade (especialmente os que entraram no meu ano), no secundário, ou até no infantário e ainda mantenho contacto. Aos momentos que passámos juntos, alguns por vezes menos bons, que serviram para crescermos enquanto pessoas e sobretudo a melhorar as relações, mas especialmente aos bons que neste momento são fantásticas memórias. Obrigado.

À Teresa Rebelo Pinto por me ter ajudado a manter o pensamento no sítio nestes últimos anos e a ser uma melhor pessoa a cada dia que passa, ensinando-me a crescer de dentro para fora. Obrigado.

Aos meus colegas de trabalho, por me terem acolhido tão bem apesar de ter sido a minha primeira experiência profissional na área de informática. Obrigado.

Ao Ricardo Castanheira, manager do projeto na empresa onde estagiei, por me ter dado esta oportunidade e responsabilidade que me ajudaram a crescer enquanto profissional. Obrigado.

E por último, mas igualmente importante, ao Professor Doutor Luís Antunes por ter aceite orientar o meu trabalho, possibilitando a conclusão do Mestrado em Engenharia Informática e o fim do meu percurso académico. Obrigado.

Resumo

O estágio sob a forma de Projeto de Engenharia Informática do correspondente Mestrado consiste na integração de uma equipa de *OutSystems*, através do desenvolvimento de uma aplicação de gestão de recursos e gestão de projetos, e a migração de uma aplicação interna para um grupo de utilizadores do cliente, através da metodologia *Scrum*.

Ao longo do estágio existiram também pequenos projetos de desenvolvimento ou novas funcionalidades a serem associadas aos produtos de responsabilidade da equipa, como por exemplo alguns simuladores, feitos inteiramente em *OutSystems* (versão 10), mas sempre consumindo serviços em diferentes tecnologias como C# ou *Web Services* pelas equipas de *Middleware* e EAI (*Enterprise Application Integration*). A correção de eventuais *bugs* foi feita maioritariamente através de manutenção corretiva e preventiva, tendo a manutenção proativa sido menos relevante devido ao elevado número de projetos concorrentes durante o período. A prevenção e correção é feita dando uso a ferramentas que a tecnologia *OutSystems* disponibiliza, como o *Service Center* ou *LifeTime*, que permitem análises e a extração de estatísticas coletiva ou individualmente enquanto aplicações, bem como *reports* de clientes que chegam até à equipa passando pelas diferentes linhas de suporte para ser analisada a origem do problema.

Palavras-chave: Outsystems, Web Services, Report Horas, Migração

Abstract

In some companies the time that is consumed reporting the time spent by employees and developers in their tasks can be very inefficient. When companies don't use an appropriate tool to do so, the jobs of the managers can also be difficult extracting the necessary data for analysis of how the projects did in terms of time, or if it was well predicted.

The main goal of this internship was to develop a tool in the *Outsystems* (version 10) technology that could help both employees and managers in their hourly reports, in which each user could be assigned to their specific projects and at the end managers could extract important information about it, in order to drop the traditional Excel sheet reports.

As a side goal, there was also a migration from a deprecated technology to *Outsystems* for an application that is used by a part of users in the company, in order to give it a cleaner look, as well as some minor projects related to both the car and health insurance simulators of the company. Most of these were for bug fixing or improvement, using *Outsystems* specific tools such as Service Center or LifeTime, for errors and statistics analysis. These minor projects included the usage of some other technologies, like C# from *Microsoft* or *Soap Web Services.*

Keywords: Outsystems, Web Services, Hourly Reports, Software Migration

Conteúdo

Lista de Figuras 4

Lista de Tabelas 7

Capítulo 1 Introdução 9

1.1 Motivação 9

1.2 Objetivos 9

1.3 Organização do documento 10

Capítulo 2 Tecnologias Usadas 12

2.1 Plataformas de *low-code* 12

2.2 *OutSystems* 12

2.2.1 Service Studio 13

2.2.2 Service Center 13

2.2.3 Lifetime 13

2.2.4 Integration Studio 13

2.3 Web Services 14

2.3.1 Remote Procedure Calls 14

2.3.2 REST e SOAP 15

SOAP 16

REST 16

Capítulo 3 Metodologia e planeamento 20

Planeamento 20

Arquitetura do sistema onde foram realizados os projetos 21

Capítulo 4 Trabalho Realizado 24

4.1 Aplicação de Gestão de horas e projetos 24

4.1.1 O problema e a análise de requisitos 24

4.1.2 Definição do modelo relacional 25

4.1.3 Implementação da camada Login 26

4.1.4 Desenvolvimento da aplicação - Interface 27

Criação/visualização de projetos 28

Criação/visualização das fases 29
Reportar horas 31
Aprovação de semanas submetidas 33
Visualização geral da aplicação 35
4.2 Migração de páginas de Caixa 36
4.2.1 Análise funcional 37
Listagem de cheques 37
Confirmação de lista de cheques 39
Cobranças do mediador 40
4.2.2 Implementação da camada de *Web Services* 41
Inputs e outputs comuns 42
Implementação em *OutSystems* de um serviço 43
4.2.3 Implementação da camada intermédia 44
4.2.4 Implementação da interface 46
Listagem de Cheques 46
Confirmação de lista de cheques 50
Cobranças do mediador 52
4.3 Testes dos desenvolvimentos 56
Capítulo 5 Conclusões e trabalho futuro 59
5.1 Conclusões 59
5.2 Trabalho Futuro 59
Bibliografia 62
Anexos 63
Anexo A 63
Anexo B 67
Anexo C 70

Lista de Figuras

Figura 3.1 Planeamento para o decorrer do estágio 20
Figura 3.2 Arquitetura do sistema 22
Figura 4.1 Entidade estática vs entidade 26
Figura 4.2 Interface do agregado, com respetivas tabelas e filtros 27
Figura 4.3 Página de detalhe de um projeto 29
Figura 4.4 Widget de input, com as respetivas propriedades 30
Figura 4.5 Interface para o report de horas semanal 31
Figura 4.6 Campo para o utilizador introduzir comentários sobre o dia reportado 33
Figura 4.7 Interface onde estão disponíveis as horas reportadas, por utilizador 34
Figura 4.8 Semana reportada por um utilizador, mas rejeitada pelo superior 35
Figura 4.9 Dashboard inicial da aplicação 35
Figura 4.10 Interface da página de lista de cheques, antes da migração, com respetivos requisitos 37
Figura 4.11 Interface da página de confirmação de lista de cheques, antes da migração, com respetivos requisitos 39
Figura 4.12 Interface da página de cobranças do mediador, antes da migração, com respetivos requisitos 40
Figura 4.13 Exemplo de um serviço importado, e estruturas criadas pela plataforma através do WSDL 42
Figura 4.14 Mapeamento de estruturas idênticas em Outsystems 42
Figura 4.15 Inputs e outputs comuns entre os serviços utilizados 43
Figura 4.16 Fluxo da implementação de um dos serviços, na plataforma Outsystems 44
Figura 4.17 Implementação do serviço na camada intermédia na plataforma Outsystems 45
Figura 4.18 Exemplos de identificadores, com respetivos campos e valores, do serviço de impressão 46
Figura 4.19 Página de lista de cheques, após a migração para Outsystems 47
Figura 4.20 Combobox populada com a lista do serviço que retorna as contas bancárias 48
Figura 4.21 Exemplo do Widget If usado na própria interface 48
Figura 4.22 Estrutura criada em Outsystems para converter em XML 50
Figura 4.23 Comando JavaScript para abrir a página num novo separador do browser 50
Figura 4.24 Página de confirmação de lista de cheques, após a migração para Outsystems 51
Figura 4.25 Diferentes ações consoante o estado da entrada da tabela 53
Figura 4.26 Página de cobranças do mediador, após a migração para Outsystems 55

Lista de Tabelas

Tabela 4.1 Tabela de requisitos da página de lista de cheques.. 37
Tabela 4.2 Tabela de requisitos da página de confirmação de lista de cheques............................... 39
Tabela 4.3 Tabela de requisitos da página do cobranças de mediador... 40

Capítulo 1 Introdução

Este relatório descreve o trabalho realizado no âmbito de uma Dissertação de Engenharia Informática/Projeto de Engenharia Informática, na Faculdade de Ciências da Universidade de Lisboa, no ano letivo 2018/2019. Foi um trabalho desenvolvido nas instalações de um dos clientes (uma seguradora) da empresa Unipartner IT Services, focando-se numa primeira fase na criação de uma aplicação para gestão de horas e projetos da empresa no cliente, e posteriormente numa migração de uma aplicação do cliente. A tecnologia usada em ambos os projetos foi *OutSystems*, uma plataforma capaz de produzir aplicações totalmente funcionais num curto espaço de tempo.

1.1 Motivação

Este relatório engloba-se no contexto de um estágio do Mestrado em Engenharia Informática - Engenharia de Software, na Faculdade de Ciências da Universidade de Lisboa. A motivação para ter sido escolhida uma dissertação deste estilo, ao invés da tradicional investigação autónoma, passou por haver a necessidade de envolvimento no mercado de trabalho o mais rapidamente possível, para começar a ter noções das diferenças que existem em informática numa empresa *vs* a informática enquanto estudante. A tecnologia escolhida, quase desconhecida até ao momento de entrada no estágio, teve também algum impacto na decisão. Mesmo sendo uma tecnologia desconhecida, *Outsystems* é uma tecnologia que de há alguns anos para cá começou a emergir no mercado e veio para ficar, muito graças às vantagens que potencia em relação a outras tecnologias.

A participação no estágio inclui o desenvolvimento de uma aplicação de gestão das horas dos trabalhadores, bem como a gestão de projetos dentro da equipa. Numa fase mais avançada do estágio, foi feita uma migração de páginas ASP (Active Server Page) para *OutSystems*, de uma das aplicações já existentes no cliente. A principal motivação foi não existir dentro da equipa uma ferramenta que disponibilizasse estas duas tarefas, sendo ambas feitas em Microsoft Excel e geradas as estatísticas e relatórios a partir do mesmo. O preenchimento por parte dos funcionários e agrupamento das informações dos ficheiros, extração de informação no Excel e uniformização dos dados por parte dos *Team Leaders* requeria muito tempo, necessário para a realização de outros projetos ou planeamentos.

1.2 Objetivos

Os objetivos deste estágio passam por uma primeira fase a aprender a utilizar o *IDE (*Integrated Development Environment) de *OutSystems*, o *Service Studio*. Após isto, foi feita concorrentemente uma aplicação para a gestão de projetos e *report* de horas por parte da empresa onde me insiro, e migração de páginas de uma aplicação do cliente em ASP. Visto que esta tecnologia começa a ficar obsoleta nos dias que correm, fez sentido migrar-se para uma mais recente, como *OutSystems*. A análise e requisitos funcionais, bem como lógica de *workflows*, foram definidos pelas necessidades e especificações do cliente em causa.

1.3 Organização do documento

O documento está organizado pelos seguintes pontos:

- Capítulo 2 – Tecnologias Usadas – este capítulo descreve brevemente as tecnologias usados ao longo do estágio;
- Capítulo 3 – Metodologia e Planeamento – o tipo de metodologia usada para desenvolver os projetos bem como o planeamento;
- Capítulo 4 – Trabalho Realizado – Descrição do que foi feito no projeto, através de levantamento de requisitos e funcionalidades implementadas;
- Capítulo 5 – Conclusão - onde sumariamente é descrita, através de uma visão crítica, o trabalho realizado, bem como possível trabalho futuro.

Capítulo 2 Tecnologias Usadas

Este capítulo apresenta um breve resumo sobre as plataformas *low-code*, *Outsystems*, e por fim um pequeno enquadramento com as diferentes tecnologias utilizadas durante os desenvolvimentos.

2.1 Plataformas de *low-code*

As plataformas de *low-code* são um tipo de software que promove um desenvolvimento rápido e eficiente, quer em ambientes de pequena ou grande escala. A necessidade deste tipo de plataformas surgiu quando começou a haver uma imensa procura por aplicações às quais os modelos de trabalho habituais não eram capazes de responder, tanto em termos de tempo de desenvolvimento como em termos de satisfação por parte dos clientes. Umas das principais vantagens na utilização deste tipo de tecnologia é o baixo custo e a facilidade de formação, o que permite a alguém sem conhecimento da tecnologia começar a produzir aplicações e *software* num curto espaço de tempo. As opções disponíveis são imensas, desde *User Interfaces*, *Data Modelling* e *Management* ou aplicações Mobile. Neste tipo de tecnologias, a componente de programação é feita quase exclusivamente pela plataforma, sem que o programador se aperceba. O simples arrastar de caixas e setas para a interface gera por si só uma quantidade significativa de código (HTML - HyperText Markup Language, PHP – Hypertext Preprocessor, JavaScript), completamente transparente e de forma quase imediata para quem o faz.
Forrester Wave[8] fez um estudo sobre os principais benefícios no uso deste tipo de tecnologia, tendo chegado às seguintes conclusões:

- Oferece uma variedade vasta de ferramentas que permitem definir fluxos, lógica, dados, diferentes tipos de aplicações e dispositivos suportados;
- Visto ser uma tecnologia de rápido desenvolvimento, facilmente se adapta às necessidades e alterações que os clientes possam querer;
- Apesar de muitas das plataformas *low-code* necessitarem de licenças para o seu potencial, grande parte disponibiliza um conjunto considerável de ferramentas na sua versão grátis que permite o desenvolvimento de aplicações com alguma complexidade.
- Aparecem como PaaS (*Platform as a Service*), disponibilizando assim os recursos como servidores e bases de dados remotamente, sem que os clientes se tenham de preocupar com um reinvestimento nas suas infraestruturas.

Alguns dos pontos menos bons destas tecnologias passam pela falta de certificados de segurança, ou a adaptação a temas do presente como Inteligência Artificial ou *Internet of Things* (IoT).

2.2 *OutSystems*

A tecnologia *OutSystems* envolve vários componentes, sendo que cada um tem um papel importante quando se fala de *OutSystems* como uma aplicação *low-code* e de desenvolvimento acelerado.

2.2.1 Service Studio

O Service Studio é o Ambiente de Desenvolvimento Integrado (IDE) onde quem produz *OutSystems* passa a maior parte do seu tempo. Após fazer a ligação ao servidor de *OutSystems*, pode-se desde logo começar a desenvolver uma aplicação, através da criação de módulos. Estes módulos podem ser responsivos (inclui os temas ou *layouts*, que a própria *OutSystems* disponibiliza através do Forge, e que carrega logo algumas funções RESS – *Responsive through Server-Side* components) ou em branco, onde o programador parte do zero. Usando estes temas, é gerado todo o código por trás, de modo à aplicação ficar funcional para *tablets*, *desktops* e *mobile*.

Após a criação dos módulos, é possível começar a definir os modelos de dados, a lógica envolvida ou *workflows* de negócio. Os módulos podem ser mais tarde referenciados por outros módulos, de modo a criar diversas camadas e transparência dentro da aplicação, ficando tudo o mais isolado possível e com menos dependências.

2.2.2 Service Center

No Service Center é possível ver um sistema de *logs* de todas as aplicações no servidor *OutSystems*, bem como análise e manutenção. A plataforma disponibiliza uma pilha de erros individual para cada aplicação, onde é possível ver com algum detalhe o que provocou a falha. Para cada aplicação, é ainda possível ver definições de segurança, o estado das integrações de API's externas ou *Site Properties* (no fundo são como variáveis globais em *Java*, mas onde é possível mudar os valores e propagar as alterações no momento). É possível ainda refrescar dependências desatualizadas ou fazer *downgrades* de versões dos módulos/aplicações. A plataforma disponibiliza também uma *grid* onde é possível analisar tempos e acessos aos diferentes *espaces* (módulos). No fundo, é uma vista global de todas as aplicações e o seu estado.

2.2.3 Lifetime

O Lifetime é similar ao Service Center, e é uma parte da plataforma *OutSystems* onde fica registada a performance das aplicações, datas de publicações e respetivo autor, ou o tempo que os pedidos demoram a ser feitos, entre cliente e servidor.

2.2.4 Integration Studio

Muitas vezes os requisitos das aplicações ultrapassam as capacidades daquilo que é possível fazer no Service Studio, como por exemplo a chamada a um *Web Service*[1] que tem um número elevado de *Inputs* e/ou *Outputs*. Uma maneira de ultrapassar este problema é com a criação de extensões Java ou C# no Integration Studio, fazendo um mapeamento dos atributos e libertando assim algum do esforço do Service Studio em chamadas aos *Web Services*.

2.3 Web Services

Atualmente, quando se fala na *Web*, muitas vezes é comum o termo *Web Service,* mas que pode ter significados muito diferentes consoante o contexto usado. Em alguns casos define-se *Web Service* vagamente como uma aplicação que está disponível na Internet. Noutros, define-se de uma maneira mais restrita, como por exemplo Gustavo Alonso et a*l* [2], que afimam que apesar de ser "aberta a outras aplicações, tem de ter interfaces bem definidas e capazes de serem definidas, descritas como artefactos XML". No fundo, um *Web Service* é definido como um componente que pode ser usado por outras aplicações mais complexas. Um *Web Service* é composto por:

- *Common base language* – por norma é XML, visto ser pouco complexa e fácil de adotar. Uma das vantagens é ser suportada globalmente por diversas aplicações, o que torna a comunicação entre as mesmas relativamente simples, independentemente das tecnologias suportadas por cada uma;
- *Interfaces* – como por exemplo CORBA IDL's [4], mas com algumas diferenças: é necessário especificar alguns parâmetros como endereços ou protocolo de transporte;
- *Business Protocols* – são as regras que indicam como os clientes que usam o *Web Service* devem proceder;
- Propriedades e semântica – para ajudar os clientes nas ligações, ou nas decisões se querem usar o *Web Service* ou não (pode ter características como preço ou qualidade);

Os *Web Services* podem assumir dois tipos de arquiteturas diferentes, *Internal Architecture* (recebem pedidos e reencaminham para outra camada, no fundo são mais uma camada de abstração) e External Architecture (que é uma infraestrutura que integra vários *Web Services*). Independentemente da arquitetura, existem três papéis dentro dos *Web Services*:

- *Provider* – é quem cria o Web Service e o torna disponível para ser usado pelas aplicações consumidoras;
- *Requestor* – quem faz a chamada ao Web Service;
- *Broker* – a camada intermédia que facilita a utilização do *Requestor,* abstraindo processos complicados ao utilizador e que garante acesso ao *Web Service*.

Além disso, os *Web Services* providenciam uma implementação de arquiteturas orientada a serviços, em que o desacoplamento que disponibilizam permite a escalabilidade das aplicações em toda a Internet. Alguns exemplos de *Web Services* são a *Cloud* ou *Grids* [12], que diariamente milhões de pessoas usam sem se aperceberem.

2.3.1 Remote Procedure Calls

Os pedidos aos *Web Services* são feitos através de *Remote Procedure Calls* [5], em que o principal objetivo é tornar o sistema o mais transparente possível para o utilizador. As chamadas remotas a outras máquinas ou processos são feitas como se localmente se tratasse, com a única diferença em que há comunicação pela rede: quando é feita uma chamada remota, o processo local é suspenso e transferido pela rede até ao local onde vai ser executada a operação; quando este acaba, envia de volta os resultados

pela rede para o ambiente onde foi chamado, onde o processo local volta a estar ativo. As grandes vantagens desta abordagem são a simplicidade que oferece a aplicações distribuídas através do uso de interface ou eficiência na gestão de recursos. Cada módulo tem uma interface onde estão definidos que métodos e variáveis podem ser acedidas pelos outros módulos, omitindo informação que não é necessária estar visível e permitindo a quem desenvolve os módulos alterar a parte "invisível" e outros módulos continuarem a aceder livremente. Algumas barreiras desta implementação são, por exemplo, as chamadas por referências. Se os processos não estiverem a ser ambos executados localmente, apenas é possível fazer chamadas por valores. Em chamadas remotas este parâmetro com o valor tem o nome de *input* se for um pedido, ou *output* se for uma resposta. Da mesma maneira, os endereços de memória também não podem ser passados como argumentos visto cada máquina guardar as variáveis de maneira diferente. As garantias de entrega de mensagens podem ser feitas de diferentes maneiras:

- *Maybe call semantics* – esta chamada remota é feita uma vez ou nenhuma, mas omissão de falhas, mensagens perdidas, ou falhas de clientes ou servidores impedem a confirmação e, portanto, são apenas usados em ambientes nos quais este tipo de falhas é inexistente ou em que são aceitáveis;
- *At-least-once call semantics* – esta chamada abstrai o problema de omissão de falhas, através de um envio de uma mensagem para quem efetuou o pedido, quer a chamada tenha sido feita com sucesso ou tenha terminado em erro; no entanto não lida com problemas como *crashs* ou falhas arbitrárias (executar a mesma ação mais que uma vez);
- *At-most-once call semantics* – tal como a chamada anterior, recebe-se uma confirmação com o resultado ou erro, mas impede falhas arbitrárias visto que cada chamada RPC é executada apenas uma vez.

Por norma a implementação de *Remote Procedure Calls* é feita usando o protocolo referido anteriormente, com uma maior utilização das duas últimas *call semantics* descritas. Em termos de *software*, o cliente tem um *stub* (uma implementação em que o seu único propósito é o de testar ou usar algo em específico) que para ele funciona como um procedimento local. Quando é executado, é transferido pela rede até ao servidor juntamente com o identificador atribuído ao recetor e os argumentos. Do outro lado, o servidor também possui um *stub* para cada serviço e consoante o pedido vindo na mensagem escolhe um que seja apropriado. Para enviar a resposta, é feito o mesmo processo.

Um exemplo é o *Sun RPC* [6], usado no *Network File System* (NFS), desenvolvido para comunicação entre cliente e servidor e no qual os clientes podem decidir se usam as chamadas remotas através de UDP ou TPC, sendo a diferença o tamanho das mensagens. Para cada protocolo existem ainda três tipos de *RPC's* (*batching* – não bloqueante que não espera resultados, blocking e *broadcasting* – o servidor só responde quando a chamada é feita com sucesso) oferecendo assim um total de seis configurações. Para a linguagem C, o *Sun RPC* oferece ainda uma interface, *rpcgen*, que funciona como um *stub*.

2.3.2 REST e SOAP

Ao referir-se Web Services, dois nomes que frequentemente estão associados a este termo são REST (REpresentation State Transfer) e SOAP (Simple Object Acess Protocol).

SOAP

Neste protocolo desenvolvido pela Microsoft são usadas mensagens XML para se trocar dados. Tal como todos os protocolos, existem algumas regras nas mensagens a garantir para que a comunicação seja cumprida na sua totalidade e sem falhas:

- SOAP Envelope – é um elemento do XML que identifica o documento como sendo uma mensagem do tipo SOAP. É obrigatório e subdividido em dois elementos:
  - *Header* – Só existe um na mensagem e contém informações sobre a comunicação entre a aplicação que fez a chamada e o Web Service em si, bem como definições de parâmetros, caso sejam complexos;
  - *Body* – Para a mensagem ser considerada válida é necessário pelo menos um elemento deste tipo e é onde estão os valores dos parâmetros.

Alguns dos principais obstáculos deste protocolo são a sua intolerância a erros relacionados com os parâmetros (por exemplo, inserir um valor *null* num campo que não pode ser *null*) e a complexidade, bem como o tamanho, dos *inputs* e *output*.

Uma maneira de evitar ou perceber o problema é através do WSDL, ou Web Service Description Language, que funciona como documentação do Web Service. É aqui que está descrito como deve ser feita a comunicação e definida a gramática do XML.

Ao usar documentos WSDL, que definem conjuntos abstratos de *endpoints* e *ports*, é possível separar as mensagens das suas definições concretas ou tipos de dados, tornando-se assim reutilizáveis. É constituído pelos seguintes elementos:

- Type – onde são definidos os tipos, usando um *type system* (preferencialmente é usado o XSD, ou XML Schema Definition);
- Message – os dados a serem trocados, abstratos;
- Operation – uma descrição abstrata do tipo ou tipos de ações que são aceites e suportadas pelo Web Service;
- Port Type – um conjunto de operações abstratas suportadas por um ou mais *endpoints*;
- Binding – um protocolo definido para os tipos de dados, específico a um *port* (exemplo, HTTP GET, HTTP POST);
- Port – é um *endpoint* definido pela combinação de uma *binding* e um *port type*;
- Service – conjunto de *endpoints*

REST

Mais recente do que o SOAP, o REST é uma arquitetura que usa o protocolo HTTP para a sua comunicação. É usado para aceder a recursos espalhados por toda a *Web*. É constituído pelos seguintes elementos:

- Resources – É o primeiro elemento, que indica a que tipo de recurso vamos aceder através do protocolo;
- Request Verbs – São o que descrevem as ações no pedido. Funcionando à base de HTTP, temos a possibilidade de usar métodos GET, POST ou PUT;

- Request Headers – São elementos que atuam como informações adicionais sobre as operações a realizar juntamente com o pedido, podendo ainda definir, por exemplo, permissões de acesso;
- Request Body – Consoante os tipos de pedido, pode ser necessário enviar dados ao Web Service: usando o método POST, é necessário indicar os dados que são obrigatórios atualizar ou adicionar;
- Response Body – É o elemento que contem a resposta vinda do Web Service;
- Response Status codes – São os códigos pré-definidos pelo HTTP, que vêm juntamente com a resposta do Web Service. Alguns exemplos são o Código 200 para uma resposta OK, ou 404 para Not Found.

A arquitetura REST tem algumas características específicas:

- Funciona como uma aplicação cliente-servidor;
- É Stateless, o que significa que é o cliente que efetua o pedido ao servidor onde está o *Web Service* o responsável por enviar a informação necessária para que a ação seja efetuada com sucesso. Além disso, o servidor não tem contexto de diferentes pedidos, mesmo que sejam feitos pela mesma entidade, garantindo assim o estado *Stateless* pelo qual a arquitetura é reconhecida;
- Suporta *cache* do lado do cliente, o que ajuda a controlar o tráfego de pedidos quando a quantidade de acessos ao servidor é elevada. Sendo uma arquitetura que se diz Stateless, havendo um cliente a fazer dois pedidos iguais cuja respostas serão também elas iguais, faz todo o sentido a existência de cache. Havendo este mecanismo, a arquitetura garante ainda uma maior escalabilidade;
- É uma arquitetura em camadas, permitindo ainda a inserção de camadas adicionais entre o cliente e o servidor. Por exemplo, uma camada adicional de *middleware* que trata regras de negócio.
- Ao contrário do protocolo SOAP que só funciona com mensagens do tipo XML, é possível haver diferentes tipos, tais como JSON, HTML ou também XML.

Estas caraterísticas fazem com que a arquitetura REST possa ser utilizada em diferentes ambientes, bem como em diferentes dispositivos, sem o utilizador se ter de preocupar em construir interfaces.

Em suma, os Web Services constituem uma parte importante da Web como a conhecemos hoje em dia, e distinguem-se por caraterísticas como a utilização de documentos no formato XML (no caso da arquitetura REST podem ser usados outros), desacoplamento entre cliente e servidor, permitindo ao cliente a recorrente utilização do Web Service, mesmo que este venha a mudar internamente, possibilita tipos de pedidos síncronos, mas também assíncronos e suporta a utilização de RPC's.

A utilização de Web Services facilita a comunicação entre aplicações, independentemente da tecnologia ou linguagem por elas utilizadas, garantindo assim inúmeras vantagens tais como o seu funcionamento sobre o protocolo HTTP, abrangendo a maior parte da Web, e podendo ser acedidos em qualquer parte e abstraindo diferenças entre aplicações a nível de tecnologias, arquiteturas e semântica, evitando esforço adicional na análise, construção e desenvolvimento de novos protocolos ou interfaces que ambas as aplicações percebam.

Em suma, de forma a ter uma arquitetura estável, eficiente e capaz de desempenhar as funções para a qual foi desenhada, é necessário incluir diferentes tipos de tecnologias para as diferentes funcionalidades.

Capítulo 3 Metodologia e planeamento

Neste capítulo é explicado sumariamente o tipo de metodologia adotados durante o estágio, bem como o planeamento no decorrer no mesmo.

Para a realização de ambos os projetos referidos no capítulo 1.1 não foi adotada uma metodologia em específico. No entanto, em ambos os casos houve um processo semelhante à metodologia ágil idêntica ao *Scrum*, através de reuniões periódicas com o cliente, com o intuito de conseguir aproximar a ideia inicial com o produto final.

O *Scrum* baseia-se em pequenas iterações de desenvolvimento, chamadas *Sprints*, normalmente dias e nunca mais que uma a duas semanas, onde o programador tenta apresentar ao cliente algo próximo do planeado. Existindo esta regularidade de reuniões, o cliente pode, ao longo do período de desenvolvimento, guiar o produto para o caminho por ele pensado ao início. Com esta forma de conceção evita chegar-se ao fim do produto e perceber-se que o que tinha sido inicialmente imaginado se tornou em algo que no final não era o pretendido, custando tempo dos programadores e dinheiro aos clientes.

O responsável pelo projeto organiza reuniões diárias rápidas ao início do dia com os programadores, com duração nunca superior a quinze minutos, onde cada membro indica o que fez no dia anterior, que obstáculos encontrou e se se encontra bloqueado ou como os ultrapassou, e o plano de trabalhos para o dia. Desta maneira, o responsável, também chamado de *Scrum Master,* pode orientar os recursos da maneira mais eficiente e ter uma ideia do estado do projeto.

Uma das vantagens deste método é a organização do projeto por fases, facilitando assim custeios e estimativas, que por vezes podem ser difíceis de calcular à partida devido à grande extensão do projeto.

Planeamento

Na figura 3.1 é apresentado o planeamento inicialmente feito para o decorrer do estágio, seguido de uma breve explicação de cada etapa

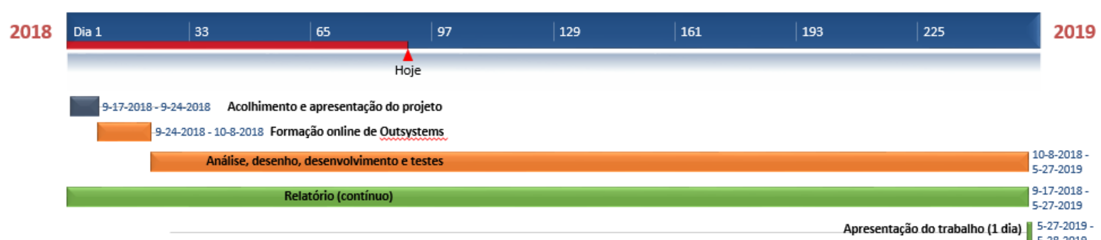

Figura 3.1 Planeamento para o decorrer do estágio

- Acolhimento e apresentação do projeto (1 semana) - Aspetos logísticos e ambientação à empresa/cliente, apresentação à equipa, introdução ao projeto e seus objetivos, enquadramento global e análise da documentação existente.
- Formação online de *OutSystems* (2 semanas)

- Análise, desenho, desenvolvimento e testes (33 semanas) – Participação na análise e desenho das funcionalidades identificadas para implementação. Desenvolvimento e testes às funcionalidades atribuídas e produção da documentação necessária.
- Relatório (contínuo) - Elaboração de um relatório preliminar a apresentar cumpridos os primeiros 60 dias e elaboração do relatório final de projeto sobre as atividades desenvolvidas.
- Apresentação do trabalho (1 dia) - Apresentação à equipa do trabalho realizado ao longo do projeto, lições aprendidas, aspetos positivos e menos positivos.

Arquitetura do sistema onde foram realizados os projetos

A arquitetura do sistema (fig. 3.2) do cliente onde foi realizado o estágio é composta por três níveis: apresentação, *Middleware* e *Back-End*. A arquitetura tem a capacidade de interpretar pedidos dos diferentes canais e domínios dentro da companhia do cliente, onde os utilizadores podem ter diferentes permissões. É utilizada uma arquitetura orientada a serviços (SOA), através de um componente que integra os serviços entre as tecnologias – EAI.

Na camada de apresentação é onde está presente toda a interação do utilizador com o sistema. Existem três componentes:

- LifeRay Gestão de Interfaces (LGI) – é usado o projeto *opensource* como *framework* e a sua tecnologia é *Java*;
- Outsystems – usa a sua própria plataforma;
- Camada de Gestão de Interfaces (CGI) – assenta em tecnologia *Microsoft*, baseada em ASP.net.

A principal diferença entre estas tecnologias é o facto de o componente CGI conseguir consumir os serviços diretos da camada inferior, visto partilhar a mesma tecnologia, ao contrário de Outsystems e LGI, que necessitam da ajuda do EAI para fazer a encapsulação dos serviços.

Na camada de lógica de negócio é feita a abstração dos dados e integrações entre tecnologias, sendo os seus dois grandes componentes EAI e *Middleware*. Através deste conceito de abstração, quem recorre aos serviços apenas se tem de preocupar em enviar no serviço a informação necessária, cabendo ao EAI resolver os problemas entre diferentes linguagens ou implementações. Os serviços disponibilizados utilizam *Simple Object Acess Protocol (SOAP)* como protocolo, com definição em WSDL e linguagem XML. O *Middleware* utilizada C# como linguagem e é responsável pela lógica do negócio, expondo serviços ao EAI. O acesso aos dados pode ser feito de duas maneiras, consoante a base de dados:

- Se for uma base de dados do próprio *Middleware*, é feito diretamente, através de *stored procedures* (lógica de base de dados que pode ser executada do lado do servidor);
- Se for base de dados com informações relativas a apólices, podem ser usadas também *stored procedures*, caso a tecnologia seja a mesma, ou através de serviços expostos pelo EAI.

Por último, a camada de dados, está dividida em dois grupos:

- As bases de dados geridas pelo *Middleware* e algumas equipas de negócio do cliente (Central), que a tecnologia é Microsoft SQL Server utilizando Transaction-SQL para

interagir com os dados, contendo sobretudo informações sobre simulações e parametrizações de produtos;

- As bases de dados geridas apenas pelo Central, contendo informações sobre apólices e usam a tecnologia COGEN.

Existe ainda a tecnologia *Salesforce* presente na arquitetura do cliente, implementada através da *Cloud* e integrada pelo EAI, para fazer gestão de clientes, gestão de sinistros ou consulta de documentos.

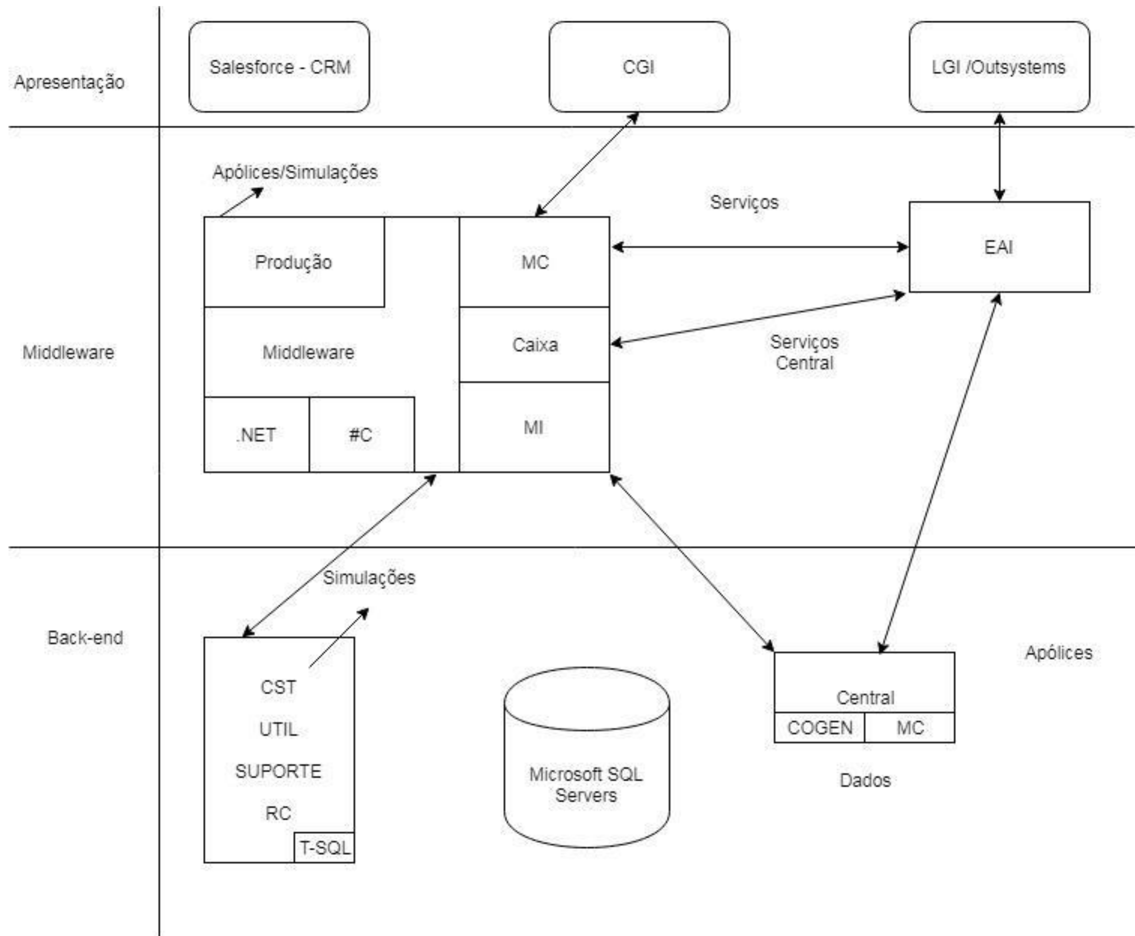

Figura 3.2 Arquitetura do sistema

Concluindo este capítulo, para que um projeto decorra conforme pensado é necessário que existam bons métodos de trabalho e que as metodologias aplicadas sejam as indicadas. É percetível também a heterogeneidade de tecnologias presentes, e que todas são fundamentais para um bom funcionamento da arquitetura. O planeamento acima referido sofreu um pequeno atraso, tendo sido concluído apenas no mês seguinte.

Capítulo 4 Trabalho Realizado

Este capítulo descreve o trabalho realizado durante o estágio. Numa primeira parte é explicado o desenvolvimento de uma aplicação para gestão de horas e de projetos dentro da equipa no cliente. Teve como objetivos a resolução de um problema de longa duração reportado pelos *Team Leaders* e um primeiro contacto com a tecnologia. Na segunda parte é descrita uma migração de uma pequena parte de uma aplicação que já existia no cliente feita em páginas ASP.net e que utilizava tecnologias como Javascript, HTML ou PHP para *OutSystems*.

4.1 Aplicação de Gestão de horas e projetos

4.1.1 O problema e a análise de requisitos

No cliente da empresa onde o estágio decorreu existia um problema que consistia na falta de ferramentas apropriadas para reportar as horas gastas em cada projeto, por pessoa. A solução adotada pela equipa era o preenchimento de um ficheiro Excel pelos *Team Leaders*, onde estavam representados todos os projetos e as suas características, e um segundo ficheiro onde cada pessoa colocava os projetos a que estava alocado e preenchia a grelha das horas. A gestão e cruzamento de dados finais eram feitos pelo *Manager* ou *Team Leaders,* que consumiam demasiado tempo numa tarefa que deveria ser relativamente simples.

Como proposta de integração na tecnologia que mais tarde iria ser a minha principal ferramenta de trabalho, foi criado um projeto para o desenvolvimento em *OutSystems* duma aplicação que facilitasse esta tarefa. Após algumas reuniões para perceber as funcionalidades desejadas no sistema e as necessidades da equipa, surgiram os seguintes requisitos.

Referentes ao utilizador:

- A aplicação tem de ter *login*, de modo a que os utilizadores possam reportar apenas as suas horas. Um dos problemas com a utilização dos ficheiros Excel era a possibilidade de qualquer pessoa conseguir editar as horas de todos os outros.
- Cada utilizador deverá estar associado a uma equipa (Middleware, LGI, Salesforce ou *OutSystems*).
- Deverão existir diferentes papéis dentro da aplicação. Ficaram definidos *Member* e *Team Leader*. Cada um deles terá privilégios com base nas suas funções e responsabilidades.
- Os *roles* (conjunto de permissões associados a um utilizador) com acesso à aprovação de horas reportadas deverão ter uma vista global dos utilizadores com horas por reportar, bem como uma lista das horas que já foram reportadas, mas ainda não foram aceites/recusadas.
- Associar os utilizadores a projetos, e consequentemente os utilizadores só podem reportar horas nos projetos a que estão associados. Deverá ser possível ainda inserir comentários em cada dia reportado pelo utilizador.

- As horas deverão de ser aprovadas ou recusadas pelos utilizadores com tais permissões, após estas terem sido submetidas pelos utilizadores. No caso de serem recusadas, deverá ser possível inserir um comentário sobre o porquê da não aceitação, ficando estas horas disponíveis novamente para o utilizador atualizar e com acesso ao comentário associado.

Referentes a funcionalidades:

- O ficheiro de projetos, já extenso e com algumas dezenas de colunas, tem de ser importado diretamente para a base de dados, para evitar a tarefa de inserção manual.
- Os projetos podem ter fases, isto é, o identificador do projeto é o mesmo, mas existem subprojetos, podendo ser independentes uns dos outros associados ao projeto principal.
- À semelhança da funcionalidade nativa do Excel na criação de filtros por colunas, deverá ser possível filtrar por valores de colunas específicas, como por exemplo, projetos de um *Team Leader*.
- Ainda na mesma tabela, podem existir entradas com certos atributos não preenchidos. Apesar de não serem obrigatórios, são campos que podem fornecer informações importantes aos utilizadores. Estes deverão conseguir perceber imediatamente se existem campos em falta.
- Colocar diretamente dúvidas na página correspondente ao projeto, em formato de comentário.
- Subdividir as fases do projeto em diferentes tarefas, cada uma com um custeio e uma equipa responsável. A soma do custeio das tarefas equivale ao total do esforço dessa fase. Além disso, ao criar novas fases estas deverão ter certas tarefas por omissão.
- Eventos importantes dos projetos e/ou fases, como entradas em ambientes de testes ou produção, são associados a *milestones*. Deverão também ter uma lista de *milestones*, incluindo as definidas por omissão, aquando da sua criação.
- Deverá ser possível a extração de informação sob a forma de ficheiros Excel de, por exemplo, listas de *milestones* ou horas reportadas pelos utilizadores num intervalo de duas datas.

Aspectos visuais:

- Deverá existir uma vista global com estatísticas dos projetos por ambiente, bem como as datas de projetos a entrar em cada ambiente.
- Deverá existir uma tabela com a listagem de todos os projetos, sendo o seu aspeto visual idêntico ao que existe no ficheiro e com todos os atributos do projeto.

4.1.2 Definição do modelo relacional

Um dos principais problemas à partida era não haver nenhum modelo de dados disponível em que o novo sistema encaixasse, pelo que teve de ser criado um totalmente novo. A plataforma *OutSystems* permite a criação de entidades (tabelas) e a definição das suas colunas, desde o tipo das variáveis até ao tamanho ou obrigatoriedade. Além das entidades ditas normais, existem também entidades estáticas, que ao invés de colunas ou campos são definidas através de *Records*, definidos por um identificador, uma *label*, uma ordem e um booleano para indicar se está ou não ativo (fig.4.1). Todas as tabelas foram criadas num módulo *Core* à parte de modo a criar independência entre camadas.

Estas entidades estáticas funcionam como um enumerado, úteis para definir atributos que à partida se sabe serem imutáveis. Neste projeto um dos casos da utilização deste tipo de dados foi para a definição dos estados em que o projeto pode estar, visto o universo de estados ser o mesmo ao longo do tempo.

Testes Record	
Identifier	Testes
Icon	Default Icon
Attribute Values	
Id	15
Label	"Testes"
Order	15
Is_Active	True

CodigoProjeto Entity Attribute	
Name	CodigoProjeto
Description	...
Label	Codigo Projeto
Data Type	Text
Length	50
Is Mandatory	No
Default Value	

Figura 4.1 Entidade estática *vs* entidade

A estratégia e desafio para o modelo de dados da aplicação foi perceber como ligar todas as tabelas. Haveria pelo menos três grupos:

- as tabelas associadas ao utilizador: ficou definido que iria ser usado um módulo diferente para os utilizadores (capítulo 4.1.3). A gestão de utilizadores em *OutSystems* é feita através de uma única tabela que tem uma coluna chamada Tenant_ID. Este campo é o identificador do módulo que criou os utilizadores. Definindo depois o *User Provider* da aplicação, o módulo principal teria acesso à tabela de utilizadores com o *Tenant_ID* especificado. Como a manutenção de utilizadores seria feita exteriormente, foi criada uma tabela *UserExtended* em que a chave primária seria o identificador dos *Users*, onde foram acrescentados depois todos os atributos relevantes.
- as tabelas associadas aos projetos/fase: existência de uma tabela para definir o projeto e uma outra para a fase. Foram criadas também diversas entidades estáticas que estão maioritariamente ligadas à fase, para definição de campos que se sabem ser fixos
- as tabelas associadas ao registo de horas: por último entidades para a associação das horas e respetivos utilizadores.

Nota: a definição de cada tabela está disponível nos Anexos A, B e C.

4.1.3 Implementação da camada Login

Como definido no capítulo 4.1.1 um dos requisitos funcionais da aplicação é a integração de autenticação, de modo a garantir que a ligação entre o utilizador e as informações gravadas na base de dados. Para desempenhar tal função, foi utilizado um módulo disponível no *Forge* que permite a criação de utilizadores, grupos, ou *roles*, através de um *BackOffice,* termo utilizado em *OutSystems* para uma aplicação que tem como objetivo a criação ou atualização de dados diretamente através de uma interface. Neste caso em específico foram criados os utilizadores, um para cada pessoa da equipa, com informações como o *username*, *password* ou *email*. Foram também criados os *roles TeamMember* e *TeamLeader*, e

posteriormente designados às pessoas apropriadas. Estes *roles* irão garantir certas permissões ou restrições a funcionalidades da aplicação.

4.1.4 Desenvolvimento da aplicação - Interface

Após a conclusão do modelo relacional e do módulo de *login*, a primeira iteração da aplicação começou por criar uma página que disponibiliza a importação dos ficheiros para a base de dados e apresenta a tabela com as fases dos projetos e respetivas características. Na preparação do ecrã são apenas usados *aggregates, queries* otimizadas para aceder à base de dados. Para a utilização deste elemento da plataforma *OutSystems*, basta arrastar as entidades (tabelas) e acrescentar os *joins* entre tabelas. É possível ainda adicionar filtros (equivalente à escrita SQL – por exemplo, *where projeto.codigoProjeto = 1885*), agrupar linhas por valor (GROUP BY) ou ordenação (ASC ou DESC).

Figura 4.2 Interface do agregado, com respetivas tabelas e filtros

A importação do ficheiro em *OutSystems* é feita através da utilização do elemento *Upload* que abre uma janela para o utilizador escolher o ficheiro desejado. Após a seleção, este fica guardado no contexto da página, onde é atribuído um identificador e o nome, tamanho e conteúdo podem ser utilizados pela página corrente. Para efetivamente ficar guardado na base de dados, é necessário persistir os dados na base de dados. Para tal efeito foi utilizado um botão com o evento OnClick a apontar para a função auxiliar ImportarExcel, em que no início da ação é utilizado o elemento ExcelToRecordList, também ele nativo do Service Studio, que recebe como *inputs* o ficheiro, a página de Excel e o r*ecord,* que define a estrutura que vai ser carregada. Foi criado uma estrutura ProjetoExcel, com os mesmo campos que a folha de Excel para este efeito. Depois disso é necessário percorrer todos os *records* e gravar a informação na base de dados usando as funções nativas de cada entidade para o efeito. Através da funcionalidade *For Each*, também ela nativa da plataforma, a lista é percorrida e a informação de cada *record* é tratada. Como podem existir várias fases associadas ao mesmo projeto, é utilizado um agregado cujo objetivo é filtrar os projetos pelo identificador do *record*. Consoante a lista retornada pelo agregado há dois fluxos possíveis: se estiver vazia, então significa que esse projeto ainda não foi criado e é necessário fazê-lo. Caso contrário, já existe e pode passar-se diretamente à criação da fase. Na

primeira situação em que o projeto não existe, este é criado com os campos nome e codigoProjeto, que são abastecidos pelas duas primeiras colunas do ficheiro. Se o projeto existir, então a chave estrangeira da fase a ser criada é o identificador do projeto retornado pelo agregado. A partir deste momento, são abastecidos todos os campos relativos à fase: datas de início e fim em cada ambiente, responsáveis ou custeios. Por último são criadas as tarefas e *milestones* que cada fase tem por omissão, com a chave estrangeira sendo o identificador da fase, e a tabela da página que contém a informação é refrescada.

Esta página inclui ainda a opção de filtrar a tabela por determinados campos, como ficou definido nos requisitos iniciais. Na figura 4.1 temos um exemplo com alguns desses filtros, através de *input texto* ou *comboboxes*. Estas últimas são abastecidas com um agregado colocado na preparação da página cuja função é ir à base de dados e retornar a tabela com as entidades estáticas. Cada um destes filtros tem associado uma variável local que é usada no agregado dos projetos e fases. Para os filtros das entidades estáticas é comparado se a variável local é nula ou se o campo do agregado equivale ao escolhido. Nos filtros em que o utilizador escreve é mais uma vez semelhante à linguagem SQL – Projeto.Nome like "%" + NomeProj + "%" equivale à função LIKE, sendo NomeProj a variável local que guarda o *input.*

Este ecrã, além de ter disponível a importação a partir do ficheiro, possibilita ainda a criação de um novo projeto através de um link para uma nova página, detalhada na secção seguinte.

Criação/visualização de projetos

Um dos requisitos definido ao início foi a importação do ficheiro Excel existente e previamente usado. Para evitar constantes importações e tentar adotar por completo a aplicação desenvolvida, foi incorporada uma página para a criação ou edição de projetos. A página recebe como *input* um ProjetoID, e se for nulo então é apresentada a página em modo de criação, caso contrário é possível ver as características do projeto escolhido. Na página anterior foi adicionado uma ligação para este ecrã com o projetoID a nulo, através da função NullIdentifier(). Caso o utilizador carregue na coluna que contém o código do projeto na tabela que lista as fases e respetivos projetos, então é enviado para a página o código do projeto correspondente. Na preparação é usado um agregado para obter os dados do projeto e da fase, através de um *inner join* e filtrado pelo identificador. Além disso, neste ecrã ainda é possível fazer a associação de pessoas a um projeto. É utilizado um segundo agregado para a tabela de relações entre projetos e utilizadores, UserProjeto, e uma *Advanced Query* para retornar todos os utilizadores que não estão associados ao projeto e popular a *combobox* na figura 4.2. Esta ação permite mais tarde ao utilizador reportar as horas aos projetos a que for associado. Por último a página contém uma listagem das fases do projeto, onde é possível criar ou carregar num dos *links* e ir para o detalhe da fase.

1745 - Legal DAUD Relatório 2016-007 aplicativo caixa

(Apagar Projeto)

Código do Projeto 1745

Nome do Projeto Legal DAUD Relatório 2016-007 aplicativo caixa

Guardar Voltar atrás

Fases do Projeto

Adicionar Fase

DESCRIÇÃO

Orçamentação (apagar)

Fase 1 (apagar)

CR - Impressão de cheques + TPA (apagar)

Custeio Total

Custeio do Serviço Unipartner (8h/dia): 69.515 dias

Lista de recursos

NOME

André Corda (remover)

-Associar recurso-

Figura 4.3 Página de detalhe de um projeto

Criação/visualização das fases

Da mesma forma que ao carregar na coluna do identificador do projeto existe um *link* associado, foi desenvolvido o mesmo comportamento para a coluna com o identificador da fase. Para a criação de fases, é necessário a associação a um projeto, pelo que o *link* para efetuar tal ação está apenas disponível no ecrã do projeto, como demonstrado na figura 4.3.

A preparação deste ecrã é bastante mais complexa que a do projeto, visto ter uma quantidade considerável de atributos. Através dos *inputs* FaseID e ProjetoID é utilizado um agregado para validar a existência da fase. Caso o agregado não retorne qualquer *record*, significa que a fase não existe e é apresentado o ecrã em modo de criação. Caso retorne uma linha, então é necessário apresentar ao utilizador informações como lista de tarefas, *milestones* ou comentários que se encontram guardados nas tabelas respetivas. Nestes agregados são usadas as chaves estrangeiras destas entidades para fazer a ligação entre tabelas. Existm ainda *advanced queries* para retornar custeios específicos de cada equipa.

Tanto neste ecrã como no de projetos, ocorrem situações onde a definição de *roles* tem efeito: se o utilizador tiver o *role* de TeamLeader, pode criar ou editar os campos, bem como *milestones* e tarefas. Caso contrário, só poderá visualizar e colocar comentários. Este comportamento é obtido usando a função CheckTeamManagerRole(), criada automaticamente aquando a criação dos *roles* no módulo de utilizadores e que recebe o identificador do utilizador com *login* efetuado, e devolve um booleano. É usada no campo *Enabled* dos elementos previamente identificados.

Tarefa_Descricao

Name	Tarefa_Descricao
Variable	TarefaTable.List.Current.Tarefa.Descricao
Validation Parent	
Mandatory	True
Null Value	...
Max. Length	1000
Width	(fill parent)
Text Lines	2
Margin Left	
Margin Top	
Style Classes	
Visible	True
Enabled	CheckTeamManagerRole(UserId:)
Type	Text
Prompt	

Figura 4.4 Widget de *input*, com as respetivas propriedades

Para a representação das *milestones* e tarefas de cada fase, foi usado um *widget* (componente base) de *OutSystems* chamado Editable Table. Este elemento tem a particularidade de desempenhar as mesmas funções que a tabela normal e não ter um tamanho estático conforme a *Source Record List* que lhe é atribuída. Isto significa que é uma tabela à qual podem ser adicionadas ou removidas linhas em *runtime*. Para tal basta definir os eventos *onRowSave* e *onRowDelete,* que são as ações para adicionar ou remover, respetivamente. A tabela no ecrã associada às tarefas, por exemplo, tem *inputs* para o utilizador introduzir a descrição, o custeio por equipa e o custeio total, que é calculado automaticamente após soma total dos custeios de todas as equipas. Na função que guarda as tarefas, o fluxo começa por garantir que os custeios de equipas que não foram selecionadas estão a zero, verificado através do atributo booleano na tabela de tarefas. Após isto, os *inputs* introduzidos pelo utilizador são colocados numa variável local do tipo Tarefa, e é utilizada a função disponibilizada pela plataforma em todas as entidades, *CreateOrUpdate*. Esta função recebe um *record* do tipo definido e, se o idenficador for nulo então faz um *create*, caso contrário faz um *update* ao *record* com o identificador passado. Depois da tarefa estar criada, é usado o identificador retornado pela função anterior para ser passada à variável local EquipaTarefa, que é uma entidade do mesmo tipo para fazer a relação entre equipas e tarefas. Por último, como pode ter havido alterações nos custeios de cada equipa, é feito um refresh nas *advanced queries* definidas na preparação do ecrã para refletir as mudanças. Quando se apaga uma tarefa o fluxo é mais simples, bastando apenas apagar a tarefa, que por sua vez apaga os *records* da EquipaTarefa devido à chave estrangeira ter a propriedade Delete Rule como “Delete”, que indica à base de dados para apagar o *record* com aquele identificador caso esta chave primária seja apagada (equivalente ao *CASCADE* em *SQL)*. Ao custeio total da fase é então subtraído o custeio desta tarefa, as *advanced queries* são refrescadas de novo e a fase é atualizada. Os eventos para adicionar ou remover entradas na tabela de *milestones* são mais curtos e diretos: para adicionar uma nova linha, o utilizador preenche o nome, datas prevista e real de início, e um valor. Por último pode escolher um valor inteiro como custeio total da *milestone*, ou uma percentagem do custeio total da fase, que é calculado antes de efetuar a persistência na base de dados através da função *CreateOrUpdate* da *milestone*. Para apagar é simplesmente usada a função *Delete* da entidade.

Reportar horas

Para o utilizador conseguir reportar horas foi desenvolvido um mecanismo na página das fases para associá-lo e, mais tarde, conseguir reportar horas aos projetos de que fizesse parte, de maneira individual. As horas seriam reportadas semanalmente e como tal faria sentido um calendário com os cinco dias úteis, havendo depois um *input* onde o utilizador pode escolher uma data para reportar horas num intervalo temporal diferente. O modelo de dados desenvolvido para obter tais características foi uma tabela para a semana (*TimeSheet*), uma tabela para cada linha da semana (*TimeSheetLine*) e uma tabela para cada dia individual (*TimeSheetItem*), existindo ainda uma tabela para a entidade estática *TimeSheetStatus* que indicaria o estado do report de horas (aprovado, submetido, rejeitado ou rascunho).

Figura 4.5 Interface para o report de horas semanal

O ecrã tem como *input* uma variável do tipo *Date* e na preparação da página esta é validada se é nula. Em caso afirmativo, é inicializada com a data atual. Como ficou definido que as semanas para reportar começam sempre a uma Segunda-feira, é necessário validar se o dia atual corresponde a Segunda-feira, ou se é preciso calcular. Através da função nativa do Service Studio *DayOfWeek*, é retornado um inteiro de 1 a 7 para cada dia da semana de Segunda a Domingo. Se o inteiro retornado for diferente de 1, então é necessário mais um passo para retornar a Segunda-feira dessa semana, também através de uma função nativa chamada WeekGetMonday, que recebe como *input* uma data e retorna a data da Segunda-feira dessa semana. Caso a data passada como *input* não seja nula, é necessário fazer as mesmas validações para ter a data de Segunda-feira correta. É depois usado um agregado que filtra a tabela das *TimeSheets* por utilizador e por data. Se o agregado retornar uma lista vazia, significa que ainda não existe uma semana para o utilizador reportar, pelo que é chamada uma função auxiliar *TimeSheet_Inicialize*, que cria uma *TimeSheet* com a data inicial para o utilizador. De modo a obter os projetos a que o utilizador está associado e pode reportar horas é usada a *advanced query* abaixo, que retorna as fases onde o utilizador está associado mas que ainda não foram associadas àquela semana específica. Cada vez que o utilizador escolhe uma fase para reportar, esta desaparece da *query* seguinte graças à segunda parte onde são filtradas as linhas que já existem no ecrã.

```
SELECT {Fase}.*, ({Projeto}.[CodigoProjeto] + ' - ' +
{Fase}.[Descricao])

FROM {Fase}
INNER JOIN {Projeto} ON {Fase}.[ProjetoId] = {Projeto}.[Id]
INNER JOIN {UserProjeto} ON {Fase}.[ProjetoId] =
{UserProjeto}.[ProjetoId]
AND {UserProjeto}.[UserExtendedId] = @UserExtendedId
```

```
AND EXISTS (
    SELECT *
    FROM {UserProjeto}
    WHERE {UserProjeto}.[ProjetoId] = {Fase}.[ProjetoId]
    AND {UserProjeto}.[UserExtendedId] = @UserExtendedId
)
AND NOT EXISTS (
    SELECT *
    FROM {TimeSheet}
    JOIN {TimeSheetLine} ON {TimeSheetLine}.[TimeSheetId] =
{TimeSheet}.[Id]
    WHERE {TimeSheet}.[StartDate] = @WeekStartDate
    AND {TimeSheet}.[UserExtendedId] = @UserExtendedId
    AND {TimeSheetLine}.[FaseId] = {Fase}.[Id]
)

ORDER BY {Fase}.[Descricao]
```

Query utilizada no elemento *advanced query* onde:

- O que está dentro de { } significa que é uma entidade;
- O que está entre [] a vermelho significa que é um atributo da entidade imediatamente anterior;
- O que está a verde seguido de @ são *inputs* da query,

Por último é usado um agregado para retornar as linhas que possam ter sido reportadas anteriormente pelo utilizador (*TimeSheetLines*) e é feito o cálculo do esforço, para ser apresentado no ecrã.

No evento onChange da *combobox* da figura 4.5 foi implementada uma ação que adiciona os projetos/fases ao calendário do utilizador. Quando este seleciona uma opção da lista, é espoletado a ação que começa por revalidar a existência da Timesheet. Se esta não existir, é criada e posteriormente todas as linhas são iteradas à procura de alguma com o identificador da fase que foi selecionado. Se for encontrado, a variável local *isInSheet* do tipo boolean toma o valor *True*. No fluxo imediatamente a seguir é validado esta variável, onde é criada uma entrada para a *TimeSheetLine* caso não exista. A tabela é refrescada, bem como a *query* que tem as fases a adicionar ao *report* de horas.

Na tabela onde o utilizador introduz as horas, é possível remover fases, através do *link* disponibilizado para o efeito. Para tal é usado um agregado que filtra o identificador da fase pelo selecionado, e é apagada a *TimeSheetLine* através da função *Delete* da entidade. A *advanced query* é refrescada para voltar a aparecer a fase apagada nas opções de escolha, bem como a tabela para refletir as mudanças. Cada dia da semana tem uma caixa de texto *input* que espoleta um evento associado à ação *UpdateTimeSheetLine*, onde cada *TimeSheetItem* é atualizado com o valor que o utilizador introduz, e é associada a data respetiva. De notar que a última linha da tabela representa sempre o total de horas para cada dia da semana, por coluna.

Por fim é necessário submeter as horas, para posterior aprovação. Através do botão disponibilizado para o efeito "Submeter para aprovação", figura 4.5, o utilizador aciona o mecanismo de submissão das horas para aquela semana em específico. Para obter uma submissão com sucesso é validado se o total de cada dia da semana equivale a oito horas, bem como as quarenta horas semanais. Se passar nas validações, então o *Status* da *TimeSheetLine* fica "*Submitted*" e é feito o *update* à base de dados. Na tabela os campos de *input* passam a ficar bloqueados e ficam apenas de leitura, voltando a ficar disponíveis se a submissão for recusada pelo *TeamLeader*. É ainda possível adicionar comentários a cada dia individualmente através dum ícone disponibilizado para o efeito.

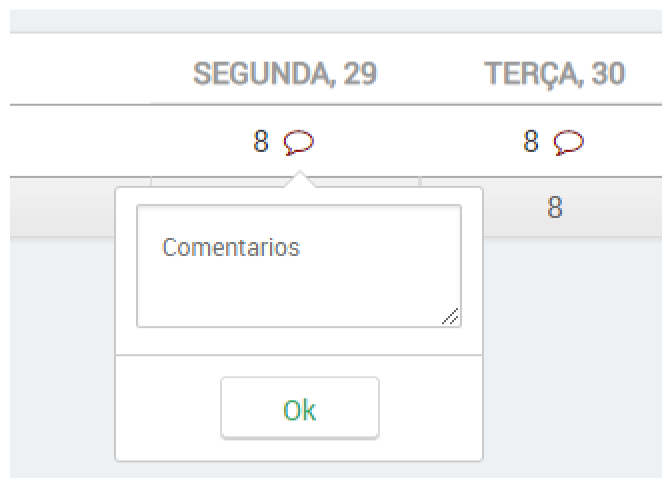

Figura 4.6 Campo para o utilizador introduzir comentários sobre o dia reportado

Aprovação de semanas submetidas

O ecrã de aprovação de horas só está disponível para aqueles com *role* de TeamLeader. Todas as semanas submetidas com sucesso ficam então disponíveis para mais tarde serem aprovadas. A preparação deste ecrã é composta por apenas uma *advanced query* cuja função é obter todos os utilizadores que têm semanas submetidas, através do excerto SQL em baixo:

```
SELECT {User}.*
FROM {User}
WHERE EXISTS (
     SELECT *
     FROM {TimeSheet}
     WHERE {TimeSheet}.[UserExtendedId] = {User}.[Id]
     AND {TimeSheet}.[StatuId] = @Submitted
)
ORDER BY {User}.[Name]
```

Cada utilizador é passado como *input* para um *Web Block* no ecrã, que se situa dentro de uma tabela para iterar a lista na sua totalidade. Na preparação do *Web Block* é usado um agregado para filtrar a tabela de *TimeSheetLines* pelo utilizador recebido como *input*, e pelas linhas que têm o estado "*Submitted*". Tal como o ecrã principal, este *Web Block* também tem uma tabela e a lista que a popula é a retornada pelo agregado, que contém todas *TimeSheetLines*. Dentro da tabela existe um segundo *Web Block*, que recebe a *TimeSheetLine* e o utilizador como *input.* Na preparação deste último é usado também um agregado para retornar os projetos e fases a que o utilizador e a *TimeSheetLine* pertencem, através das chaves estrangeiras. A lista retornada pelo agregado é então usada como *Source Record List* na tabela, além de ajudar a preencher o ecrã com os nomes do utilizador e projetos/fases.

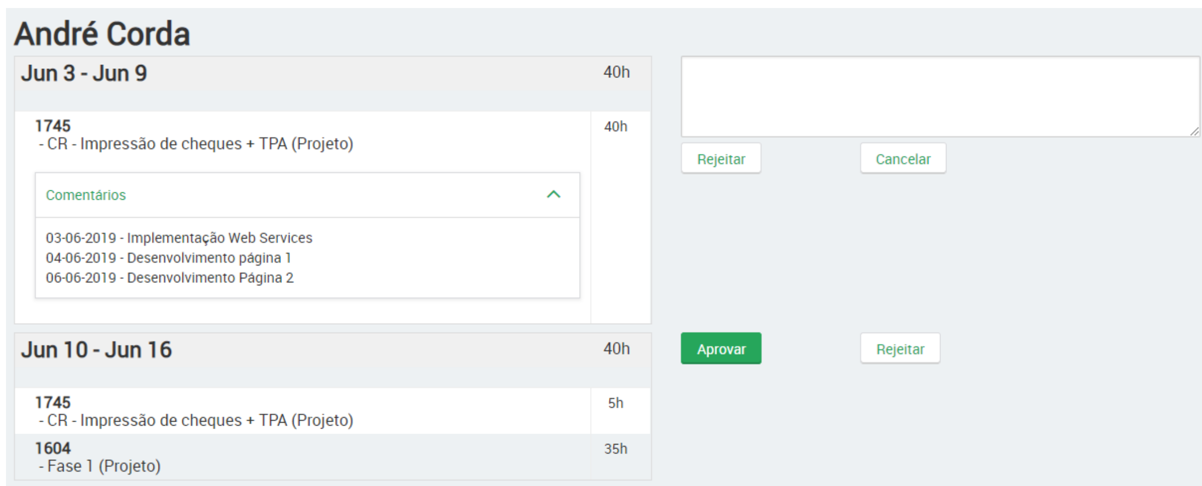

Figura 4.7 Interface onde estão disponíveis as horas reportadas, por utilizador

Um utilizador que tenha acesso a esta página pode então visualizar todas as semanas submetidas por todos os utilizadores, como o exemplo da figura 4.7. Os dados estão separados por utilizador, e para cada utilizador há um bloco para cada semana, onde são indicados o tempo total, tempo por projeto ou até comentários que o utilizador possa ter inserido. Pode aprovar ou rejeitar individualmente cada semana de cada utilizador, através dos botões com as *labels* respetivas. Cada botão tem o seguinte comportamento:

- Aprovar – chama a ação *ApproveOrRejectTimeSheet* com o *input* "*Approve*" a *True*, atualizando o *status* da TimeSheet para Aprovada. A seguir, é usado um *For Each* para percorrer as linhas da *TimeSheet* e adicionar as horas reportadas ao custeio consumido de cada fase e esta é atualizada;
- Rejeitar – chama a ação *ToggleApproveOrReject* com o *input* "*setShowApprove*" a *False*, colocando a variável booleana local *ShowApprove* com o mesmo valor e fazendo um *Ajax Refresh* (acção que recarrega apenas o elemento da página em específico, não havendo necessidade de efetuar um novo pedido HTTP ao recarregar a página toda) ao elemento que contém os botões e caixa de texto para o comentário de rejeição. Esta variável local controla o que aparece no ecrã, sendo que se for *True* aparecem os botões Aprovar e Rejeitar, e se for *False* os botões Rejeitar, Cancelar e o *input* do comentário. Na situação em que o utilizador quer rejeitar as horas reportadas, o botão Rejeitar chama a mesma função utilizada pelo botão, mas com o *input* a *False*, colocando o estado da *TimeSheet* a Rejeitado e adicionando o comentário na caixa de texto ao campo *RejectionReason*. O botão Cancelar chama a função *ToggleApproveOrReject* com o *input* a *True*, voltando os dois botões anteriores a ficarem disponíveis.

Quando uma semana é rejeitada, esta volta a estar disponível no ecrã de *report* de horas para que o utilizador que a reportou volte a submeter com as devidas alterações, e com a razão pela qual foi rejeitada, como mostra a figura 4.8.

Figura 4.8 Semana reportada por um utilizador, mas rejeitada pelo superior

Visualização geral da aplicação

De modo a permitir os utilizadores terem uma vista global sobre estatísticas dos projetos e horas dos utilizadores foi adicionada uma página onde é possível ver a quantidade de projetos em cada ambiente, os próximos projetos a entrar em cada ambiente (ambos baseado nas datas inseridas) e horas que foram reportadas ou faltam reportar por utilizador.

Figura 4.9 Dashboard inicial da aplicação

A figura 4.9 acima é um exemplo da página de sumário, onde:

1. É um menu superior onde existem todas as ligações para navegar até às diversas páginas da aplicação. Este menu está também disponível nos restantes ecrãs desenvolvidos.
2. Existem quatro *Cards*: projetos em fase de planeamento, em desenvolvimento, em testes e em produção. O número representa a quantidade de projetos num dado ambiente e é obtido através de uma *query* à base de dados que verifica à data corrente em que os projetos se situam, e é feita uma soma das linhas retornadas.

3. Por baixo de cada *Card* existe uma tabela com cinco linhas, que indicam os próximos projetos a entrar nesse ambiente, à exceção do ambiente de produção que indica os últimos cinco que ficaram disponíveis nesse ambiente. Os dados também são obtidos através de um agregado que filtra os projetos por data de cada ambiente, e são ordenados por essa mesma data. No agregado existe uma propriedade *Max Count*, que tem o mesmo comportamento quando se usa TOP em SQL, para evitar leituras à base de dados desnecessárias. Neste campo foi utilizado o valor 5
4. Nesta tabela é apresentado ao utilizador as horas reportadas por todos os utilizadores e que ainda não foram aprovadas ou rejeitadas.
5. Nesta tabela é apresentado ao utilizador as semanas dos utilizadores que ainda não foram inicializadas, e consequentemente reportadas.

4.2 Migração de páginas de Caixa

Numa segunda etapa do estágio, surgiu a oportunidade de integrar um projeto que consistia na migração de três páginas de uma aplicação em ASP.net para a tecnologia *OutSystems*, visto a linguagem estar a ficar descontinuada e desatualizada em relação às outras aplicações na estrutura.

Antes da migração da aplicação começar, existiram algumas reuniões com as pessoas responsáveis pelas páginas antigas para se perceber o seu funcionamento, funcionalidades que deveriam ser melhoradas ou excluídas, ou novas funcionalidades que poderiam acrescentar valor à aplicação.

Numa fase inicial, foi disponibilizado o acesso ao *source code* para realizar a análise funcional e fazer uma lista dos requisitos que a migração tinha de cumprir.

Uma das razões que levou à decisão da migração para *OutSystems* foi ficar tudo centralizado numa única tecnologia, ao invés de várias tecnologias (algumas com vários anos de idade e obsoletas), o que ajuda futuramente uma manutenção e análise mais simples, bem como não existindo a necessidade de conhecimento de várias linguagens e tecnologias.

Enquanto a análise funcional era realizada, foi feito paralelamente o desenvolvimento de *Web Services* por parte da equipa do Middleware e expostos pela equipa do EAI que iriam ser usados na migração. Estes *Web Services* serviram para mais tarde na implementação obter e persistir informação nas bases de dados, não havendo assim necessidade de criar um modelo relacional na aplicação.

4.2.1 Análise funcional

Da análise funcional das páginas foram gerados os seguintes requisitos, separados por páginas, e numerados consoante a figura imediatamente anterior.

Listagem de cheques

Aqui são listados os cheques abertos, onde depois se gera lista de cheques, associados ou não a uma moeda.

Figura 4.10 Interface da página de lista de cheques, antes da migração, com respetivos requisitos

1. Escolha da conta bancária
2. Adicionar valor de numerário à lista
3. Listagem Global: ver cheques abertos por todos os utilizadores daquele diário
4. Definitivo: apenas gera PDF ou persiste a transação
5. Impressão de cheques, para posteriormente serem confirmados
6. Selecionar todos os cheques
7. Ordenação nas colunas

Tabela 4.1 Tabela de requisitos da página de lista de cheques

Definição dos requisitos:

1. A partir desta *combobox* é escolhida a conta bancária do utilizador a que vai ser associada a lista de cheques gerada. A lista é carregada através da invocação de um serviço, WSCX014_CxBOOpenListaCheques, que retorna todas as contas do utilizador.
2. É um campo de *input* que mais tarde ao realizar a transação para a criação da lista de cheques é adicionado ao input do serviço.
3. É uma *checkbox* que guarda um valor booleano, e ao ser atualizada executa uma chamada ao serviço WSCX013_CxBOSelListaCheques com o valor correspondente, *true* ou *false*. Serve

para retornar apenas os cheques associados ao utilizador corrente se não estiver selecionada, ou caso contrário para retornar todos os cheques de todos os utilizadores.

4. É uma checkbox que indica ao serviço que gera a lista de cheques se a transação é temporária ou definitiva, a ser enviado no *input* do serviço associado ao botão "Imprimir".
5. É o botão que gera a lista de cheques, enviando para o serviço WSCX015_CxBOImprimirListaChequ todas as linhas selecionadas na tabela. Além disso, tem associado uma chamada a um outro serviço, WSDOD11_ObterDocumentoImprimir, que gera um documento PDF com as informações dos cheques e numerário, caso exista, abrindo numa Tab adicional do browser este mesmo ficheiro. Além disso, refresca a página, e os cheques associados à lista gerada desaparecem da tabela, caso a *checkbox* definitiva esteja assinalada.
6. Ao carregar neste botão, todas as linhas da tabela são selecionadas, ou caso estejam todas selecionadas, tem o comportamento inverso.
7. Ao carregar no *Header* de cada coluna da tabela, esta é ordenada crescente ou decrescentemente pelos valores dessa coluna.

Sendo estes os requisitos presentes na página antiga, surgiram mais alguns para facilitar a usabilidade do utilizador:

8. As linhas selecionadas na tabela apenas se diferenciavam das restantes com a mudança de cor, pelo que foi sugerido o acréscimo de uma *checkbox* em cada linha para este efeito.
9. Os valores dos cheques totais, a totalidade de cheques, o valor dos cheques selecionados e o respetivo número estavam dispostos de uma maneira pouco intuitiva para o utilizador, sendo que algumas destas informações se encontram no topo da página e as restantes na parte inferior. Como tal, foi aceite como requisito agrupar estes campos num espaço adequado.

Confirmação de lista de cheques

Este segundo ecrã, relativamente mais simples tanto a nível visual como lógico, é onde aparecem as listagens de cheques geradas a partir do primeiro ecrã. Aqui são alterados o valor do campo "Confirmado" conforme a ação executada.

Figura 4.11 Interface da página de confirmação de lista de cheques, antes da migração, com respetivos requisitos

1. Tabela com lista de cheques e respetivos campos
2. Anular listagens de cheques
3. Confirmar listagens de cheques
4. Ordenação nas colunas

Tabela 4.2 Tabela de requisitos da página de confirmação de lista de cheques

Definição dos requisitos:

1. Ao carregar a página, aparecem todas as listas de cheques associadas ao utilizador com *login* efetuado.
2. Este botão tem executa a chamada a um serviço que tem dois comportamentos diferentes, consoante o estado "Confirmado". Se a lista estiver confirmada, volta a ficar não confirmada, se não estiver confirmada, é anulada e a lista é desfeita, aparecendo assim os cheques de novo no ecrã de listagem de cheques.
3. Botão que tem o comportamento inverso do anterior. Se a lista estiver no estado não confirmada, altera o valor, caso contrário não tem qualquer impacto.
4. Ao carregar no *Header* de cada coluna da tabela, esta é ordenada crescente ou decrescentemente pelos valores dessa coluna.

Sendo estes os requisitos presentes na página antiga, surgiram mais alguns para facilitar a usabilidade do utilizador:

5. Para anular ou confirmar uma lista de cheques, era necessário selecionar uma linha na tabela e posteriormente carregar no botão adequado à ação desejada. Ambos os botões irão desaparecer, sendo o "Anular" um link associado ao Web Service antigo e que estará presente em cada linha da tabela. Além disso, irá existir uma *checkbox* em cada linha na coluna "Confirmado", que consoante o seu valor irá espoletar duas ações diferentes. Se a checkbox estiver confirmada, é chamado o *Web Service* correspondente à anulação da

listagem, se não estiver selecionada, é chamado o *Web Service* que confirma. Por último, se a lista estiver confirmada, o *link* para a anulação só deverá estar visível caso a lista esteja no estado "Não confirmado".

Cobranças do mediador

No terceiro e último ecrã da aplicação, o utilizador escolhe um dos seus lotes abertos e preenche as entradas da tabela, associando um número de apólice a um recibo.

Figura 4.12 Interface da página de cobranças do mediador, antes da migração, com respetivos requisitos

1.	Escolher um lote das opções
2.	Ordenação nas colunas
3.	Possibilidade de edição/apagar/adicionar linhas na tabela
4.	Certos campos serem limitados a determinados valores, alguns tornam-se obrigatórios
5.	Botão guardar só esta disponível caso não haja alterações a guardar

Tabela 4.3 Tabela de requisitos da página do cobranças de mediador

Definição dos requisitos:

1. Quando a página é carregada, a tabela não tem qualquer informação presente. É necessário escolher uma opção na *combobox* para carregar os dados associados a esse lote.
2. Ao carregar no *Header* de cada coluna da tabela, esta é ordenada crescente ou decrescentemente pelos valores dessa coluna.
3. A primeira linha da tabela é de inserção de dados. Ao preencher os dados corretamente, é gerada uma nova entrada. Carregando numa linha já existente, as informações passam para a primeira linha, e qualquer alteração aos dados presentes altera a entrada carregada inicialmente. Ao carregar numa linha, o botão "Eliminar linha" elimina os dados correspondentes. No entanto, as alterações são apenas visuais, persistindo realmente a eliminação aquando da utilização do botão guardar.

4. O preenchimento de campos como "Tipo de recibo" com valores predeterminados exigem a obrigatoriedade de outros campos, como por exemplo Data ou Valor.
5. Guarda as alterações feitas desde o carregamento inicial da página, ou desde a última ação guardar. O botão só deve estar disponível caso haja alterações que ainda não estejam na base de dados.

4.2.2 Implementação da camada de *Web Services*

Nesta secção vai ser explicada a implementação dos serviços disponibilizados para o projeto por parte das outras equipas (EAI e *Middleware*). Visto ser um processo de migração de uma aplicação que ainda não tinha qualquer tipo de tecnologia *OutSystems*, toda a estrutura foi desenhada de raiz.

O primeiro módulo criado foi o "Caixa_IS", responsável por conter apenas o consumo dos *Web Services* necessários ao projeto, e eventualmente futuros serviços que façam sentido no contexto da aplicação Caixa. Este módulo, onde está toda a implementação dos serviços, é a camada que produz toda a transparência da tecnologia *OutSystems* com as restantes. É usado este tipo de arquitetura, ao invés de ter tudo agrupado no mesmo módulo, face à possibilidade de mudanças exteriores ao que está implementado em *OutSystems*. Caso haja uma decisão em mudar *inputs*, *outputs* ou até mesmo nomes de estruturas ou seus campos, basta fazer um refrescamento ao serviço neste módulo e mais tarde propagar as mudanças nas camadas restantes.

Para esta migração, foram disponibilizados um total de 10 Web Services, agrupados por página onde são utilizados:

- Página de lista de cheques
  - WSCX013_CxBOSelListaCheques
  - WSCX014_CxBOOpenListaCheques
  - WSCX015_CxBOImprimirListaCheques
  - CallWSDOD11_ObterDocumentoImprimir
- Página de confirmação de lista de cheques
  - WSCX011_CxBOOpenConfirmarListaCheques
  - WSCX012_CxBOConfirmarListaCheques
  - WSCX016_CxBOAnularConfirmarListaCheques
- Página de cobranças do mediador
  - WSCX017_CxBOGetCobrancasBackOffice
  - WSCX018_CxBOGetLote
  - WSCX019_CxBOCobrancasMediadorGuardar

Após a importação dos serviços pela plataforma do *OutSystems*, esta gera automaticamente todas as estruturas definidas pelo WSDL. É possível então usar estas estruturas para mapear os *inputs* necessários que vêm das camadas superiores e após a execução do serviço fazer o mesmo para o *output*.

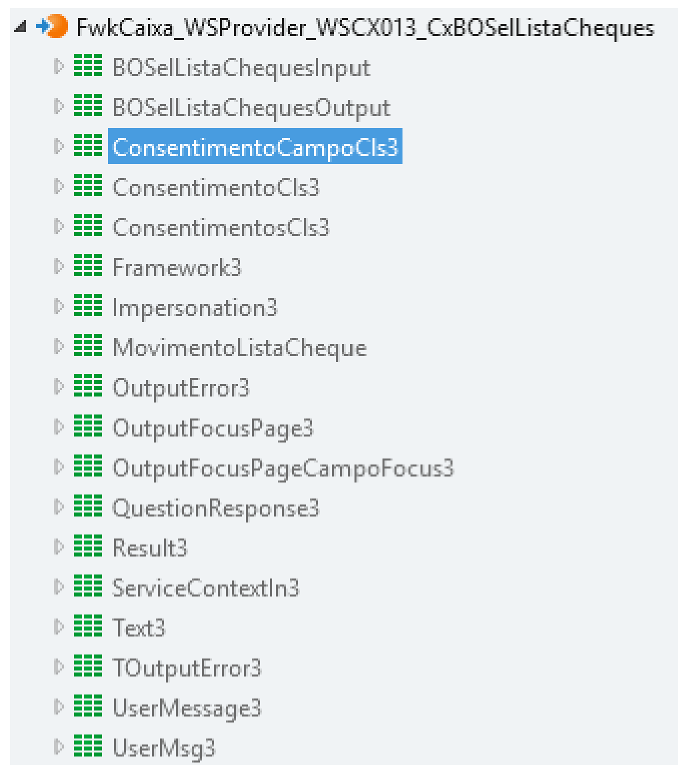

Figura 4.13 Exemplo de um serviço importado, e estruturas criadas pela plataforma através do WSDL

Na figura 4.13 temos um exemplo das estruturas criadas pela plataforma. Notar que as estruturas que têm números imediatamente a seguir são estruturas idênticas que já tinham sido criadas previamente, mas a plataforma considera que são diferentes e incrementa uma unidade no final do nome. Apesar das estruturas poderem ser reutilizadas, é aconselhado que se usem as estruturas geradas especificamente para o serviço. É depois necessário fazer um mapeamento de cada um dos atributos para a plataforma saber que valores associar entre si. De seguida é dado um exemplo na figura 4.14 de um mapeamento da estrutura *ServiceContextIn,* que existe como *input* em todos os serviços, mas a plataforma gera uma nova.

Mapping from ServiceContextIn11 to ServiceContextIn9	
RootTransactionID	RootTransactionID
ParentTransactionID	ParentTransactionID
RequestID	RequestID
ClientId	ClientId
ClientContextId	ClientContextId

Figura 4.14 Mapeamento de estruturas idênticas em *Outsystems*

Inputs e outputs comuns

Em todos os serviços utilizados existem certas estruturas ou variáveis de input e outputs comuns, com o mesmo nome e tipo.

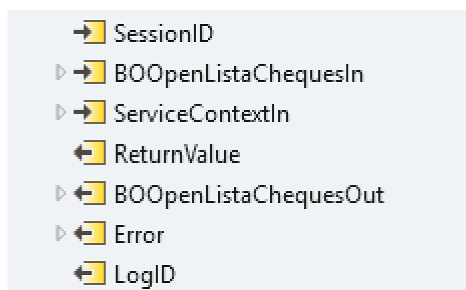

Figura 4.15 Inputs e *outputs comuns entre os serviços utilizados*

- *SessionId* – esta variável do tipo *Text*, indica ao serviço o identificador da sessão. Este identificador é criado automaticamente através do login no portal da empresa. Se a sessão expirar, é necessário fazer login novamente para refrescar a sessão, ou caso tenha ultrapassado o seu limite de vida, é gerada uma nova sessão. Através do valor deste campo, é possível verificar todas as ações efetuadas com este identificador, no sistema de *logs*. A função que devolve este identificador é explicada com maior detalhe na secção 4.2.3.
- *ServiceContextIn* – é uma estrutura composta por 5 campos: *RootTransactionId*, *ParentId*, *RequestId*, *ClientId* e *ClientContextId*. Tal como o *SessionId*, estes campos são gerados a partir de uma função já existente na plataforma da empresa e que devolve os identificadores dos pedidos.
- *Return value* – devolve um inteiro. 0 caso o serviço tenha sido executado com sucesso, ou -1 em caso de erro.
- *Error* – é uma estrutura composta por 4 campos: *Code* (identificador do erro, caso tenha), *Description* (mensagem de erro), *Source* (origem do erro) e *Type* (existem 2 grandes tipos, Business que são regras de negócio e *System*, que tal como o nome indica são erros de sistema, por exemplo, sessão inválida). Esta estrutura é sempre devolvida, em caso de sucesso todos os seus campos são devolvidos vazios.
- *LogId* – variável do tipo *Text* que contém o identificador da chamada ao serviço.

Implementação em *OutSystems* de um serviço

Após a importação dos serviços e criação das respetivas estruturas necessárias à chamada de cada um, foi desenvolvida, para cada serviço individualmente, uma função com *inputs* e *outputs* idênticos ao serviço em si. Estas funções irão ser usadas posteriormente na camada Caixa_CS, descrita na secção 4.2.3, responsável por grande parte da lógica necessária ao desenvolvimento da aplicação. Nesta secção é apresentado apenas a implementação de um dos dez serviços, visto serem todos semelhantes. As únicas diferenças são as estruturas de *input* e *output* específicas a cada um dos serviços.

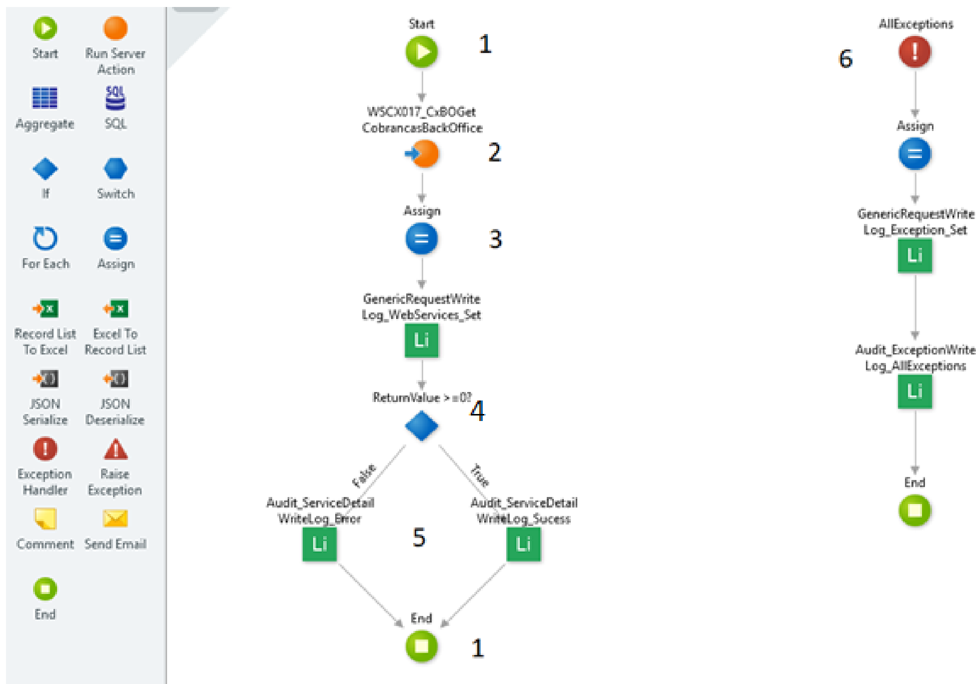

Figura 4.16 Fluxo da implementação de um dos serviços, na plataforma *Outsystems*

Legenda:

1. Nós de entrada e saída, indicam o começo e fim, respetivamente, de cada ação;
2. Chamada ao serviço consumido pela plataforma. Aqui são enviado os *inputs* necessários à sua chamada;
3. Após a chamada do serviço, o *assign* coloca os valores de *output* do serviço nas variáveis de *output* que vão ser expostas pela ação;
4. Neste nó compara-se o valor de retorno do serviço para validar se ocorreu um erro, caso o valor seja inferior a zero, ou uma chamada com sucesso, caso o valor seja maior que zero;
5. Estas são ações já disponibilizadas anteriormente que guardam o registo da chamada. Uma para caso de erro, que recebe como *input* a estrutura *Error* do serviço e uma para caso de sucesso, que recebe a estrutura de dados.
6. No caso de ser lançada uma exceção pela plataforma *OutSystems*, o fluxo a decorrer é este. A exceção é registada no sistema de *logs*.

4.2.3 Implementação da camada intermédia

Este módulo, Caixa_CS no contexto do projeto, é o intermediário entre a camada da interface e a camada previamente explicada. Se a aplicação for relativamente complexa, isto é, bastante lógica no

tratamento de *inputs* ou *outputs*, é aconselhável que esse desenvolvimento seja feito neste módulo de modo a deixar o(s) módulo(s) de interface independentes de lógica.

Nesta secção é apresentado apenas o encapsulamento de um serviço em que a lógica necessitou de funções auxiliares, juntamente com duas funções criadas para lidar com tratamento de *inputs.*

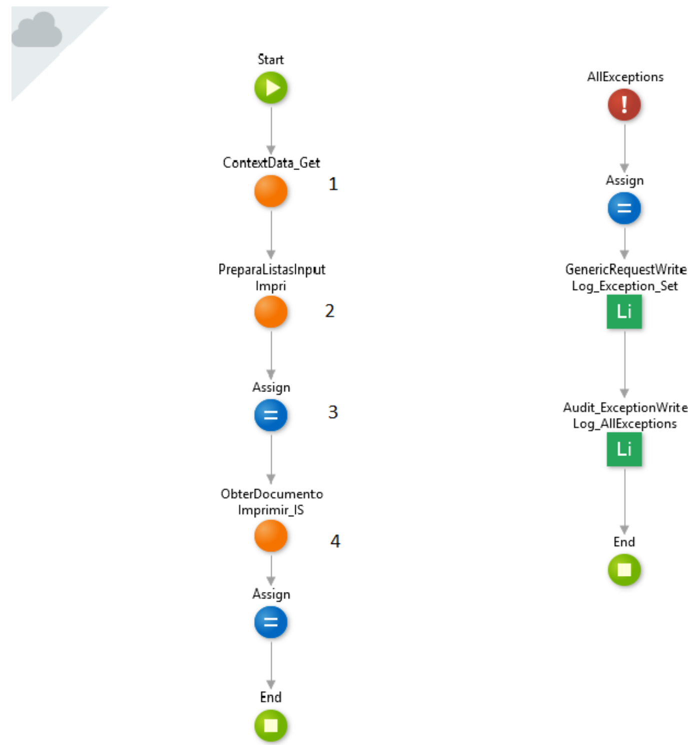

Figura 4.17 Implementação do serviço na camada intermédia na plataforma *Outsystems*

Legenda:

1. Função disponibilizada anteriormente para obter o *SessionId,* identificador da sessão e parâmetro de entrada para os serviços usados;
2. Função auxiliar criada para criar as listas com campos e valores necessárias como *input* do serviço de impressão;
3. Mapeia a lista devolvida pela função anterior para o *input* do serviço;
4. Chamada ao serviço previamente criado no módulo Caixa_IS e respetivo mapeamento para as variáveis de *output.*

```
- <array name="ListaIdentificador" type="record" depth="1">
  - <record javaclass="com.wm.util.Values">
      <value name="Campo">LstQuestionResponse</value>
      <value name="Valor" />
    </record>
  - <record javaclass="com.wm.util.Values">
      <value name="Campo">IDPacote</value>
      <value name="Valor">00000000-0000-0000-0000-000000000000</value>
    </record>
  - <record javaclass="com.wm.util.Values">
      <value name="Campo">IndDevolverDados</value>
      <value name="Valor" />
    </record>
  - <record javaclass="com.wm.util.Values">
      <value name="Campo">TipoModelo</value>
      <value name="Valor">ND</value>
    </record>
```

Figura 4.18 Exemplos de identificadores, com respetivos campos e valores, do serviço de impressão

Foram criadas duas funções para ajudar no tratamento do fluxo de dados. Uma delas, referida na listagem anterior, é responsável por preparar uma lista de identificadores para o serviço de impressão (figura 4.18) do documento CallWSDOD11_ObterDocumentoImprimir (código do documento, modelo de dados e alguns indicadores booleanos) e uma segunda responsável por receber a lista de cheques do módulo de interface e filtrar pelo campo *isSelected* de cada cheque, que tem o valor da *checkbox* na tabela, definida na análise funcional, secção 4.2.1. Se o valor for *True*, então este cheque é adicionado à lista de output usada pelo serviço de impressão WSCX015_CxBOImprimirListaCheques.

4.2.4 Implementação da interface

Esta secção foi divido em três partes, uma para cada interface. No início de cada secção, é apresentada uma imagem do aspeto final do ecrã migrado e o que acontece na ação *preparation* de cada um (ação que ocorre imediatamente antes do carregamento do ecrã), seguido das respetivas implementações e decisões. Todos os dados que aparecem nos ecrãs são de ambientes de desenvolvimento e/ou teste. Para o aspeto visual dos ecrãs foram utilizados módulos Tema do cliente da empresa desenvolvidos anteriormente, para garantir homogeneidade entre as aplicações produzidas.

Listagem de Cheques

Como definido na secção 4.2.1, este ecrã apresenta uma vista com os cheques disponíveis do utilizador em sessão. A partir de algumas ações, é possível gerar listas de cheques para mais tarde, no ecrã de confirmação de lista de cheques, serem dados como tratados. Podem existir dois tipos de cheque: multibanco ou TPA.

No *preparation* deste ecrã, que recebe apenas um *input* por *query string*, é mapeado o valor para uma variável local TipoLista. Após isso, é chamada a ação definida na camada de lógica Caixa_CS que contém um dos serviços disponibilizados, WSCX014_CxBOOpenListaCheques, onde é passado o utilizador. Este serviço retorna a lista de contas de delegação, em que cada conta contém informações como as contas bancárias disponíveis, banco associado e respetivos titulares.

Além disso, em cada delegação do banco existe um campo que indica qual é a conta bancária por omissão, através do ContaDefault (valores possíveis são S para a conta default, N para as restantes) e

que é apresentada como primeiro elemento na *checkbox* conta bancária. Esta lista é filtrada através de uma função auxiliar e devolve a conta bancária com o campo ContaDefault com o valor S. No seguimento do fluxo é guardada na variável local ContaBancaria.

Em seguida é chamado o serviço WSCX013_CxBOSelListaCheques, responsável por retornar a listagem de todos os cheques disponíveis. Recebe como *input* o utilizador, o número da conta bancária por omissão, o tipo de seleção (por omissão é "Próprio", associado à *checkbox* listagem global), e o tipo de lista. Após a chamada com sucesso, o *output* é mapeado para uma variável local OpenSelListaChequesOutput e a lista que contém os cheques é filtrada pelo atributo *isSelected*, através de uma função auxiliar GetChequesSelecionados. Nesta ação, a lista é iterada e todos os cheques que satisfaçam a condição, os seus valores são adicionados à variável ValorCheques da estrutura local ChequesSelecionados e é incrementado uma unidade à variável NumCheques da mesma estrutura.

Por último, é usado uma função auxiliar GetNumeroTotalCheques, idêntica à anterior, mas que apenas itera a lista sem qualquer tipo de condição, adicionando os valores e quantidades à estrutura local TotalCheques. Ambas as estruturas locais são usadas para popular os campos no ecrã imediatamente abaixo da tabela, que indicam o total de cheques e valor, para os cheques selecionados e todos, respetivamente.

Figura 4.19 Página de lista de cheques, após a migração para *Outsystems*

A partir da *combobox* assinalada pelo número 1 na figura 4.19, o utilizador pode alterar a conta bancária a que as listas de cheques criadas são associadas. Os valores possíveis, tal como explicado anteriormente, são originários de uma lista devolvida pelo serviço no carregamento da página. Esta lista é associada à *combobox* através do atributo *Source Record List*. Os campos *Source Attribute* e *Source Itentifier Attribute* são os valores que são mostrados no ecrã, e guardados na variável, respetivamente. Neste caso, ambos os campos assumem valores iguais, e é guardado dentro da estrutura local ContaBancaria, no campo NrContaBancaria.

Cada vez que o utilizador carrega na *combobox* e altera o seu valor, é espoletada a ação *OnChange*, que executa a ação associada. Neste caso a função implementada, OnChangeContaBancaria, faz uma nova chamada ao serviço WSCX013_CxBOSelListaCheques, mas com o *input* conta bancária que o utilizador escolhe. O novo *output* é então atualizado para a variável OpenSelListaChequesOutput, e atualizada com a conta bancária, ao invés da que é retornada por omissão do serviço. Por último,

elementos como total e valor dos cheques e a tabela são atualizados com uma funcionalidade do *OutSystems*, *Ajax Refresh*, que apenas refresca os elementos em específico ao invés de recarregar a totalidade da página.

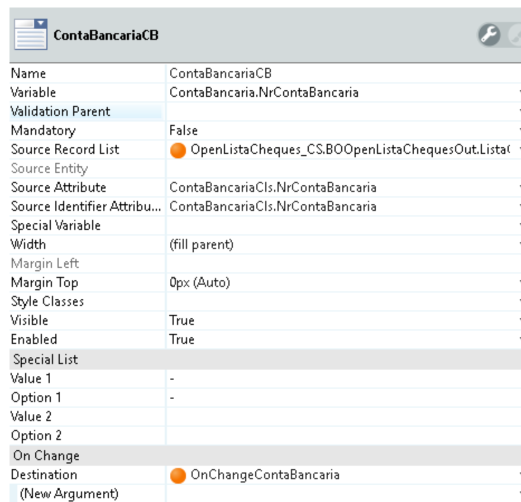

Figura 4.20 Combobox populada com a lista do serviço que retorna as contas bancárias

O valor numerário, assinalado pelo número 2, é o campo não obrigatório associado à variável local ValorNumerario e que guarda o valor. No entanto, este campo tem a particularidade de estar apenas visível nas listas de cheques e omisso nas listas de TPA's, pelo que a *Label* e o campo onde o utilizador introduz o valor estão encapsulados dentro de um *If* que valida o tipo de lista que vem como *input* no carregamento inicial da página. Foi usado também um *Web Block* (pequenos blocos que são reutilizáveis em diversos módulos, dentro ou fora da aplicação) disponibilizado pelo cliente que mascara o *input,* tendo opções como definir o tipo de moeda, casas decimais ou o separador entre o numerário inteiro e o numerário decimal.

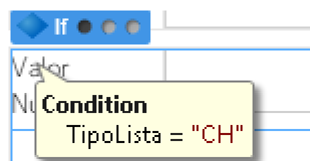

Figura 4.21 Exemplo do *Widget If* usado na própria interface

No terceiro e quarto ponto temos duas *checkboxes* associadas às variáveis locais Definitiva e ListagemGlobal. Ambas têm implementada a funcionalidade *OnChange,* executando as ações ConverterDefinita e OnChangeListagemGlobal. Visto a *checkbox* estar por norma associada a valores booleanos (*True* se estiver selecionada ou *False* caso contrário), o intuito da primeira ação é converter esse valor para "S" ou "N". No segundo caso, além da conversão para "Próprio" ou "Todos, executa novamente o serviço WSCX013_CxBOSelListaCheques de modo a refrescar os cheques da tabela. Tal como acontece com a mudança da conta bancária, a variável local OpenSelListaChequesOutput é atualizada com o novo *output* e são utilizados *Ajax Refresh* para atualizar os elementos necessários no ecrã.

O ponto seguinto refere-se à seleção de todos os cheques, ou nenhum cheque, através de uma única ação. Esta *checkbox*, colocada no início do cabeçalho da tabela, tem associada a variável local booleana SelecionarTodos, que alterna os valores entre *True* e *False*. Na sua funcionalidade *OnChange*, é chamada a função SelecionarAll, que começa por iterar a tabela e colocar o mesmo valor da variável local SelecionarTodos no campo isSelected de cada cheque, de modo a alternar entre os cheques todos selecionados, ou nenhum. Após a iteração, a variável que guarda o total do numerário e quantidade de cheques selecionados é reiniciada, colocando ambos os campos ValorCheques e NumCheques a zero. É chamada a função que calcula estes valores para posteriormente refrescar os elementos do ecrã com os valores corretos. Idêntico a este comportamento, o ponto seis refere-se à seleção individual de cada cheque da tabela. A *checkbox* de cada linha não está associada a uma variável local, mas ao campo isSelected de cada entrada da tabela. Como a quantidade e numerário de cheques estão diretamente relacionados com este requisito, na funcionalidade OnChange é usada a função auxiliar GetTotalChequesSelecionados, que começa por reiniciar aos campos NumCheques e ValorCheques da estrutura local ChequesSelecionados. De seguida é iterada a totalidade da tabela, e é validado o campo isSelected de todas as entradas. Se a condição for verdadeira, então é adicionada uma unidade ao campo NumCheques, e adicionado o seu valor ao campo ValorCheques, ambos da estrutura local ChequesSelecionados. Por último, são refrescados os elementos da página que sofreram possíveis alterações neste processo, como por exemplo os elementos do ponto sete.

Em *OutSystems*, tal como as *checkboxes* ou *comboboxes*, os botões também podem ter eventos associados. No caso do botão imprimir, foi criada uma função para ser associada ao evento OnClick, que é obrigatório. Esta ação começa por fazer a chamada ao serviço WSCX015_CxBOImprimirListaCheques, que necessita de *inputs* como o número da conta bancária, se é definitivo, o valor do numerário, o tipo de lista e a lista de cheques que foram selecionados. Se na chamada ao serviço for retornado um erro, então a ação acaba. Caso contrário, o fluxo continua e é feito uma nova chamada ao mesmo serviço usado na preparação do ecrã, WSCX013_CxBOSelListaCheques. Isto porque se o serviço de impressão ocorrer com sucesso, os cheques que foram enviados passam a estar numa lista de cheques e não são mais retornados pelo serviço inicial. A variável que guarda o *output* do serviço é atualizada com os novos valores e todo o processo de refrescamento dos elementos do ecrã repete-se. No final do fluxo do botão, é chamado ainda um segundo serviço do motor de impressão, que retorna um endereço URL para uma página com as informações dos cheques impressos. Contudo, este serviço tem a particularidade de ter um atributo XML como *input,* contendo toda a informação necessária para gerar o documento. Para tal, foi criada uma estrutura local com os mesmos campos que o XML do serviço, e feito o mapeamento do *output* do serviço de impressão para esta nova variável.

Figura 4.22 Estrutura criada em *Outsystems* para converter em XML

Após o mapeamento, é necessário converter a estrutura do *OutSystems* para uma estrutura XML. A função RecordToXml de uma extensão disponibilizada no *Forge*, XmlRecords, recebe um objeto como input e devolve-o em linguagem XML. Através da função nativa ToObject do Service Studio, é possível converter qualquer tipo de dados para um objeto. Por último, caso a chamada ao serviço do motor de impressão ocorra com sucesso, é aberta uma nova página no browser com o documento usando uma função, também ela nativa de *OutSystems*, RunJavaScript.

```
"window.open('" +
ObterDocumentosImprimir_CS.ObterDocumentoImprimirOut.ListaDocumento.Current.Lista
Documento.DodURL + "', '_blank');"
```

Figura 4.23 Comando JavaScript para abrir a página num novo separador do browser

A implementação da ordenação das colunas vai ser descrita na secção seguinte, visto ter sido utilizado o mesmo método em ambos os casos.

Confirmação de lista de cheques

Neste ecrã são apresentadas as listas de cheques geradas com sucesso pelo ecrã implementado na secção anterior. Tal como foi definido no capítulo 4.2.1 de análise funcional da aplicação, os botões de confirmação e anulação foram retirados do ecrã e acrescentado um comportamento semelhante associado a cada entrada da tabela.

Na preparação desta página apenas é feita uma chamada ao serviço que devolve uma lista de lista de cheques, WSCX011_CxBOOpenConfirmarListaCheques, tendo o utilizador como único *input*. Cada elemento contém todas as informações referentes a cada lista, como data em que foi criada, número de

documentos, o estado de confirmação ou o tipo de lista de cheques (multibanco ou TPA's). Após a chamada com sucesso, o *output* do serviço é mapeado para uma estrutura local OpenConfirmarChequesOutput. Para popular o *widget Table Records* do Service Studio é passada esta variável.

Figura 4.24 Página de confirmação de lista de cheques, após a migração para Outsystems

Para o primeiro ponto foi acrescentada uma última coluna que pode conter, ou não, uma ligação a uma ação, algo semelhante ao descrito sobre a funcionalidade *OnClick* dos botões em *OutSystems*. A visibilidade desta ligação "Anular" está diretamente relacionada com o estado da lista de cheques, isto é, a lista de cheques não pode estar confirmada para ser possível a anulação da mesma. Como tal, as entradas da tabela que estão confirmadas não podem ser anuladas. A ação auxiliar de nome "Anular" associada à ligação começa por fazer uma chamada ao serviço WSCX016_CxBOAnularConfirmarListaCheques, em que o *input* é a lista de cheques selecionada. Após o sucesso do *Web Service*, é então refeita a chamada ao serviço inicial WSCX011_CxBOOpenConfirmarListaCheques para ser atualizada a lista de lista de cheques. De notar que ao anular uma lista de cheques que não esteja confirmada esta é apagada e todos os cheques associados voltam a aparecer disponíveis no ecrã implementado anteriormente. Por último, a variável local que guarda a lista de listas de cheques é atualizada e é feito um *Ajax Refresh* na tabela para mostrar os novos dados.

O ponto número dois foi uma nova implementação relativamente ao ecrã anterior em que o botão "Confirmar" desapareceu e o comportamento necessário ao correto funcionamento da página é espoletado através da ação associada à *checkbox* na coluna Confirmado. Esta ação auxiliar de nome Confirmar tem a particularidade de executar dois fluxos complemente diferentes consoante o estado da *checkbox*. No caso em que a lista de cheques não está confirmada e se confirma, o fluxo é a chamada ao serviço WSCX012_CxBOConfirmarListaCheques que recebe como *input* a lista de cheques assinalada. Ao contrário do serviço de anulação que não devolve a lista e como tal é necessária a chamada ao serviço descrito inicialmente na preparação da página, o serviço de confirmação tem como *output* a lista de lista de cheques com o estado de confirmação atualizado. Como tal, não é necessário proceder-se a uma invocação do serviço novamente, mapeando o *output* diretamente para a variável local OpenConfirmarChequesOutput e procedendo ao refrescamento da tabela com a função *Ajax Refresh*. Se a lista estiver confirmada, então o fluxo consiste em executar o serviço de anulação, que além de anular por completo a lista de cheques quando esta não está no estado confirmado, tem a particularidade de o anular. Na continuação do fluxo o comportamento é idêntico ao descrito na função associada à anulação: é feita uma nova chamada ao serviço WSCX011_CxBOOpenConfirmarListaCheques e a variável local é atualizada. Antes do fim da ação, é feito um refrescamento da tabela com os novos dados.

Como último requisito deste ecrã foi implementada a ordenação das colunas. A plataforma do *OutSystems* tem por omissão uma funcionalidade para este efeito, mas devido a comportamentos inconsistentes quando se executam *Ajax Refreshes* e a falta de persistência da ordenação cada vez que se refrescava o ecrã, foi implementada uma ordenação de colunas específica para este projeto. Foram acrescentadas duas variáveis locais neste ecrã. A variável booleana isSorted para saber o tipo de ordenação (ascendente ou descendente) na tabela e a estrutura Sorting com três atributos: keepSorting, que indica se a ordenação é para se manter, indexColuna que indica qual a coluna pela qual a tabela está ordenada e NomeColuna, que indica o nome da coluna. Usando o *Web Block* disponibilizado pelo Service Studio List_SortColumn, foi implementada no seu evento OnNotify (é a forma como o *Web Block* comunica com o *parent*, neste caso o ecrã principal) uma função auxiliar OnNotifySort, que recebe como *input* o índice da coluna pelo qual se quer ordenar a tabela. No início da ação o valor da variável isSorted é invertido e o *input* recebido é colocado na estrutura Sorting, no atributo IndexColuna. Após isso foi feita uma função auxilar para converter os índices da coluna no nome correspondente. Dentro desta ação, o campo keepSorting da estrutura local Sorting é atualizado para *True*¸e todos os seus campos estão preenchidos. Por último foi usado a função SortRecordList, da extensão disponibilizada no *Forge* SortRecordList, que recebe os seguintes *inputs*: recordList, sortBy e isAscending. Os parâmetros passados são a lista de listas de cheques OpenConfirmarChequesOutput.ListaCheques, a variável NomeColuna e isSorted, respetivamente. Após o sucesso da ação está garantida a ordenação da lista e é apenas necessário refrescar a tabela, para propagar as mudanças no ecrã. Como a ordenação não se mantém após a chamada de serviços, em todas as ações que alteram a lista de listas de cheques (preparação, anulação e confirmação) é validado o campo keeSorting. Se este for *True,* então também é chamada a função da extensão SortRecordList.

Cobranças do mediador

Neste ecrã são carregados os lotes disponíveis do utilizador, onde o mediador pode registar pagamentos que recebeu dos seus clientes e associar a recibos e/ou apólices geradas previamente pelo sistema. Tal como na primeira página, existem três variáveis de input para o carregamento: o diário, o ano e o NrLote. No início da preparação do ecrã é chamado o serviço WSCX018_CxBOGetLote, que recebe como *input* o utilizador e o tipo de seleção que para este caso é sempre uma *String* com o valor "Abertos". Isto indica ao serviço que o utilizador apenas quer os seus lotes que ainda não tenham sido fechados. O serviço retorna uma lista de todos os lotes disponíveis. Caso as três variáveis iniciais estejam todas preenchidas, então a lista de lotes é iterada à procura do lote com o valor igual ao da variável local NrLote. Caso haja um lote na lista que satisfaça a condição, então a variável local LoteAux que guarda as informações sobre o lote selecionado é atualizada. Após a iteração sobre a lista, é chamado o serviço WSCX017_CxBOGetCobrancasBackOffice que recebe como *inputs* o ano, o número do lote, o tipo de seleção ou utilizador. Se a chamada ocorrer com sucesso, a variável local ListaCobrancasAux guarda o output do serviço. Por último, cada cobrança tem associado um número de lote e de número de mediador. Por último, foram adicionados dois campos à estrutura das Cobranças, que indicam se houve alterações no registo e o tipo de ação a executar no serviço guardar, explicado em detalhe mais à frente.

Um dos requisitos definidos para esta página era a necessidade da tabela ser totalmente editável, isto é, apagar, acrescentar ou editar registos. Tal como tinha acontecido na página de confirmação de lista de cheques em que algumas funcionalidades da plataforma funcionavam de forma inconsistente, o uso do *widget* Editable Table que em teoria satisfaria este requisito demonstrava um comportamento erróneo, levando a muitas das tentativas a espoletar erros da plataforma no ecrã. Desta forma, foi necessária a criação de algo que tivesse o comportamento esperado. Usando o *widget* Table Records

como base que o Service Studio disponibiliza com algumas limitações, foram acrescentadas as seguintes funcionalidades: criar uma entrada da tabela, editar uma entrada já presente na tabela, e apagar uma entrada da tabela.

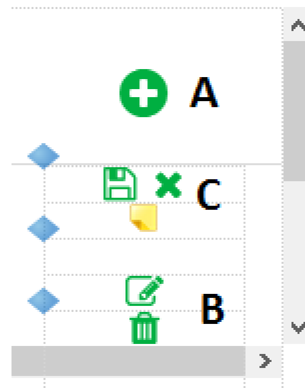

Figura 4.25 Diferentes ações consoante o estado da entrada da tabela

Para ajudar na criação destas funcionalidades foram adicionadas duas variáveis locais ao ecrã, isAddCobrança que é do tipo booleano e valida se uma nova entrada está em curso e EditingIndex que guarda o índice da linha da tabela que está a ser editada. Por omissão e quando não existe nenhuma ação a decorrer na tabela a variável EditingIndex tem o valor de -1.

A nova entrada na tabela, representada pela letra A da figura 4.25, foi feita através da criação de uma função auxiliar EditNew. Esta ação valida se a variável isAddCobrança é *True* ou *False*. Se for *True*, então a ação acaba, caso contrário são feitos mapeamentos para a variável local CobrancasAux, que guarda os valores da cobrança cada vez que uma linha sofre alguma alteração. Este mapeamento serve para colocar alguns campos da cobrança com o valor correto, como por exemplo o tipo de alteração (Nova), o tipo de ação para o serviço (I, que indica ser uma nova cobrança), o seu estado inicial ou número do lote e do mediador. Os variáveis isAddCobrança e EditingIndex têm os seus valores atualizados para *True* e 0 (zero), respetivamente. Neste fluxo é ainda feito uma ListInsert, função nativa de *OutSystems* para introduzir um elemento numa lista, da variável local CobrancasAux já com os valores corretos na lista que popula o *widget* da tabela e na lista auxiliar ListaCobrancasAux. No final da ação, é feito um *Ajax Refresh* do botão Guardar, que só fica disponível quando não existem alterações em curso na tabela ou alterações a guardar (o estado da tabela é igual ao que foi retornado pelo serviço na preparação do ecrã).

Representado pela letra B da figura 4.25, estão as ações disponíveis quando existe uma linha da tabela já criada. Sendo a primeira a edição, esta ação chamada EditRow altera a variável EditingIndex para o índex da linha em questão. Após isso a tabela é refrescada e todos os campos passam a estar disponíveis para serem editados. A segunda ação, EditDelete, serve para apagar a entrada da tabela, mas apenas visualmente. A entrada só é devidamente apagada aquando da chamada ao serviço que está associado ao botão guardar, explicado em detalhe nesta secção. A ação EditDelete é a mais complexa de todas as funcionalidades implementadas para a edição da tabela, devido às regras de negócio do cliente. Se a linha a apagar for uma nova linha, ou seja, foi acrescentada mais ainda não foi confirmada, o fluxo é relativamente simples e a cobrança é retirada da tabela e da lista de cobranças através da ação nativa ListRemove, passando o seu índice. Para finalizar, as variáveis locais EditingIndex e isAddCobrança ficam com os seus valores default, -1 e *False*, respetivamente. No outro caso em que se

quer apagar uma linha da tabela que já existia na base de dados, há quatro possíveis verificações a fazer, pela seguinte ordem:

1. Se o campo reconciliado da cobrança tiver o valor * (significa que está pendente), não é possível apagar, terminando assim ação e enviando uma mensagem para o ecrã com mensagem informativa através da ação FeedBackMessage;
2. Se o campo Reconciliado da cobrança tiver o valor X (significa que está no estado final) e SituacaoEnvio for P (significa que está pago) é chamada uma função auxiliar ValidaLotesPorRecibo. Esta ação executa novamente o serviço WSCX017_CxBOGetCobrancasBackOffice com os mesmos *inputs* utilizados na preparação do ecrã, à exceção do tipo de seleção que é “Recibo” ao invés de ser “Todos”. Esta mudança no *input* indica ao serviço que apenas é necessário retornar as cobranças com número de recibos. Após isso, a lista de cobranças é iterada e para todos os lotes com um número que o identifica diferente, é validado se o recibo pertence a esse lote. Em caso afirmativo, e se o lote estiver encerrado, então a mensagem retornada no *output* desta função é atualizada. Se não existir noutro lote, a mensagem de *output* é uma *String* vazia. Após isto, e retornando ao fluxo da função EditDelete, é validada a mensagem que vem do *output* da função ValidaLotesPorRecibo e podem acontecer duas situações:
    a. Se a mensagem for uma *String* vazia, significa que o recibo não se encontra noutro lote e pode ser anulado. Os campos Reconciliado e SituacaoEnvio passam a ter os valores # (significa reconciliado) e A (significa anulado), respetivamente. No entanto estes valores só são confirmados quando se guarda efetivamente na base de dados através de um serviço, pelo que é preciso preparar os campos necessários que indicam que houve alterações. Os campos Accao e Alteracoes da cobrança são atualizados com os valores U (o serviço irá interpretar como *update*) e “Alterações”. Este último campo não é enviado para o serviço, sendo utilizado única e exclusivamente para validar se é necessário disponibilizar o botão que contém o serviço para guardar as alterações. No final do fluxo a tabela é atualizada para refletir os novos dados através de *Ajax Refresh*, e é apresentada uma mensagem de *feedback* ao utilizador a informar que o recibo se encontra anulado.
    b. Se a mensagem for diferente da *String* vazia, significa que o recibo está contido noutro lote que já foi encerrado. Neste caso, essa mesma mensagem é apresentada ao utilizador através da função FeedBackMessage, onde é apresentado o lote em questão.
3. Se o campo Reconciliado da cobrança tiver o valor # (significa reconciliado) e SituacaoEnvio for A (significa que está anulado), então cancela-se a anulação. Para tal, os campos Reconciliado, SituacaoEnvio e ComentarioAnulacao são atualizados para X (estado final), P (pago) e *String* respetivamente. Como nos outros fluxos descritos anteriormente, a tabela é atualizada e é feito o refrescamento do ecrã.
4. No último caso, em que o recibo pode ser efetivamente eliminado, foi utilizada a função nativa ListRemove para remover da tabela a linha em questão. No entanto, como explicado anteriormente, apenas é apagado visualmente. Para tal ser confirmado na base de dados, o campo Accao é atualizado com o valor D (o serviço interpreta como *delete*). As variáveis locais EditingIndex e isAddCobrança são também atualizadas com os valores default.

De notar que em todas as situações descritas a variável local ListaCobrancasAux é também mapeada com os valores atualizados na tabela e antes da remoção dos elementos, para ficarem guardadas todas as alterações e evitar a perda de dados. Os *Ajax Refresh* na tabela também são feitos após esta etapa, para visualmente a linha parecer apagada para o utilizador, mas internamente na lógica da aplicação continuar disponível.

Por último, representado pela letra C da figura 4.25, são as ações disponíveis quando a entrada da tabela está a ser editada. Sendo a primeira a ação EditSave, é começado por verificar se os *inputs* do número de apólice e número de recibo têm caracteres em branco. Para tal é utilizada a ação nativa Trim, que recebe como *input* dados do tipo *Text* e retorna esse mesmo *input* sem caracteres desnecessários. Para ser possível guardar as alterações na base de dados, é obrigatório a cobrança ter um número de recibo válido associado. Se, ao tentar gravar a entrada da tabela, o valor se encontrar vazio, a ação acaba logo e é apresentada uma mensagem de *feedback* ao utilizar. De seguida existem duas possibilidades: ou se está a gravar uma linha que já existia na tabela, ou uma linha que acabou de ser inserida. De modo a perceber qual das situações é necessário tratar, existe uma condição que valida se a variável isAddCobrança tem o valor *True* e o campo Alteracoes da cobrança tem o valor "Nova". Se a entrada da tabela já existia, então iguala-se a ListaCobrancasAux à tabela e o campo Acçao da tabela é atualizado para U (o serviço interpreta como *update*). No cenário em que a entrada é nova, então faz-se um ListInsert do elemento da tabela na variável auxiliar ListaCobrancasAux, para ambas as listas terem a mesma informação. Por fim, são refrescados a tabela e o botão. A última ação tem o nome de EditCancel, que serve para cancelar uma edição. O fluxo desta função passa apenas por colocar as variáveis locais EditingIndex e isAddCobrança no seu estado inicial, e caso seja uma entrada da tabela que acabou de ser inserida, é removida através do ListRemove.

Figura 4.26 Página de cobranças do mediador, após a migração para Outsystems

A migração começou pela implementação da *checkbox,* visto ser uma peça fundamental para a navegação do ecrã. A *Source Record List* que abastece este *widget* é a lista retornada pelo serviço WSCX018_CxBOGetLote e o evento *OnChange* chama a função OnChangeLote, que guarda os valores das variáveis Ano, Diario e NrLote. No fim da ação é recarregado o ecrã com estes novos valores, através de *query string*. Como foi explicado, na preparação do ecrã caso estes três campos estejam preenchidos, é chamado o serviço que devolve a lista de cobranças. A partir deste momento, a tabela fica visível e são mostrados os dados que foram retornados do serviço WSCX017_CxBOGetCobrancasBackOffice.

Na figura 4.26 é dado o exemplo de duas entradas da tabela, uma linha que foi adicionada e está em modo de edição e uma linha que já existia no lote. As validações que indicam se o número de apólice

e o número do recibo são feitas não na interface, mas apenas quando se chama o serviço. No entanto existem validações ao nível da interface, para evitar chamadas desnecessárias aos serviços. Por exemplo, o preenchimento dos tipos com determinados valores pode implicar a obrigatoriedade de certos valores ou campos, figura 4.26 ponto 2 e 3. No caso do tipo de Apólice, foi implementado o evento OnChange da caixa de *input,* que valida se o que o utilizador introduziu é um A (apólice normal) ou R (apólice-recibo). Se o introduzido for diferente, então é apresentada uma mensagem ao utilizador a indicar que só estes dois valores são possíveis e o campo é apagado. Se o utilizador introduzir um R, então o tipo de recibo é obrigatoriamente P (recibo) e os campos Data e Valor ficam disponíveis para preenchimento, sendo ambos de preenchimento obrigatório. Se o utilizador introduzir um A todos os campos nas colunas seguintes são reiniciados, para evitar o preenchimento de campos que não são necessários neste tipo de apólice. Para o tipo de recibo, apenas podem ser introduzidos quatro valores: E (estorno), P (recibo), X (entrada) e Z(prémio).

No ponto quatro estão as ações possíveis para a edição da tabela: adicionar nova linha, depois gravar ou cancelar a edição de uma linha, e por último editar ou apagar uma linha.

Por fim foi implementado o botão guardar. De reforçar que esta ação só deve estar disponível caso não existem alterações ou linhas no estado de edição na tabela. Para garantir tal requisito, na propriedade *Enabled* (é do tipo booleana, ficando visível se for *True* ou escondida se for *False*) do botão foi acrescentada a condição da figura 4.26. A função auxiliar ExistemAlterações itera a lista que recebe como input, e retorna um booleano se algum elemento tiver o campo Accao com o valor N (não existem alterações). Na lógica do botão, é filtrada a lista ListaCobrançasAux pela condição que a cobrança não tem os campos Accao e Alteracoes com os valores D e Nova, respetivamente. Na prática esta validação remove da lista as cobranças que foram introduzidas e apagadas da lista, antes de serem guardadas, garantindo assim que apenas as cobranças novas ou que já existiam e foram apagadas, e atualizações são enviadas para o serviço. No final do fluxo, o serviço WSCX019_CxBOCobrancasMediadorGuardar é chamada com a lista filtrada e o ecrã é recarregado com os novos *inputs*.

4.3 Testes dos desenvolvimentos

Um processo mandatório no desenvolvimento de *software*, qualquer que tenha sido a tecnologia utilizado, é a execução de testes.

No decorrer de ambos os projetos, foram feitos testes unitários, antes de dado por completo o trabalho de desenvolvimento. Contudo, sendo projetos de caracter diferente, foram adotados testes diferentes.

Por norma os projetos entram numa primeira fase de testes de qualidade, realizados por uma equipa de testes externa, que assegura o funcionamento sem erros. O *feedback* é dado através de uma plataforma interna, onde os *bugs* são assignados aos responsáveis do projeto, que por sua vez os distribuem à equipa que fez o desenvolvimento. Após a correção dos problemas, estes são devolvidos à pessoa que reportou, para voltar a retestar o seu caso de teste. De forma a ver a evolução dos projetos e consoante a duração dos testes, existem relatórios diários ou semanais, feitos por esta equipa, nos quais são apresentados números como totais de *bugs* abertos, *bugs* por corrigir, *bugs* fechados ou *bugs* para os quais ainda não é possível efetuar novamente testes. Após o sucesso, o projeto entra em fase de testes de aceitação, por norma efetuados pelas pessoas do cliente responsáveis pelo projeto, onde são efetuados

testes para garantir o correto funcionamento dos produtos (regras de negócio, por exemplo). Este modelo de testes foi usado para a migração da aplicação Caixa.

Na entanto, apesar do primeiro desenvolvimento da aplicação de gestão de horas e projetos ser interno à equipa no cliente, os testes foram efetuados por algumas pessoas da equipa que se voluntariaram em testar, devido ao trabalho em duplicado: reportar as horas no método tradicional (ficheiro Excel) e na aplicação.

Capítulo 5 Conclusões e trabalho futuro

5.1 Conclusões

O trabalho apresentado neste documento foi desenvolvido ao longo de um período de acolhimento pela Unipartner, em parceria com a Faculdade de Ciências da Universidade de Lisboa. O objetivo inicialmente proposto foi a formação na tecnologia *Outsystems*, visto ser o primeiro impacto com a tecnologia, para posterior integração nos desenvolvimentos da equipa no cliente.

Após a conclusão das formações, houve a oportunidade de inserção num projeto interno, para melhorar a gestão de horas das pessoas da equipa. Rapidamente foi percebido que o desejado pela aplicação era mais complexo do que inicialmente se pensava, comparativamente ao modelo de gestão de horas e recursos anterior, o Excel. Neste projeto, os requisitos iniciais eram vagos, e o primeiro obstáculo foi a construção de um modelo de dados que suportasse os requisitos funcionais. Ao longo das diversas iterações com os requerentes do projeto, o caminho a percorrer ficou mais nítido. Contudo, é percetível que quando não existem ideias bem definidas inicialmente, as alterações a fazer de *sprint* para *sprint* podem ser notórias, como foi o caso do modelo relacional que sofreu bastantes alterações no crescimento da aplicação, o que acabou por ter um impacto relativamente grande nas restantes funcionalidades. Os testes desta aplicação para integração total na equipa foram feitos por alguns elementos que além de reportarem as horas nos ficheiros habituais tinham o trabalho duplicado em reportar na aplicação, para garantir a coerência da informação.

Na segunda parte deste documento, na qual é descrito um projeto onde os requisitos estavam mais explícitos, o desenvolvimento ocorreu de uma forma mais natural. Existiu a oportunidade de analisar tecnologias e trabalhar com pessoas de equipas diferentes, o que permitiu obter uma visão mais abrangente de como funciona uma empresa de informática, mas sobretudo trabalhar com diferentes áreas do negócio. Este projeto, após passar por testes de qualidade e aceitação por equipas específicas do cliente, encontra-se no ambiente produtivo.

Em suma, foi uma primeira experiência profissional positiva onde houve oportunidade de empregar os conhecimentos adquiridos no curso de Mestrado em Engenharia Informática no mundo real.

5.2 Trabalho Futuro

A aplicação da gestão das horas tem as funcionalidades iniciais pedidas a funcionar, e neste momento serve como primeiro contacto para as novas pessoas da equipa que integram a equipa de *Outsystems.* Existem ainda algumas funcionalidades a implementar, como um pequeno fórum por cada projeto onde os utilizadores podem colocar comentários ou dúvidas sobre o projeto em si ou as diferentes fases. Uma alteração importante a realizar é uma revisão do modelo relacional, devido a não haver uma ideia inicial projetada. Um requisito que poderá ser melhorado é a exportação de dados, visto ter sido utilizada apenas a função nativa, muito limitativa, de *Outsystems* que converte *records* em ficheiros. Existem algumas extensões em *Java* e *C#* que poderão vir a ser usadas e alteradas para permitir a análise de dados e estatísticas de uma forma mais complexa e útil.

A aplicação Caixa, que neste documento apenas tem três páginas completamente migradas para *Outsystems* e em uso, é de uma extensão e complexidade grandes. No futuro poderão existir mais projetos para migrar páginas desta linha de negócio, e este projeto poderá ser útil para os responsáveis por esse desenvolvimento.

Bibliografia

[1] Schiller, Jochen, and Agnès Voisard, eds. *Location-based services*. Elsevier, 2004.

[2] Gustavo, Alonso, et al. "Web Services: concepts, architectures and applications." (2004).

[3] Performance comparison of SOAP and REST based Web Services for Enterprise Application Integration

[4] Yang, Zhonghua, and Keith Duddy. "CORBA: a platform for distributed object computing." SIGOPS Operating Systems Review 30.2 (1996): 4-31.

[5] Birrell, Andrew D., and Bruce Jay Nelson. "Implementing remote procedure calls." ACM Transactions on Computer Systems (TOCS) 2.1 (1984): 39-59.

[6] Tay, Beng Hang, and Akkihebbal L. Ananda. "A survey of remote procedure calls." ACM SIGOPS Operating Systems Review 24.3 (1990): 68-79

[7] Web Services Description Language (WSDL) 1.1

[8]https://hosteddocs.emediausa.com/new-forrester-lowcode-development-wave-report-2017.pdf (Outubro 2018)

[9]http://www.scrummaster.dk/lib/AgileLeanLibrary/People/MikeBeedle/Scrum%20Patterns%20-Beedle.PDF (Fevereiro 2019)

[10] Boehm, Barry. "Get ready for agile methods, with care." Computer 1 (2002): 64-69.

[11] Rising, Linda, and Norman S. Janoff. "The Scrum software development process for small teams." IEEE software 17.4 (2000): 26-32.

[12] Fox, Armando, et al. "Above the clouds: A berkeley view of cloud computing." Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS 28.13 (2009): 2009.

Anexos

Anexo A

Tabelas das entidades do modelo relacional da aplicação de gestão de horas

EquipasTarefa	
Atributo	**Tipo**
ID	Long Integer
TarefaID	Tarefa Identifier
hasLGI	Boolean
hasOutsystems	Boolean
hasMicrosoft	Boolean
hasSalesforce	Boolean
custeioLGi	Decimal
custeioOutsystems	Decimal
custeioMicrosoft	Decimal
custeioSalesfoce	Decimal

Milestone	
Atributo	**Tipo**
ID	Long Integer
FaseId	Fase Identifier
Descricao	Text
DataPrevista	Date
DataAceitacao	Date
Valor	Decimal
isPercentage	Boolean
Custeio	Decimal

Tarefa	
Atributo	**Tipo**
ID	Long Integer
FaseId	Fase Identifier
Descricao	Text
DataPrevista	Date
Custeio	Decimal
Percentagem	Decimal

Fase	
Atributo	**Tipo**
ID	Long Integer
ProjetoID	Projeto Identifier
Nome	Text
CusteioEquipa	Decimal
DataPedidoCusteio	Date
DataAprovacaoEstimativa	Date
DataConclusaoEstimativa	Date
EstimativaConsumida	Decimal
EstimativaPorConsumir	Decimal
MicrosoftHoras	Decimal
SalesforceHoras	Decimal
LGIHoras	Decimal
OutsystemsHoras	Decimal
DataInicioDV	Date
DataFimDV	Date
DataInicioTAQ	Date
DataFimTAQ	Date
DataInicioTAA	Date
DataFimTAA	Date
DataPR	Date
DataPrevistaEntregaTA	Date
DataPrevistaPR	Date
Descricao	Date
DataInicioReal	Date
DataFimReal	Date
TotalHoras	Decimal
Estado	Estado Identifier
Observacoes	Text
TLID	TeamLeader Identifier
PrioridadeID	Prioridade Identifier
PMOEngagManager	Text
OrigemPedidoID	OrigemPedido Identifier
EstadoAFID	EstadoAF Identifier
TipoPedidoID	TipoPedido Identifier
TrimestreID	Trimestre Identifier
InicioPosArranque	Date
FimPosArranque	Date
DataInicioPrevista	Date
DataFimPrevista	Date
TipoServicoID	TipoServico Identifier
Faturado	Text
PreContratoID	PreContrato Identifier

ListaMilestonesAux	Text
PendenteOutraEquipa	Text
EmEsclarecimentoCliente	Text
SuspensoCliente	Text
Aprovado	Text
ImplementadoCliente	Text
Cancelado	Text
ChaveNaMao	Text
EstadoAutomatico	EstadoIdentifier
EstadoAlinhado	Text
MesDesenvolvimento	Text
MesInicioTestes	Text
MesEntradaPR	Text
isOrcamento	Text
Key	Text

Projeto	
Atributo	**Tipo**
ID	Long Integer
CodigoProjeto	Text
Nome	Text

Timesheet	
Atributo	**Tipo**
ID	Long Integer
UserExtendedID	UserExtended Identifier
StartDate	Date
StatusID	TimeSheetStatus Identifier
RejectionReason	Text

Tarefa	
Atributo	**Tipo**
ID	Long Integer
FaseID	Fase Identifier
Descricao	Text
Custeio	Decimal
Percentagem	Decimal
Ordem	Text

TimeShetItem	
Atributo	**Tipo**
ID	Long Integer
TimeSeetLineID	TimeSheetLine Identifier
Date	Date
Effort	Decimal

TimeSheetLine	
Atributo	**Tipo**
ID	Long Integer
TimeSheetID	TimeSheet Identifier
MondayEffort	Decimal
TuesdayEffort	Decimal
WednesdayEffort	Decimal
ThursdayEffort	Decimal
FridayEffort	Decimal
FaseID	Fase Identifier
MondayComment	Text
TuesdayComment	Text
WednesdayComment	Text
ThursdayComment	Text
FridayComment	Text

UserExtended	
Column1	**Column2**
ID	Long Integer
EquipaID	Equipa Identifier

UserProjeto	
Column1	**Column2**
ID	Long Integer
UserExtendedID	User Extended Identifier
ProjetoID	Projeto Identifier

Anexo B

Tabelas das entidades estáticas do modelo relacional da aplicação de gestão de horas

EquipaResponsavel
Column1
Microsoft
Salesforce
LGI
Outsystems

Estado
Column1
Novo
EsclarecimentoCliente
SuspensoCliente
ParaAprovar
Aprovado
Planeado
Execucao
ImplementacaoEquipaCliente
Cancelado
Testes
EmProducao
PosArranque
Orcamentacao
PendenteOutraEquipa
ChaveNaMao
ParaAprovarChaveNaMao
Atrasado

PreContrato
Column1
Sim
Nao

EstadoAF
Column1
PorIniciar
EmElaboracao
VaiExistir
NaoVaiExistir
Especificado

OrigemPedido
Column1
Cliente
Equipa
ManutencaoCorretiva
Outros
Unipartner

Prioridade
Column1
MuitoAlta
NaoUrgente
Legal

TimeSheetStatus
Column1
Approved
Draft
Rejected
Submitted

TipoPedido
Column1
Projeto
ChangeRequest
PequenoEvol
SuporteOU
Fase
ManutencaoProativa

TipoServico
Column1
KTLO
Projeto
AssystPE
Ausencia
Ferias

Trimestre
Column1
Primeiro
Segundo
Terceiro
Quarto

Anexo C

Modelo relacional da aplicação de gestão de horas