INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de

Electrónica e Telecomunicações e de Computadores

Projecto *Timecloud*

Software de gestão de tempo laboral numa plataforma *cloud*

João Paulo Encarnação Neto

(Licenciado)

Trabalho de projecto para obtenção do grau de Mestre

em Engenharia Informática e de Computadores

Orientador

Mestre Paulo Alexandre Leal Barros Pereira

Júri

Presidente: Mestre Pedro Alexandre de Seia Cunha Ribeiro Pereira

Vogais: Mestre Pedro Miguel Henriques dos Santos Félix

Mestre Paulo Alexandre Leal Barros Pereira

Novembro de 2011

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de

Electrónica e Telecomunicações e de Computadores

Projecto *Timecloud*

Software de gestão de tempo laboral numa plataforma *cloud*

João Paulo Encarnação Neto

(Licenciado)

Trabalho de projecto para obtenção do grau de Mestre

em Engenharia Informática e de Computadores

Autor

(João Paulo Encarnação Neto)

Orientador

(Paulo Alexandre Leal Barros Pereira)

Abstract

The project aims at providing a service platform for managing and accounting work hours, allowing workers to set their work time, vacations and absences. The system enables the production of work reports and the automatic analysis of the data, e.g. excessive absences or overlapping vacations of workers. The emphasis is to provide an architecture that facilitates the inclusion of such functionalities.

The project implementation targets *Google App Engine* (i.e. GAE) in order to provide a solution based on the *Software as a Service* paradigm, providing high availability and automatic data replication. The platform's choice resulted from the analysis of the current main cloud platforms: Google App Engine, Windows Azure and Amazon Web Services. The analysis considered the features of each platform, namely programming model, data model, provided services and costs.

The result is a layered solution composed of the following modules: interface, business logic and data management. The platform's interface is designed according to the *REST* architectural constraints, and supports *JSON* and *XML* formats. It also includes an authorization component, based on Spring-Security. Authentication is performed through *Google Accounts*. In order to increase decoupling, the solution resorts to the *Dependency Injection* design pattern, which provides technology independence at the different layers.

It was also implemented a prototype to provide the current work's proof of concept. The prototype issues AJAX requests to the platform services and its implementation is simplified by the use of several javascript libraries and design patterns, such as the model-view-viewmodel, used through a data binding mechanism.

The project development used a Scrum based agile approach, in order to implement the system requirements, which were expressed in user stories. Unit tests were produced in order to ensure the quality of the developed code, and UML was used to document and analyze the implemented functionalities.

Keywords

Timesheets management software, *Cloud application*, SaS solution, *Google App Engine*, *REST interface*

Resumo

O presente projecto tem como objectivo a disponibilização de uma plataforma de serviços para gestão e contabilização de tempo remunerável, através da marcação de horas de trabalho, férias e faltas (com ou sem justificação). Pretende-se a disponibilização de relatórios com base nesta informação e a possibilidade de análise automática dos dados, como por exemplo excesso de faltas e férias sobrepostas de trabalhadores. A ênfase do projecto está na disponibilização de uma arquitectura que facilite a inclusão destas funcionalidades.

O projecto está implementado sobre a plataforma *Google App Engine* (i.e. GAE), de forma a disponibilizar uma solução sob o paradigma de *Software as a Service*, com garantia de disponibilidade e replicação de dados. A plataforma foi escolhida a partir da análise das principais plataformas *cloud* existentes: *Google App Engine*, *Windows Azure* e *Amazon Web Services.* Foram analisadas as características de cada plataforma, nomeadamente os modelos de programação, os modelos de dados disponibilizados, os serviços existentes e respectivos custos. A escolha da plataforma foi realizada com base nas suas características à data de iniciação do presente projecto.

A solução está estruturada em camadas, com as seguintes componentes: interface da plataforma, lógica de negócio e lógica de acesso a dados. A interface disponibilizada está concebida com observação dos princípios arquitecturais *REST*, suportando dados nos formatos *JSON* e *XML*. A esta arquitectura base foi acrescentada uma componente de autorização, suportada em *Spring-Security*, sendo a autenticação delegada para os serviços *Google Acounts*. De forma a permitir o desacoplamento entre as várias camadas foi utilizado o padrão *Dependency Injection*. A utilização deste padrão reduz a dependência das tecnologias utilizadas nas diversas camadas.

Foi implementado um protótipo, para a demonstração do trabalho realizado, que permite interagir com as funcionalidades do serviço implementadas, via pedidos AJAX. Neste protótipo tirou-se partido de várias bibliotecas *javascript* e padrões que simplificaram a sua realização, tal como o *model-view-viewmodel* através de *data binding.*

Para dar suporte ao desenvolvimento do projecto foi adoptada uma abordagem de desenvolvimento ágil, baseada em *Scrum*, de forma a implementar os requisitos do sistema, expressos em *user stories*. De forma a garantir a qualidade da implementação do serviço foram realizados testes unitários, sendo também feita previamente a análise da funcionalidade e posteriormente produzida a documentação recorrendo a diagramas UML.

Palavras-Chave

Software de gestão de tempo laboral, Aplicação *Cloud*, Solução *SaaS, Google App Engine*, interface *REST*

Agradecimentos

Aos familiares, pelo apoio ao longo do projecto

À Mafalda, por me aturar

A Paulo Pereira, pela orientação dada

A Hugo Raimundo, pelo apoio prestado

A todos os demais que contribuíram para o desenvolvimento do projecto

Índice

Índice de Figuras

Índice de Listagens

Índice de Tabelas

Capítulo 1

INTRODUÇÃO

1.1 CONTEXTO

O presente projecto tem como objectivo a disponibilização de uma plataforma de serviços para gestão e contabilização de tempo remunerável. Actualmente existem diversas formas de contabilização de tempo remunerável em ambiente empresarial, seja de uma forma menos interactiva, registando as horas de trabalho (como "picar o ponto" ou marcação em folhas de cálculo), seja através de soluções mais dinâmicas, onde se indicam as várias tarefas diárias, através de aplicações como *TimeReporting* (1), *toggl* (2) ou *cube* (3). Apesar de cumprirem o seu objectivo, estas aplicações são focadas numa única tarefa, como marcação de horas ou geração de relatórios. Assim, não fornecem uma solução que permita responder simultaneamente às necessidades de diversos departamentos da empresa (financeiro, gestão e recursos humanos).

O presente projecto tem o objectivo de colmatar esta necessidade, agregando as vertentes existentes de uma plataforma de marcação de tempo remunerável, nomeadamente a agregação das funcionalidades de marcação de horas de trabalho, gestão de trabalhadores e geração de relatórios.

1.2 OBJECTIVOS

Este projecto consiste na implementação de uma infra-estrutura orientada ao serviço, extensível e adaptável. A implementação tem como objectivo a contabilização de tempo *billable* (tempo de trabalho a ser pago) e a geração de relatórios. A plataforma possibilita a marcação de horas de trabalho, férias e faltas com categoria associada, inseridas pelo trabalhador. Os dados inseridos são validados, sendo automaticamente analisadas as marcações de faltas e de férias, através de regras configuráveis.

A plataforma é constituída por uma *Application Programming Interface* (i.e. *API*) de gestão de tempo para inserção de dados relacionados com o tempo de trabalho, por parte do colaborador, e gestão de clientes, por parte do gestor, de forma a gerar relatórios mensais com base no trabalho realizado. Uma *API* de contabilidade poderá ser futuramente incluída para viabilizar o uso da plataforma pelos serviços de contabilidade das empresas.

A partir dos dados inseridos podem ser obtidos, não só os dias trabalhados por colaborador, mas também a quantidade de trabalho investida por projecto e por cliente (em casos de empresas de consultoria). Por exemplo, é possível obter a quantidade de trabalho no mês dedicada a um projecto específico e a quantidade de trabalho investida por colaborador.

Pretende-se também disponibilizar na plataforma a funcionalidade de gerar alertas relacionados com horas de trabalho (e.g. indicação de existência de trabalhadores em sobrecarga de trabalho), permitindo uma redistribuição de trabalho mais eficaz.

As funcionalidades são expostas através de *web services,* podendo a plataforma ser utilizada em qualquer modelo de negócio (i.e. escritório, trabalho exterior). Assim, não existe nenhuma dependência da rede interna da empresa, sendo permanentemente acessível através de qualquer aparelho com ligação à *internet*. A disponibilização da plataforma através de *web services* viabiliza o acesso às suas funcionalidades a partir de outras aplicações.

O serviço deve ter uma vertente por empresa que é personalizável de acordo com o modelo de negócio. A solução é disponibilizada adoptando um modelo *SaaS* (i.e. Software as a Service), maximizando a disponibilidade da plataforma e permitindo o controlo de acessos.

1.3 SOLUÇÃO

A solução começa pela avaliação das características do modelo *SaaS* e pela escolha da plataforma *cloud* a usar, partindo-se posteriormente para a implementação de um projecto que tire partido da plataforma escolhida.

O projecto está dividido em duas partes, o serviço e o cliente. O serviço é o principal foco do projecto, disponibilizando um modelo *SaaS* sobre uma plataforma *cloud*, através de uma interface *REST*. A ênfase do projecto está na disponibilização de novas funcionalidades através de uma arquitectura que facilite a sua implementação.

O serviço implementa um mecanismo de autenticação e autorização de utilizadores, fazendo o processamento de pedidos através de uma camada de negócio. A camada de negócio, por sua vez, usa uma camada de acesso a dados de forma armazenar a informação de forma persistente. Existe ainda uma componente de tratamento de erros, que faz a tradução de excepções para as respostas *HTTP* adequadas. Através destes mecanismos estão

implementadas parte das funcionalidades dos objectivos descritos: edição de *timesheets*, para marcação do tempo *billable*, e gestão de utilizadores e recursos do sistema.

A implementação do cliente disponibiliza um protótipo funcional onde podem ser executadas as operações do serviço, através de uma aplicação *web* que realiza pedidos *AJAX*. A aplicação está estruturada por controlos, sendo cada recurso do sistema manipulado através de um controlo específico. Cada controlo está dividido em componentes de forma a separar as diferentes responsabilidades - validação e lógica local, comunicação com o serviço e aspecto gráfico. O cliente disponibilizado é apenas uma de várias possíveis implementações, servindo sobretudo de referência para a comunicação entre o cliente e o serviço.

Capítulo 2

PLATAFORMAS CLOUD

A solução do projecto é disponibilizada através de um modelo *SaaS*, um modelo de negócio assente através de um serviço acessível com ligação à internet, cuja aplicação não reside na máquina de um cliente mas na entidade vendedora. Ao contrário da vertente tradicional, em que as licenças são pagas por máquina, nesta vertente a aplicação é paga por mensalidades ou conforme as necessidades de acesso e consumo (e.g. por utilizador, por largura de banda, por quantidade de dados guardados).

As características deste modelo são conseguidas através da disponibilização de um ponto de acesso centralizado, através da *web*, em vez de localmente no cliente. O acesso e a gestão podem assim ser feitos remotamente e as actualizações à aplicação são centralizadas, não sendo necessárias actualizações no computador do cliente. Desta forma, este modelo facilita tanto a eliminação de erros como o lançamento de novas funcionalidades.

2.1 *CLOUD COMPUTING*

Cloud computing refere-se a um modelo de utilização da *internet* e de um conjunto de servidores remotos que fornecem serviços de hospedagem de aplicações e de armazenamento de dados. Este modelo é utilizado por empresas para hospedar as suas aplicações servidoras sem ser necessário o investimento na criação e/ou manutenção dos seus próprios servidores.

Devido ao foco específico das aplicações *web* hospedadas, que podem necessitar de suporte para o uso simultâneo de um número elevado de utilizadores, estes modelos fornecem também outros serviços e *APIs.* Destes, destacam-se suporte para a escalabilidade das aplicações, a capacidade de balanceamento de carga e outras funcionalidades comuns, como a identificação de utilizadores e controlo de acessos.

Quando comparada com uma aplicação hospedada *in premises* (i.e. a localização física do cliente), uma aplicação hospedada numa plataforma *cloud* apresenta vantagens: não existe a

necessidade de investimento inicial, devido ao serviço ser cobrado com base no uso, e os recursos necessários de administração de sistemas são inferiores aos de um *data-center* local. Da mesma forma, é fácil de reforçar a infra-estrutura pois é possível adicionar dinamicamente novas máquinas, pagas conforme a sua utilização. Estas características viabilizam a adição de novos servidores em períodos de intensa utilização e a existência de redundância, que quando geográfica, permite a recuperação rápida do serviço em caso de desastres naturais. O consumo é pago por utilização (i.e. recursos computacionais usados) ou subscrição (i.e. tempo de utilização) ou uma combinação dos dois.

Concluindo, a hospedagem de aplicações na *cloud* tem vantagens relativamente à hospedagem *in premises*: a ilusão da existência de recursos de computação infinitos (i.e. quantidade de máquinas disponíveis e de armazenamento de dados), inexistência da necessidade de um grande investimento inicial, sendo apenas necessário o pagamento dos recursos computacionais efectivamente usados.

O presente projecto pressupõe utilização diária por parte dos consultores e utilização variável (conforme necessário) pelos gestores, recursos humanos e outros departamentos a integrar. A sua execução numa plataforma *cloud* tem vantagens, devido ao facto dos recursos serem cobrados à hora e poderem ser aumentados quando necessário: se forem precisos recursos computacionais adicionais para serem executadas tarefas mais rapidamente, o aluguer de 1000 máquinas por 1 hora é igual ao pagamento do uso de uma máquina por 1000 horas (no caso do *Windows Azure* e dos *Amazon Web Services*).

A utilização de uma plataforma *cloud* permite a disponibilização de um *SaaS* com as vantagens apresentadas, existindo diversos serviços de *cloud* disponibilizados, com diferentes características. De forma a analisar a plataforma mais adequada para o projecto a desenvolver, foram estudadas as plataformas *cloud* consideradas mais importantes:

- *Google App Engine* (4)
- *Windows Azure* (5)
- *Amazon Web Services* (6)
- *Joyent* (7)
- *VmWare vFabric cloud application platform* (8)

Nas secções seguintes são apresentadas as conclusões retiradas da análise de cada uma das plataformas enumeradas. As características analisadas podem não corresponder à realidade das plataformas na entrega do presente relatório, devido a terem decorrido oito meses entre o momento da análise e a entrega final do projecto.

2.1.1 GOOGLE APP ENGINE

O *Google App Engine* (i.e. GAE) é o resultado da disponibilização da infra-estrutura usada internamente pelos serviços de procura e de *e-mail* da *Google*. O desenvolvimento das

aplicações da *Google* é focado nas suas funcionalidades em vez de nos problemas relacionados com as aplicações *Web* - como a disponibilidade da aplicação, conseguida pela sua capacidade de escalabilidade e por uma componente de balanceamento de carga - pelo que existe uma *framework* de abstracção que permite o programador focar-se só nas funcionalidades do que está a escrever. O *GAE* é o resultado da disponibilização desta *framework* para o público.

Nesta plataforma é possível executar *Python* (9)*, com* as suas bibliotecas *standard*, e *Java* (10), embora as máquinas virtuais limitem determinadas operações das aplicações em execução, como escrever em ficheiros ou abrir *sockets*. Os ambientes de execução permitem uma abstracção do sistema operativo e fazem a distribuição de processamento, encaminhamento de pedidos para o servidor mais adequado (com critério não divulgado), a escalabilidade e a distribuição de carga. As aplicações são carregadas quando é feito o primeiro pedido ao servidor, ficando carregadas para servir os pedidos seguintes, existindo no entanto, por cada pedido um tempo limite de processamento de 30 segundos.

O armazenamento de dados não pode ser feito localmente, mas podem ser carregados ficheiros de configuração com a aplicação. Para armazenamento existem dois serviços: o serviço de *blobstore* e o serviço *datastore*. O primeiro é indicado para armazenamento de dados multimédia e o último para armazenamento de dados no formato de pares chave-valor (facilitando o armazenamento de dados de objectos). As entidades armazenadas no *datastore* são *schemaless*, não havendo uma estrutura definida para um tipo de objecto, e armazenadas em grupos de dados (i.e. *entity groups*) sobre os quais é dado suporte transaccional. Para bases de dados relacionais é usado o serviço de *Hosted SQL*. Este serviço disponibiliza um sistema de gestão de base de dados (i.e. *SGDB*) que funciona sobre o serviço *datastore*, tendo suporte transaccional e constituição de vistas. Outras necessidades existentes nos *SGDBs* relacionais, como a configuração da base de dados e a manutenção (e.g. manutenção de índices), são geridas pela plataforma.

A plataforma fornece suporte para execução assíncrona de código através de filas: o serviço *task queue* permite inserir código que é eventualmente executado pela plataforma, processando as tarefas por uma ordem *FIFO*, e o serviço de *cron jobs* permite agendar a execução periódica de operações, identificadas por *URL*, como por exemplo a realização de operações de manutenção ou de notificação. O serviço de *task queue* permite delegar trabalho para operações que demorem mais de 30 segundos.

As aplicações hospedadas no *GAE* têm acesso às funcionalidades dos serviços *Google*, havendo *APIs* disponíveis para a sua utilização. Estas *APIs* permitem a identificação de utilizadores através do *Google Accounts*, o envio de *e-mails*, a manipulação imagens e acesso a outros recursos *online (e.g. URLFetch).* Existem outras *APIs* disponibilizadas com a finalidade de melhorar o desempenho da aplicação, destacando-se de entre estas o acesso ao

serviço de transmissão de pacotes de dados de grande dimensão, através do protocolo *Extensible Messaging and Presence Protocol* (*XMPP*), e o serviço de *MemCache*. O serviço de *MemCache* permite a utilização uma cache *RAM* distribuída, onde se podem guardar dados através de um modelo chave-valor.

As aplicações podem ser testadas localmente através da execução de um servidor local que simula o *GAE*, que vem incluído no *Software Development Kit* (*SDK*). O servidor local simula o serviço *datastore* e os outros serviços *online*. De forma a viabilizar o teste do comportamento da aplicação quando colocada *online*, é suportado o carregamento da mesma para uma zona de testes, não ficando disponível para o público (i.e. *staging*), permitindo que esta seja acedida sem ficar publicada. As aplicações colocadas online podem ser controladas através de uma página de gestão onde são monitorados vários aspectos, como a quantidade de acessos a determinado recurso, o tempo de processamento e o volume de dados armazenados no datastore. Desta forma, é possível vigiar as aplicações de forma a melhorar o seu desempenho.

2.1.2 WINDOWS AZURE

O *Windows Azure* é a plataforma de *cloud computing* da Microsoft para executar e armazenar dados de aplicações nos seus servidores.

As aplicações podem ser escritas em linguagens de programação utilizadas para escrever aplicações executáveis no sistema operativo Microsoft Windows, em *.NET* (como o *C# e Visual Basic)* ou *C++, Java* e *Ruby* (11) (12). De forma a facilitar o desenvolvimento, existe integração com o *IDE Visual Studio* e *SQL Server Management Studio* e com as tecnologias *ASP.NET*, *WCF* (para utilização de filas de mensagens) e *FastCGI* (em *PHP* ou *Python*). Cada aplicação é executada numa máquina virtual, ou instância, existindo quatro tipos diferentes de instâncias (i.e. *small, medium, large,* e *extra large*), que representam máquinas virtuais com diferentes capacidades. É também possível indicar a zona geográfica (i.e. Estados Unidos, Europa, Ásia) onde cada instância pode ser executada e onde os dados devem ser guardados. O pagamento é feito por instância, variando com a capacidade e a quantidade de dados armazenados e transferidos.

Na plataforma podem ser executados dois tipos de componentes: os *Web Roles* e os *Worker Roles*. Os componentes do tipo *Web Role* aceitam pedidos *HTTP* e *HTTPS* através do *Internet Information Services 7* (i.e. *IIS*) e têm integrado um sistema de balanceamento de carga que distribui os pedidos pelas instâncias iniciadas. Os componentes do tipo *Worker Role* são usados para executar as tarefas de suporte às aplicações do tipo *Web Role*. Cada *Role* pode ser executado numa ou mais instâncias, indicação dada através de ficheiro de configuração passível de ser ajustada em tempo de execução através da *API* disponibilizada para o efeito.

Nesta plataforma todas as máquinas são geridas pelo *fabric controller*, componente responsável pela gestão dos servidores e dos sistemas operativos e pela instalação das actualizações necessárias das versões do *Windows Server*, onde são executadas as máquinas virtuais. É nas máquinas virtuais que são executados os componentes. O *controller* vigia o estado das máquinas, iniciando novos componentes em caso de falha.

Também é possível monitorar a execução das aplicações através do sistema *Fabric*. O sistema *Fabric* permite obter informação acerca do estado das máquinas, para melhorar o desempenho de uma aplicação e ajudar na prevenção de casos de falha (e.g. permite iniciar novas máquinas quando as que estão em execução se encontram sobrecarregadas).

Para armazenamento existem dois componentes do serviço *storage*: o componente *Binary Large Objects*, usado para armazenar dados (e.g. dados multimédia), e o componente *Tables*, usado para armazenar informação estruturada, permitindo guardar objectos. Em semelhança ao *datastore* do GAE, os dados guardados no componente *Tables* não tem *schema* associado, podendo, no entanto, serem manipulados com *Linq* ou *ADO .NET*. O serviço *SQL Azure* disponibiliza uma base de dados relacional, assegurando fiabilidade, gestão de cópias de segurança e funções de administração automatizadas. Além disso, possui características semelhantes ao *SQL Server*, como o suporte de índices, *views*, *stored procedures* e *triggers*. Os dados de todos os serviços são automaticamente replicados com consistência forte (definição em (13)).

O suporte para execução assíncrona é conseguido através do componente *queue* para a utilização de filas de mensagens. Estas filas têm uma filosofia *FIFO* e as mensagens têm de ser removidas explicitamente, garantindo que foram processadas com sucesso. De forma a suportar coordenação de tarefas, uma mensagem quando lida deixa de ser visível na fila durante 30 segundos. Este componente é usado na comunicação entre *Web Roles* e *Worker Roles*.

O *Windows Azure Platform AppFabric* fornece um *bus* de serviços, permitindo comunicação entre aplicações fazendo o reencaminhamento dos pedidos feitos, e um mecanismo de controlo de acessos, que permite identificação de aplicações e estabelecer permissões de acesso.

O *SDK* fornecido inclui uma versão do ambiente *Windows Azure* que pode ser executado localmente, simulando o comportamento da aplicação na *cloud*. Uma aplicação pode ainda ser disponibilizada sem ser acessível publicamente (i.e. *staging*), podendo ser publicada posteriormente.

2.1.3 AMAZON WEB SERVICES

A *Amazon* disponibiliza a sua plataforma de c*loud computing* através dos *Amazon Web Services* (i.e. AWS). Nesta plataforma, cada funcionalidade é disponibilizada através de um serviço. Desta forma, qualquer aplicação, esteja ou não hospedada nos *Amazon Web Services*, tem acesso às funcionalidades existentes. Apesar de desacoplados, os serviços partilham a mesma lógica de nomes e de autenticação e estão estruturados para funcionarem em conjunto.

Cada serviço tem objectivos diferentes e pode ser executado em diferentes zonas geográficas, variando o pagamento conforme os serviços e a quantidade de instâncias usadas. Para execução de instâncias servidoras é usado o serviço *Amazon EC2,* onde o utilizador tem acesso total a uma máquina virtual. Nas máquinas, denominadas *Amazon Machine Instance* (i.e. *AMI*), o software tem de ser instalado pelo utilizador, desde o sistema operativo às aplicações, bibliotecas, dados e configurações necessárias. Em alternativa, podem-se usar ou modificar imagens pré-configuradas, que disponibilizam combinações de vários sistemas operativos e tecnologias.

O balanceamento de carga é disponibilizado pela componente *Elastic Load Balancer*, que distribui os pedidos por várias instâncias *EC2*. A componente *Auto Scaling* fornece escalabilidade às aplicações, ajustando a quantidade de instâncias executadas de acordo com a quantidade de processamento utilizada, utilizando informação configurada pelo utilizador.

Para armazenamento existem dois tipos de serviço: os serviços de armazenamento de dados em formato binário e os serviços de armazenamento estruturado de dados. Para armazenamento binário de dados existe o serviço designado *Amazon Simple Storage Service* (i.e. *S3*), que permite um acesso público ou privado aos dados, e o serviço *Amazon Elastic Block Storage* (i.e. *EBS*)*,* que se comporta como um dispositivo de armazenamento de ficheiros (pode conter qualquer sistema de ficheiros) e que é percepcionado pelo sistema operativo como um disco rígido. Em adição, existe suporte para criação de *snapshots* dos dados, sendo armazenados no serviço *S3*. O serviço *Amazon SimpleDB* é usado para armazenamento estruturado de dados, aceita dados no formato chave-valor e não tem *schema*. Os dados estão organizados por domínios e podem conter vários *items* (mapeando para objectos, o domínio representa o tipo e cada item uma instância*),* que por sua vez têm atributos chave-valor. Por fim, o *Amazon Relational Database Service* disponibiliza *SGBDs MySQL*, simplificando a sua utilização na *cloud*, sendo responsável pela iniciação, gestão, *backup* e escalabilidade de instâncias. Os dados de todos os serviços são automaticamente replicados numa zona definida.

Para comunicação assíncrona pode ser usado o serviço *Amazon Simple Queue Service*, um mecanismo de filas para a comunicação entre diferentes componentes/aplicações.

Estão também disponíveis vários serviços utilitários. O serviço *AWS Identity and Access Management* permite gestão de permissões e acesso granular a recursos e serviços. O serviço *Elastic MapReduce* permite processamento intensivo ou em larga escala, com paralelização de tarefas para, por exemplo, *data mining* ou aprendizagem automática. Por fim, o serviço *CloudFront* fornece uma rede de distribuição de dados para entrega de conteúdos. Este serviço complementa o serviço S3, servindo para copiar dados para a localização mais próxima (na rede) do utilizador da aplicação.

Cada instância pode ser monitorizada através da componente *Cloudwatch*. Esta componente fornece e regista informações sobre o desempenho de uma instância, relativamente ao processamento, a acessos de I/O e acessos à rede.

2.1.4 JOYENT

A plataforma *Joyent* foi analisada por disponibilizar uma arquitectura diferente das outras plataformas para as aplicações hospedadas e por ser usada por várias empresas conhecidas, como o *Linked-In* (14), a *THQ* (15) e a *KABAM* (16). Em semelhança aos *Amazon Web Services*, disponibiliza máquinas servidoras e componentes de balanceamento de carga e armazenamento de dados.

A diferença desta plataforma reside no conceito de *SmartMachines,* as máquinas servidoras da plataforma, que conseguem ter uma escalabilidade vertical (i.e. *burst*), ou seja, conseguem aumentar a capacidade de processamento e a quantidade de memória. Desta forma, em períodos com uma quantidade intensa de pedidos são usados mais recursos disponíveis na máquina física servidora, onde a capacidade de processamento e a quantidade de memória aumentam dinamicamente até ao ponto de não prejudicar outras aplicações hospedadas na mesma máquina. Estas máquinas são usadas tanto para processamento de pedidos como para armazenamento de dados. A arquitectura da plataforma é implementada pelo utilizador, apesar de estarem disponíveis várias soluções pré-definidas.

A plataforma não foi considerada para utilização pois apesar de existir documentação da *API* para o desenvolvimento de aplicações para a componente *Smartplataform (*orientada ao desenvolvimento de aplicações *web*), as outras componentes não tem documentação detalhada e é contraditória em algumas funcionalidades. Além disso, a informação sobre os serviços fornecidos é vaga e a arquitectura não fornece serviços com a mesma abstracção para o desenvolvimento de aplicações na *cloud* como as outras plataformas existentes.

2.1.5 VMWARE VFABRIC CLOUD APPLICATION PLATFORM

Esta é uma plataforma para hospedagem de aplicações na *cloud*, recentemente lançada pela *VMWare*. Inclui as ferramentas e a tecnologia *Spring* (17) e os serviços existentes em plataformas semelhantes: ambiente de execução para as aplicações servidoras, componentes

de comunicação entre aplicações e a possibilidade de montar uma infra-estrutura de paralelização de tarefas, para processamento intensivo ou em larga escala. No entanto, esta solução não fornece serviço para hospedagem de aplicações em servidores remotos. Em vez disso, oferece um pacote de *software* da *VMWare* para instalação em servidores *in premises,* de forma a ter ou fornecer serviços *cloud* a terceiros.

2.1.6 CONCLUSÕES

As plataformas da *Joyent* e da *VMWare* não foram consideradas para utilização, pois não são adequadas para o projecto a implementar. Para uma melhor análise da plataforma a usar, foram comparadas as características das outras plataformas - *Google App Engine*, *Windows Azure* e o *Amazon Web Services* - cujos resultados são apresentados no *Anexo A – Comparação de Plataformas Cloud*.

A plataforma da *Amazon* disponibiliza uma máquina virtual em vez de um ambiente de execução para aplicações, não tendo um desenvolvimento e manutenção tão facilitados como as plataformas *GAE* e *Windows Azure*. O *Google App Engine* é a plataforma que fornece maior abstracção, mas condiciona a linguagem e as tecnologias que podem ser utilizadas. As plataformas fornecem características semelhantes (i.e. armazenamento, autenticação e comunicação), sendo distintas principalmente nas *APIs* disponibilizadas e na facilidade de desenvolvimento. O *Google App Engine* tem a infra-estrutura mais completa para o desenvolvimento de aplicações enquanto a plataforma *Amazon Web Services*, por outro lado, tem as melhores características para aplicações que precisem de *cluster*.

Os custos de uso das plataformas são difíceis de calcular devido à quantidade de serviços usados variar com a aplicação e porque o preço depende da quantidade de armazenamento utilizado e da transferência de dados envolvidos. De qualquer forma, o *Amazon EC2* e o *Windows Azure* são plataformas mais caras, pois precisam de ter sempre uma instância a ser executada. No *Google App Engine* só se pagam as horas de computação utilizadas, sendo a plataforma mais barata. Além disso, tem uma fasquia de uso gratuito, a que se tem sempre direito, esteja em modo gratuito ou de cobrança. Mesmo depois de ultrapassada a fasquia gratuita, tem um tempo de computação mais barato que as outras plataformas. De forma a diminuir custos de desenvolvimento, o *Amazon Web Services* e o *Windows Azure* disponibilizam pacotes pré-pagos com horas disponíveis para ter a aplicação *online*.

Tendo em conta a relação entre características disponíveis das plataformas e os custos, a plataforma *GAE* foi a plataforma escolhida. A escolha da *GAE* como plataforma de hospedagem tem também como vantagem a opção de programação em *Java*, que facilita a migração da aplicação, em caso de problemas, para a plataforma *Windows Azure* ou para o *Amazon EC2*, pois ambas as plataformas também suportam a linguagem de programação.

Capítulo 3

LINHA DE PROJECTO

3.1 METODOLOGIA DE DESENVOLVIMENTO

A metodologia de desenvolvimento utilizada consistiu na utilização de uma abordagem semelhante a *scrum* (18). Foi feito o levantamento dos requisitos de sistema que foram depois expressados em funcionalidades, apresentadas no Capítulo 3.2, a serem implementadas por ordem de prioridade. Numa fase inicial, a abordagem para implementar cada funcionalidade foi a seguinte:

- Levantamento da arquitectura;
- Requisitos implementados;
- Aprovação em todos os testes unitários[1];
- Funcionalidade testada no protótipo e no *Google App Engine;*
- Diagramas actualizados (*UML*)

A implementação por funcionalidade foi mais tarde abandonada, sendo a metodologia descrita usada para implementar várias funcionalidades em simultâneo. Esta decisão promoveu a diminuição das restruturações da estrutura de dados e da arquitectura das *DAO* (que também mudaram devido a problemas com as *API* de dados). O uso desta abordagem aumentou a produtividade do projecto, aumentando o planeamento e diminuindo a mudança de contexto nas diferentes áreas do projecto.

De forma a evitar a perda do código escrito é usado um repositório com controlo de versões GIT (19). Este repositório está situado no *Unfuddle* (20), um serviço gratuito de hospedagem destes repositórios.

[1] Os testes unitários foram escritos com a ferramenta TestNG (52)

Os testes unitários permitiram aumentar a qualidade do código do projecto. Os testes analisam as funcionalidades disponibilizadas na lógica de negócio, sendo mostrados na Figura 3.1.

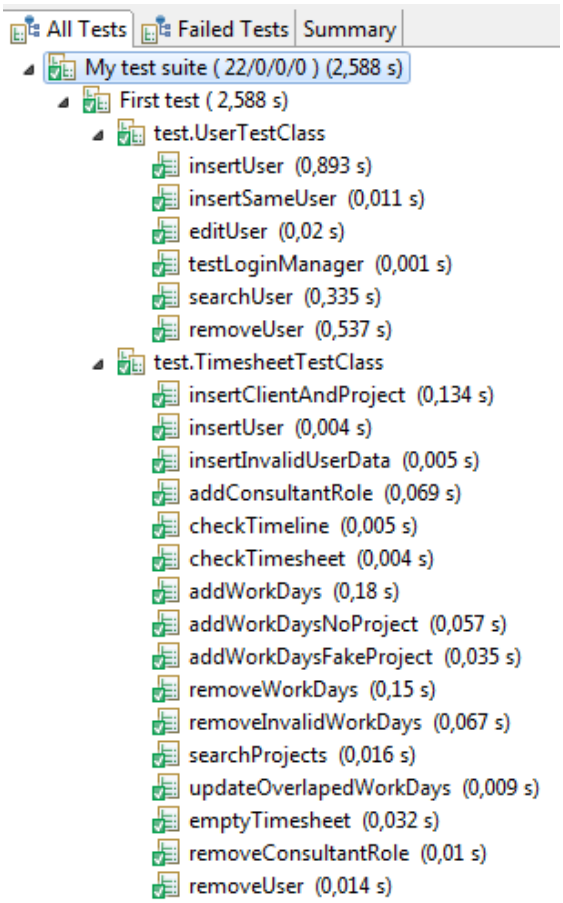

FIGURA 3.1- RESULTADO DOS TESTES UNITÁRIOS EXECUTADOS

Os testes asseguram que as novas funcionalidades não interferem no funcionamento das funcionalidades já implementadas. Apesar de não serem extensivos, testam os principais cenários de funcionamento e o comportamento em situações de erro.

Nos testes unitários são feitas chamadas a operações da lógica de negócio, sendo assim também verificado o funcionamento da lógica da camada de dados. Como trabalho futuro, podem ser acrescentados testes que verifiquem situações na camada de dados que não sejam possíveis testar através da camada de negócio. Não é possível automatizar situações de concorrência pois a sincronização é feita através de transacções, sendo necessário modificar o código da camada de dados para se poderem testar esses cenários.

Os pedidos feitos ao serviço e o formato de dados utilizados são testados pela aplicação cliente. Numa fase final, foi implementada uma versão cliente mais elaborada, aproveitando as funcionalidades já implementadas do protótipo. Esta versão, apesar de ter um funcionamento complexo, não é testada por teste unitários, estando a implementação destes testes identificada como trabalho futuro.

3.2 FUNCIONALIDADES

O projecto teve uma fase de levantamento de requisitos onde se recorreu à elaboração de casos de utilização com o detalhe necessário à validação de utilizadores habituais de soluções semelhantes. Depois de validados os casos de utilização, foram definidas as respectivas prioridades de implementação, resultando daí a identificação das principais funcionalidades a implementar. As funcionalidades são indicadas de seguida, de acordo com a sua prioridade de implementação:

1. *Timesheets*
    - Marcação de horas de trabalho e faltas
    - Marcação e remoção de férias
    - Fecho mensal de uma *timesheet*
    - Visualização intermédia da *timesheet*
    - Edição da *timesheet* após fecho
2. Mecanismo interno de validação de acções
    - Validação das acções definidas através *workflow*
    - Criação de *workflows* e modificação de *workflows* existentes
3. Gestão do sistema
    - Gestão de papéis do sistema
    - Projectos – criação, associação (de actores) e remoção de consultores
    - Fechar projectos
    - Edição do horário de um consultor
    - Visualização dos consultores
    - Transferência de gestor
    - Definição de período anual de trabalho
4. Regras (Opcional)
    - Definição, edição e remoção de regras de faltas
    - Definição, edição e remoção de regras para marcação de férias
5. Avisos (Opcional)
    - Marcação e geração de avisos
6. Relatórios (Opcional)
    - Relatórios de visão de gestor
    - Relatórios usados para contabilidade e cliente

No presente projecto foi implementado um subconjunto destas funcionalidades. A decisão de implementar apenas um subconjunto das funcionalidades inicialmente previstas deve-se a três factores: inadequação da biblioteca usada para acesso aos dados; complexidade imprevista de algumas das funcionalidades; e uma implementação do cliente mais elaborada do que inicialmente estipulado.

De forma a melhorar a produtividade optou-se por planear a arquitectura de uma forma geral, não contemplando só a funcionalidade a implementar, permitindo ter uma visão mais abrangente do projecto. A prioridade de implementação das funcionalidades mudou ao longo do projecto, de forma a ter um produto final com a sua conclusão. A implementação das restantes funcionalidades constitui trabalho futuro. As funcionalidades concluídas são as seguintes:

- *Timesheets*
  - Marcação de horas de trabalho e faltas
  - Marcação e remoção de férias
  - Visualização intermédia da *timesheet*
- Gestão do sistema
  - Gestão de papéis do sistema
  - Projectos – criação, associação e remoção de consultores
  - Edição do horário de um consultor (parcialmente implementada)
  - Visualização de utilizadores

A implementação destas funcionalidades é descrita nos próximos capítulos. Apesar da ênfase do projecto ser na criação do serviço onde estas funcionalidades são disponibilizadas, existe um capítulo dedicado à implementação cliente (i.e. Capítulo 5). Esta separação deve-se ao facto da aplicação cliente ser mais complexa do que o inicialmente planeado, e viabiliza a descrição detalhada da aplicação cliente e respectiva arquitectura.

Capítulo 4

IMPLEMENTAÇÃO DO SERVIÇO

4.1 ARQUITECTURA GERAL

O projecto tem como objectivo a disponibilização de uma infra-estrutura que dê suporte à implementação dos requisitos funcionais. A arquitectura é composta por uma componente que valida a autorização dos pedidos feitos ao servidor, uma componente que faz a transformação dos erros que possam ocorrer para uma resposta adequada e uma camada onde é implementada a lógica de negócio, independente da plataforma onde está alojada. A lógica de acesso a dados é também implementada numa camada própria, sendo utilizada pela camada de negócio. A existência da camada de acesso a dados permite a transição para outra plataforma *cloud* que pode, eventualmente, ser necessária por motivos financeiros ou devido à plataforma *cloud* escolhida não ter a disponibilidade desejada (21) (22) (23).

O sistema implementado é mostrado na Figura 4.1. É constituído por três camadas independentes: a camada com a plataforma do serviço (i.e. componentes azuis), a camada que implementa a lógica de negócio (i.e. componentes vermelhos) e a camada que faz a gestão do acesso a dados (i.e. componentes verdes).

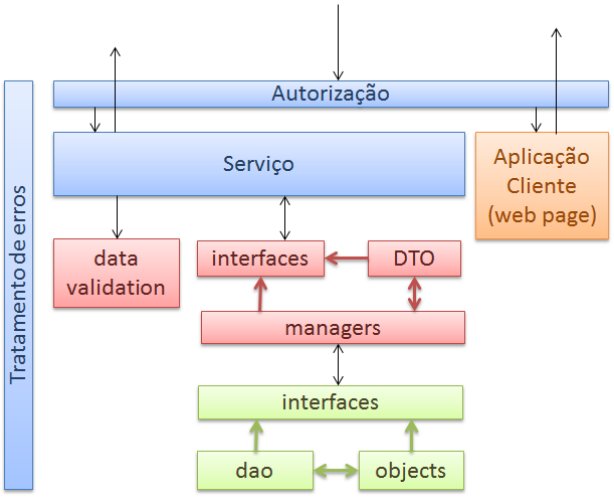

FIGURA 4.1 - ARQUITECTURA DO SERVIÇO

A plataforma do serviço faz o processamento de pedidos recebidos. Inclui os mecanismos de autorização, faz a transformação dos dados enviados e recebidos para o formato requisitado (*JSON* / *XML*), e verifica o formato dos dados recebidos. Nesta camada é também feito o tratamento de erros, traduzindo as excepções para o formato adequado (i.e. página erro em *HTML*, para pedidos para à aplicação cliente, ou resposta *HTTP* com o código apropriado, para pedidos feitos ao serviço). O processamento dos dados e o processo de validação dos dados é feito através da camada de negócio.

A camada de negócio é constituída pelas implementações dos objectos de negócio (i.e. *managers*) e os objectos que produzem (i.e. *data transfer objects* ou *DTO*). Nesta camada está implementada a lógica de negócio, sendo as suas operações disponibilizadas através de interfaces. Aqui são utilizadas operações da camada de acesso a dados para armazenar informação de forma persistente.

A camada de acesso a dados permite a manipulação de dados através de operações disponibilizadas pela interface dos *data access objects* (i.e. *DAO*). É aqui que está implementada a lógica de acesso ao *datastore*. Os dados do *datastore* são mapeados para objectos e são disponibilizados através de uma interface.

4.2 INTERFACE DO SERVIÇO

O serviço implementa uma interface *REST* (24) (25). A escolha desta interface foi tomada após uma análise das características dos dois tipos de arquitectura actualmente usados, *SOAP* e *REST*.

Um serviço com uma interface *SOAP* expõe directamente as operações de negócio e a comunicação pode ser feita através vários protocolos de comunicação (e.g. *HTTP, TCP*),

sendo as operações identificadas através de *XML* (i.e. *SOAP Headers*). Por este motivo, uma estrutura arquitectural *SOAP* não permite tirar partido do protocolo usado para a transmissão de dados. As operações são descritas através de um ficheiro *XML* designado *WSDL*.

Um serviço com uma estrutura arquitectural *REST* é implementado directamente sobre o protocolo *HTTP*. A combinação do verbo *HTTP* com o *URI* designa, de uma forma objectiva, a acção a ser executada (e.g. ***GET /timesheet/2011/06*** designa a acção de obter a *timesheet* de Junho de 2011), sendo expostas operações *CRUD* de forma transparente. O resultado da operação é indicado através dos códigos *HTTP* e o modo como a informação é transmitida não está limitado apenas a um formato, podendo ser disponibilizado em *XML* ou *JSON*, entre outros. Assim, o desenvolvimento de aplicações cliente pode feito tanto através da utilização de *AJAX* como de *Javascript*. Em semelhança aos serviços *SOAP*, pode ser disponibilizado um ficheiro *WADL* (*Web Application Description Language*) que descreve as operações disponibilizadas. O formato explícito de manipulação de dados e a maior simplicidade de implementação de aplicações cliente tornam a interface *REST* a escolha mais adequada para o serviço implementado.

4.3 AMBIENTE DE EXECUÇÃO

A plataforma é executada no *Google App Engine* em ambiente de execução Java. Esta opção permite o uso de uma máquina virtual *Java* 6, com um servidor que executa *Java Servlets,* e suporte para os serviços do *Google App Engine*. A tecnologia *Java Servlets* permite o processamento de pedidos *HTTP*, sendo a sua estrutura mostrada na Figura 4.2.

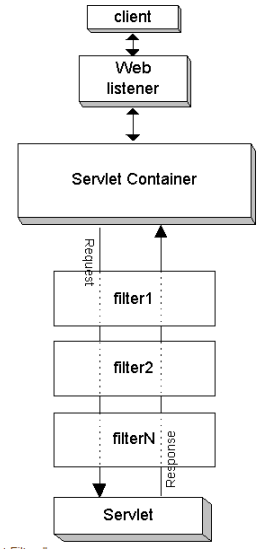

FIGURA 4.2 - PROCESSAMENTO DE UM SERVLET (26)

Os *Servlets* são classes *Java* que processam pedidos *HTTP*, colocados num *Servlet Container*. Os objectos do *ServletContainer* são instanciados uma vez por pedido. Cada *Servlet Container*

pode conter um ou mais *Servlets*. Podem também ser utilizados filtros, objectos que permitem executar código antes e depois de um pedido ser processado pelos objectos *Servlet,* de forma a adicionar suporte ao processamento dos pedidos. Existem também objectos do tipo *Listener*, instanciados uma vez por aplicação, que permitem fornecer recursos aos objectos do *ServletContainer*.

As máquinas virtuais disponibilizadas têm um conjunto de limitações que impossibilitam a execução de determinadas instruções que possam modificar o sistema de ficheiros ou o comportamento da máquina virtual. Posto isto, nem todas as tecnologias *Java* podem ser utilizadas (27) (28).

Para a implementação de um serviço com uma estrutura arquitectural *REST* optou-se, de entre as tecnologias *Servlet* disponíveis, pela utilização da biblioteca *Jersey*, a implementação de referência da *Oracle* do *standard JAX-RS* [2]. O *standard JAX-RS* permite definir recursos de forma simples e uma mudança entre bibliotecas que o implementem. O funcionamento da tecnologia Jersey é explicado em maior detalhe no *Anexo B – Utilização da tecnologia Jersey*.

4.4 AUTORIZAÇÃO

A tecnologia *Java Servlets* não tem um mecanismo de autorização nativo, sendo implementado um através de um filtro que impede que o pedido seja processado pelas classes *Servlet*. No projecto é necessário um mecanismo para estabelecer se um utilizador está autenticado e que permita atribuir uma autorização para cada pedido conforme a função do utilizador (i.e. *consultor*, *gestor, recursos humanos*).

4.4.1 MECANISMO AUTENTICAÇÃO

A autenticação é feita através da *API* integrada que usa o serviço *Single Sign-On do Google Accounts* (29). A escolha recaiu sobre esta API devido à simplicidade de utilização, não sendo um objectivo do projecto a utilização de um mecanismo de autenticação específico.

O processo de autenticação recorre à *API* para gerar um *URL* com a localização onde o utilizador se deve autenticar. Quando o utilizador se encontra autenticado, os pedidos feitos ao serviço passam a fornecer uma *cookie* que é reconhecida pela *API* de autenticação.

A *API* de autenticação tem um mecanismo de autorização que disponibiliza um sistema de *roles* (i.e. funções associadas ao utilizador) para definir a granularidade de acesso a recursos. No entanto, não tem o comportamento que se pretende, pois redirecciona um utilizador não

[2] *JAX-RS* ou *The Java API for RESTful Web Services* é uma interface usada para implementações de *API* para serviços *RESTtFul* através de anotações, também conhecida como *JSR311*.

autenticado para a página de autenticação. Além disso, só estão disponíveis dois tipos de função pré-definidas, as funções *User* e *Administrator*.

4.4.2 PLATAFORMA DE AUTORIZAÇÃO

De forma a atingir os objectivos estabelecidos para o mecanismo de autorização é utilizada a biblioteca *Spring-Security* (30), que usa tecnologias *Spring* (31) (32). Esta plataforma, compatível com o *Google App Engine*, permite indicar as funções necessárias que um utilizador tem de ter associadas para fazer um pedido a um recurso específico. A plataforma é disponibilizada através de um filtro colocado *Servlet Container*. A sua configuração é mostrada na Listagem 4.1.

```
1  <filter>
2      <filter-name>springSecurityFilterChain</filter-name>
3      <filter-class>
4          org.springframework.web.filter.DelegatingFilterProxy
5      </filter-class>
6  </filter>
7  <filter-mapping>
8      <filter-name>springSecurityFilterChain</filter-name>
9      <url-pattern>/*</url-pattern>
10 </filter-mapping>
```

LISTAGEM 4.1 - CONFIGURAÇÃO DE SPRING-SECURITY

A configuração define que o filtro faz o processamento de qualquer pedido feito ao serviço (i.e. *root, "/*",* na linha 9). Este filtro requer o carregamento do contexto que fornece os objectos necessários à autorização, carregados através do *event listener ContextLoaderListener* (pode ser visualizado na *Figura 4.2 - Processamento de um Servlet*). O objecto *listener* faz o carregamento do mecanismo de inversão de controlo usado pelas tecnologias *Spring,* o objecto *ApplicationContext*, instanciado uma vez por aplicação. No seu parâmetro de iniciação é indicada a localização do ficheiro com a descrição dos objectos que precisam de ser carregados para serem usados pelo filtro de autorização e a informação com as restrições sobre os recursos. A configuração usada é mostrada na Listagem 4.2.

```
1  <beans:beans ...>
2      <http use-expressions="true" entry-point-ref="gaeEntryPoint">
3          <intercept-url pattern="/ws/timesheets/**"
4                         access="hasRole('CONSULTANT')" />
5          (...)
6          <custom-filter position="PRE_AUTH_FILTER" ref="gaeFilter" />
7      </http>
8
9      <beans:bean id="gaeEntryPoint"
10                 class="weblayer.security.GAEAuthenticationEntryPoint" />
11
12     <beans:bean id="gaeFilter"
13                 class="weblayer.security.GAEAuthenticationFilter">
14         <beans:property name="authenticationManager"
15                         ref="authenticationManager"/>
16     </beans:bean>
17
18     <beans:bean id="gaeAuthenticationProvider"
19                 class="weblayer.security.GAEAuthenticationProvider"/>
20
21     <authentication-manager alias="authenticationManager">
22         <authentication-provider ref="gaeAuthenticationProvider"/>
23     </authentication-manager>
24 </beans:beans>
```

LISTAGEM 4.2 - ESTRUTURA DO MECANISMO DE AUTORIZAÇÃO

As autorizações estão definidas no ficheiro de configuração, como se pode verificar na linha 3 da Listagem 4.2. Um elemento *intercept-url* indica uma restrição sobre um *URL* (neste caso está aplicado sobre o *entrypoint* do serviço, começado por *ws/*) e a permissão necessária para aceder ao recurso.

O sistema de autorização utiliza dois objectos, identificados por *gaeEntryPoint* (i.e. linha 2), do tipo *GAEAuthenticationEntryPoint*, e por *gaeFilter* (i.e. linha 6), do tipo *GAEAuthenticationFilter*. O objecto identificado por *gaeFilter* executa o processo de autenticação do utilizador para cada pedido e o objecto identificado por *gaeEntryPoint* faz o processamento dos pedidos não autorizados, enviando um código de erro de *Not Authorized* (i.e. *HTTP* 404).

4.4.3 IMPLEMENTAÇÃO

A implementação da classe *GAEAuthenticationFilter* é utilizada por cada pedido feito ao serviço e tem como objectivo autenticar o utilizador que fez o pedido, na plataforma *Spring Security*, para que lhe seja atribuída uma autorização. Como a autenticação é feita através do *Google Accounts*, o utilizador pode estar autenticado mas não ser reconhecido pelo *Spring Security*, estando esse processo implementado na classe *GAEAuthenticationFilter*. A Figura 4.3 apresenta o algoritmo de autenticação usado.

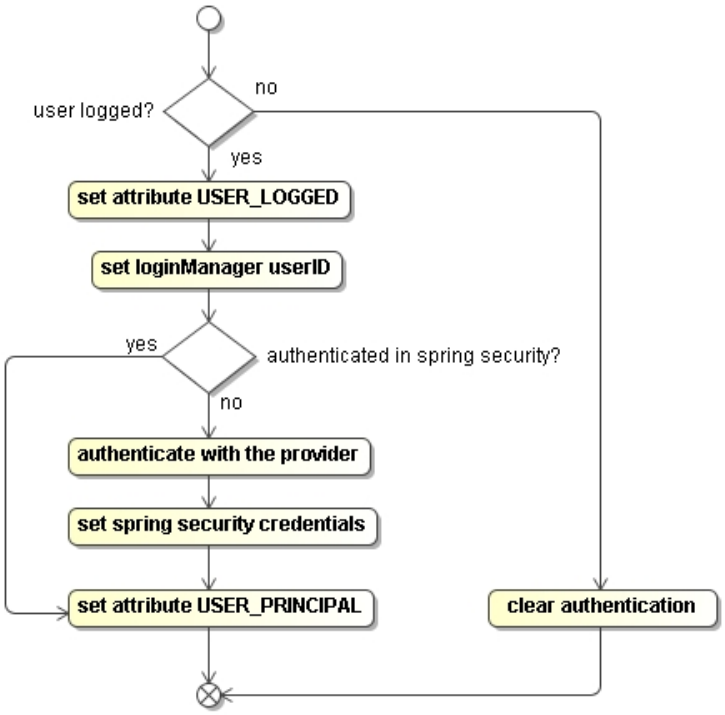

FIGURA 4.3 - PROCESSO DE AUTENTICAÇÃO IMPLEMENTADO

Na implementação do algoritmo começa por ser feita a verificação se o utilizador está autenticado através do serviço *Google Accounts*. Quando o utilizador não está autenticado, as credenciais de autenticação são removidas, que podem estar guardadas em sessão de um pedido anterior. Se o utilizador estiver autenticado, é identificado na camada de negócio, através do objecto *LoginManager*, e o seu identificador é guardado na informação do pedido com o atributo *USER_LOGGED*, ficando disponível para outros controlos (e.g. controlos do cliente). Quando o utilizador não é reconhecido pela plataforma *Spring Security*, é utilizado o objecto *AuthenticationManager* associado (i.e classe *GAEAuthenticationProvider,* definida na linha 12 da Listagem 4.2) para obter as credenciais do utilizador.

A classe *GAEAuthenticartionProvider* recorre à camada de negócio para verificar a identidade do utilizador. Se o utilizador estiver autorizado na lógica de negócio é fornecido um objecto do tipo *GAEUserAuthentication,* que fornece informação sobre o utilizador e as suas funções. As funções do utilizador são disponibilizadas através de instâncias de *TcUserRoleGrantedAuthority*, que implementam *GrantedAuthority*, sendo reconhecidas pela plataforma *Spring Security*.

Caso o pedido tenha sido autorizado, as credenciais são inseridas no contexto de segurança do *Spring Security*, ficando guardado em sessão de forma a evitar a repetição do processo de autenticação em pedidos futuros. Por este motivo, o utilizador tem de sair e voltar a iniciar sessão quando as suas funções (i.e. *roles*) mudam. Os dados do utilizador são disponibilizados na informação do pedido, sendo acessíveis através do atributo *USER_PRINCIPAL*.

O processo de autenticação pode gerar dois tipos de excepção: *UserAuthenticationNotEnabledException*, quando o utilizador não é reconhecido no sistema (por não estar registado ou activado) ou *AuthenticationServiceException*, quando ocorre um erro de acesso aos dados do utilizador. Quando o utilizador não está registado no sistema o processo continua como se o utilizador não estivesse autenticado, podendo verificar a autorização no sistema (seja através do serviço ou da aplicação cliente). Quando ocorre um erro de acesso a dados do utilizador, é enviado um erro de autorização em resposta ao pedido.

4.5 FORMATO DOS DADOS

A lógica de negócio permite que utilizadores (i.e. *users*) com o papel de consultor editem folhas de trabalho (i.e. *timesheets*). Na Figura 4.4 é mostrada a estrutura de dados, de forma a se perceber melhor a lógica de negócio.

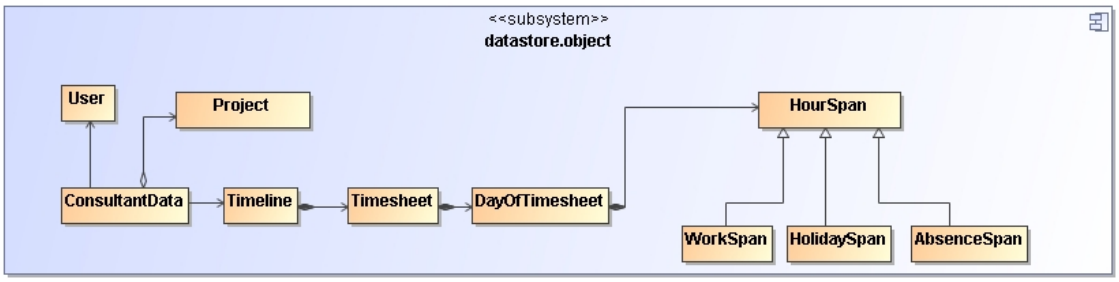

FIGURA 4.4 - ESTRUTURA DE DADOS DO SISTEMA

O sistema tem suporte para utilizadores que podem ter várias funções (i.e. *role*): consultor, gestor e recursos humanos. No projecto estão implementadas operações associadas à função de consultor e de recursos humanos. Estas funções estão directamente associadas aos dados do utilizador que, por sua vez, dão permissões e podem disponibilizar funcionalidades: a associação da função de consultor ao utilizador cria a estrutura de dados *ConsultantData*. Como nem todos os utilizadores têm esta funcionalidade associada, é a estrutura *ConsultantData* quem tem uma referência para o utilizador. Esta estrutura tem a indicação se está activa, pois a função consultor pode ser posteriormente removida, os projectos associados ao utilizador e uma *timeline*, com um conjunto de *timesheets*. Cada *timesheet* está organizada por dias e cada dia tem blocos de horas (i.e *spans*), que podem ser de três tipos: horas de trabalho, ausências (que podem ser justificadas ou injustificadas) e férias.

4.6 ORGANIZAÇÃO DO SERVIÇO

O serviço define uma interface *REST*, implementada através da biblioteca Jersey, disponibilizando *endpoints* orientados ao recurso e respondendo com os códigos de erro adequados.

As operações disponibilizadas através dos recursos têm uma resposta adequada à acção executada. Os pedidos do tipo *GET* permitem obter recursos, tendo uma resposta *OK* (i.e. código 200); pedidos do tipo *POST* são usados para inserir novos recursos no sistema, sendo devolvida uma resposta *CREATED* (i.e. código *201*) e a localização do novo recurso; pedidos do tipo *PUT* são usados para actualizar recursos já existentes e obtêm uma resposta de *NO CONTENT* (i.e. código 204); por fim, um pedido do tipo *DELETE* permite remover um recurso e o seu processamento devolve uma resposta *OK*. Em caso de erro, é devolvida a resposta adequada. Apesar de os outros verbos *HTTP* não serem usados, existem operações disponíveis que só devolvem a representação do objecto, o equivalente ao uso do verbo *HEAD*.

Estão disponibilizados 5 tipos de recursos: *authentication*, *users*, *projects*, *clients* e *timesheets*. Os recursos do sistema são *users*, *projects* e *clients*, sendo disponibilizadas operações para inserir, actualizar, remover, procurar e listar os recursos existentes. O processo de autenticação é feito através do recurso *authentication*. A autenticação do utilizador é feita através de um serviço externo (i.e. *third party login*), sendo a localização do sítio onde o utilizador se pode autenticar obtida através deste recurso. O recurso permite verificar se o utilizador depois de estar autenticado é reconhecido pelo sistema, através da operação */checkstatus*. O recurso *timesheet* permite obter e editar *timesheets*. De forma a optimizar o processo de actualização, é apenas indicada a nova informação e a informação a remover, em vez de ser recebida toda a informação da *timesheet*.

4.6.1 VALIDAÇÃO DE DADOS

Na utilização das operações da camada de negócio existe como compromisso a utilização de objectos validados, com a formatação certa (e.g. caracteres válidos no nome de um utilizador) e os campos adequadamente iniciados. Por este motivo, os dados são validados no processamento dos pedidos, no momento em que são recebidos. A validação de informação simples, como a verificação do valor de uma data, é validada localmente, enquanto que a informação mais comum ou complexa, deseriada para objectos no processamento do pedido (i.e. *data transfer objectos* ou *DTO*), é validada através da classe *Validator*.

A classe *Validator* disponibiliza um conjunto de métodos para validar formatos de dados comuns, como um *query string* ou o identificador do utilizador, que são validados através de expressões regulares. Os *DTO* são validados por uma operação cuja implementação utiliza uma classe de validação para cada tipo de *DTO* que possa ser recebido. As classes de validação implementam a interface *IDataValidator* e disponibilizam uma operação que uniformiza a informação do objecto e indica se os dados são válidos, de forma a ser indicado o motivo pelo qual o processo de validação falhou. Se um *DTO* tiver uma referência para outro objecto é utilizada a classe *Validator*, que utiliza a classe adequada para a sua validação.

A validação através da classe *Validator* em vez de uma validação local no *DTO* permite centralizar o processo, só sendo necessária a implementação da lógica de validação para os dados recebidos, em vez de para todos os *DTO*. Além disso, a classe *Validator* serve de ponto central de processamento de erros de validação.

4.7 LÓGICA DE NEGÓCIO

A interacção com a camada de negócio é feita através de operações disponibilizadas pelas interfaces dos objectos *Manager*. A sua organização pode ser consultada na Figura 4.5.

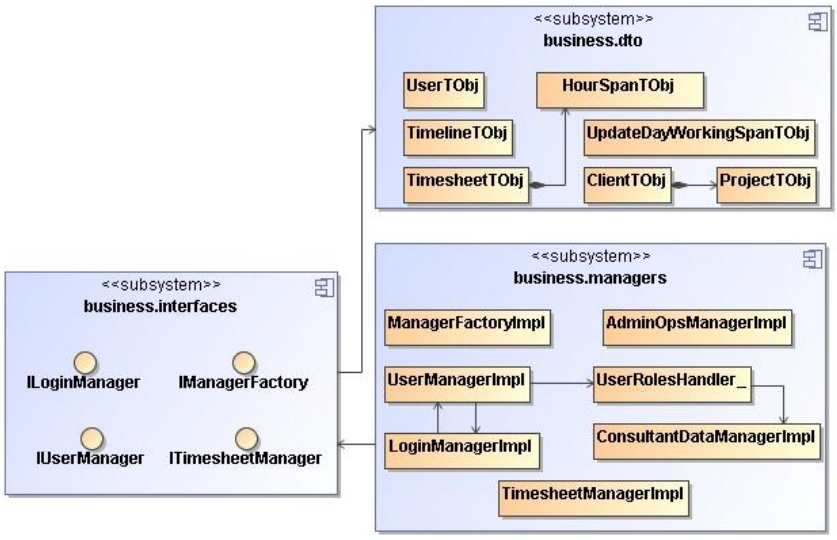

FIGURA 4.5 - CAMADA DE NEGÓCIO

A camada de negócio está dividida em três partes: interfaces dos objectos de negócio (i.e. *managers*, localizados em *business.interfaces*), as suas respectivas implementações (i.e. *business.managers*) e os objectos que produzem (i.e. *business.dto*).

Os objectos de negócio (i.e. *managers)* estão organizados por recursos. Existem, por exemplo, *managers* para manipulação de utilizadores, para gerir as *timesheets* e para gerir operações de administração. Uma acção iniciada num *manager* que manipule dois recursos é executada em dois *managers*. Quando, por exemplo, um *role* é associado a um utilizador, um *manager* actualiza o estado do utilizador e outro faz a actualização/criação de informação que permite a execução de novas acções. A associação de um *role* é gerida pela classe *UserRolesHandler*, onde são indicadas as acções a executar quando um *role* é associado ou removido a um utilizador e quando o próprio utilizador é apagado do sistema (podendo ser necessário apagar permanentemente os seus dados). Quando um *role* é associado a um utilizador o objecto *UserManager* adiciona o *role* e notifica o objecto *UserRolesHandler,* que executa as operações sobre o *manager* dos restantes dados a modificar.

Existe um objecto *manager, ILoginManager,* que não manipula um recurso. Este objecto tem dois objectivos: inserir utilizadores quando se autenticam, descritos no ficheiro de configuração, e manter o registo do utilizador que está autorizado no pedido (esta camada é independente da camada da plataforma), necessário em operações com resultado dependente do utilizador que fez o pedido. Apesar de manter a informação do utilizador que está associado ao pedido, é usada uma abordagem *lazy* (i.e. carregamento tardio*)*, só sendo lido o objecto da camada de dados quando este é explicitamente pedido. As inserções e leituras de utilizadores são sempre feitas através do respectivo objecto *manager*, *IUserManager*.

As operações dos objectos de negócio são mostradas em detalhe na Figura 4.6.

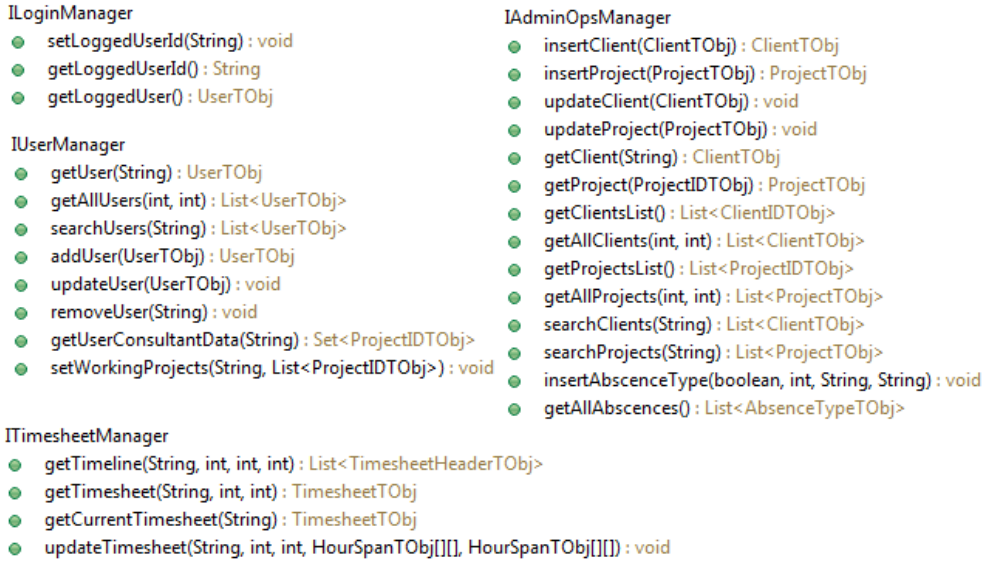

FIGURA 4.6 - INTERFACE DOS OBJECTOS DE NEGÓCIO

Cada operação é invocada no processamento de um pedido feito ao serviço. Os objectos *manager* têm a responsabilidade de implementar a lógica de negócio, validando a informação recebida com a informação armazenada. Os objectos resultantes das operações contêm anotações *JAXB*, que permitem uma seriação transparente de *JSON* e *XML*. Existem ainda operações que listam os recursos disponíveis, disponibilizando só a sua informação relevante como o identificador em detrimento de toda a informação associada, servindo para fornecer informação para outras operações (e.g. listagem de todos os projectos disponíveis para poderem ser associados a um consultor).

Existem objectos *manager* que não possuem interface pública, como o objecto *ConsultantDataManagerImpl*, e que são usados em operações de outros *managers*. Estes objectos permitem manter o processamento de cada recurso por *manager*, apesar de não serem manipulados directamente por um utilizador do serviço.

4.7.1 FUNCIONAMENTO DAS TIMESHEETS

O sistema está estruturado para permitir o acesso a qualquer *timesheet*. De forma a não desperdiçar recursos, as *timesheets* só são criadas quando é feito um pedido de actualização onde são inseridos dados. Quando é pedida uma *timesheet* que não existe no sistema, é devolvida uma *timesheet* vazia, que não tem representação na camada de dados.

Os dados das *timesheets* podem ser criados, actualizados e apagados na operação de actualização de uma *timesheet*. Além da identificação da *timesheet,* são recebidas duas listas: uma com a informação de novos *spans* e outra com a informação dos *spans* removidos. Cada posição da lista (i.e. *array*) representa um dia do mês, tendo cada dia os respectivos *spans*. A Figura 4.7 representa a operação de actualização de uma *timesheet*.

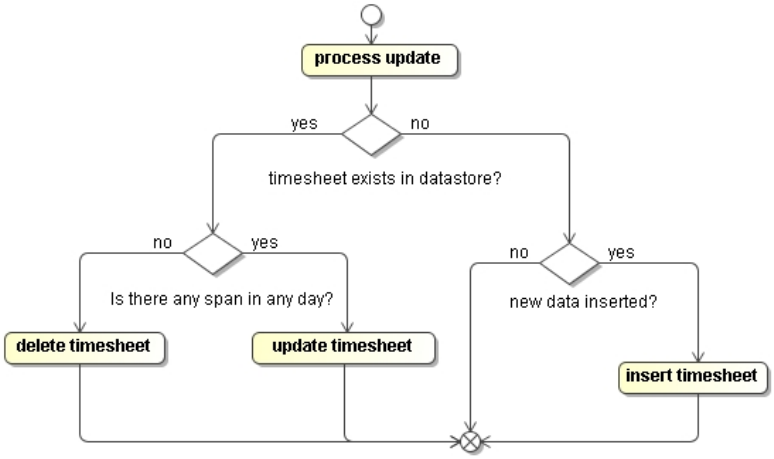

FIGURA 4.7 - PROCESSAMENTO DA ACTUALIZAÇÃO DE UMA TIMESHEET

O algoritmo de actualização das *timesheets* faz o processamento da nova informação, tendo em conta a informação já armazenada, tomando a decisão se a *timesheet* resultante deve ser inserida, removida ou actualizada. A Figura 4.8 mostra o processamento de cada dia.

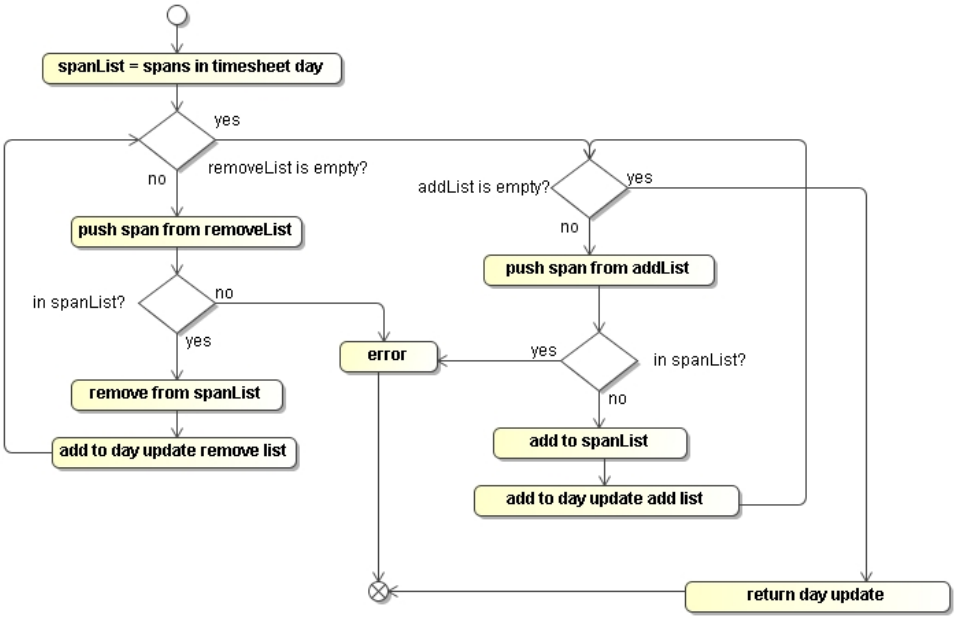

FIGURA 4.8 - PROCESSAMENTO DA ACTUALIZAÇÃO DE UM DIA DA TIMESHEET

A actualização da *timesheet* é validada por dia, onde é analisada a informação a remover e a adicionar, utilizando a informação já armazenada e gerando a informação necessária para transmitir à *DAL*. Na validação, é procurada a informação a remover e verificada se nenhuma hora de trabalho se sobrepõe. O formato de dados utilizado pela *DAL* é construído com o auxílio da classe *TimesheetManagerConverter*. Apesar da validação ser feita por dia, a informação de uma *timesheet* é transmitida toda de uma vez à *DAL,* permitindo que a camada de dados execute a operação sobre os dados da melhor forma possível - neste caso irá ser executada uma operação em *batch*, uma forma optimizada de utilização do serviço de dados.

A entidade *timeline* tem uma referência para a *timesheet* actual. Numa fase inicial do projecto, as *timesheets* não tinham uma chave que as identificasse pelo mês e pelo ano. Assim, era necessária uma referência para verificar se existia uma *timesheet* correspondente ao mês actual, sem ser necessário procurar por todas as *timesheets* existentes. A remoção desta característica está identificada como trabalho futuro, quando for modificada a API de acesso a dados e estudada a implementação da funcionalidade de geração de relatórios mensais.

Cada *timesheet* tem dois campos que servem para o controlo de versões, o campo *id* e o campo *version*. Estes campos foram incluídos para detectar a edição simultânea de *timesheets* por um consultor e pelo seu gestor, mas não chegou a ser implementada por não haver suporte para hierarquias. O objectivo não é permitir edição paralela, mas detectar se a versão da *timesheet* para a qual o utilizador está a enviar uma actualização corresponde à *timesheet* existente no serviço. Se a *timesheet* fosse modificada durante a sua edição, a aplicação cliente obteria a nova informação e faria a gestão das modificações feitas.

No funcionamento implementado, cada actualização sobre a *timesheet* faz com que o campo *version* seja incrementado. Como uma *timesheet* pode ser apagada do serviço se uma actualização remover todas as horas existentes, foi acrescentado o campo *id*. Este campo é atribuído a uma *timesheet* quando é criada, fazendo parte de um contador global associado a um utilizador, e é incrementado por cada *timesheet* criada. Assim, a detecção de conflitos da *timesheet* é feita com a combinação do campo *id* e do campo *version*. Quando uma *timesheet* não está armazenada na camada de dados do serviço, a sua informação é devolvida com um *id* de valor zero. Esta característica terá de ser conciliada quando for implementada o sistema de aprovação de acções, cujo funcionamento é explicado no Capítulo 6.1, estando sujeita a modificações.

4.7.2 FUNCIONAMENTO DE RECURSOS DE PROJECTOS

Em todo o sistema existe um conjunto de projectos, com um cliente associado, que pode ser associado aos consultores. Sobre estes projectos são definidas horas de trabalho, de forma a gerar relatórios sobre o trabalho investido em cada projecto. Os projectos, e respectivos clientes, são partilhados pelos utilizadores do sistema.

Devido a uma limitação do sistema de dados (detalhada no Capítulo 4.8) os projectos, por serem partilhados entre os utilizadores, não podem ser lidos na transacção de actualização das *timesheets*. Para haver garantia de consistência nas transacções, não existe forma de eliminar um projecto do sistema, em alternativa, este pode ser marcado como desactivado para não poder ser novamente utilizado. A eliminação de um projecto tem de ser feita de forma manual, por um administrador, de forma a garantir consistência no sistema. Por uma questão de consistência, os clientes também não podem ser eliminados sem ser de forma manual, uma vez que esta acção implicaria remover os projectos associados.

Identificada como trabalho futuro está a implementação de uma alternativa que contorna esta limitação, através do uso de uma fila de mensagens (i.e. *task queue*). Após a marcação de eliminação de um projecto este seria marcado como desactivado e um trabalho seria adicionado na fila de mensagens. O trabalho adicionado trataria de substituir as horas de trabalho no projecto a eliminar por outro projecto ou, em alternativa, eliminaria as horas de trabalho. Este trabalho repetir-se-ia até não existirem mais horas de trabalho associadas ao projecto, contornando assim limitações de tempo impostas pela plataforma ou alguma falha imprevista durante o processamento do pedido. Quando não existisse mais informação associada ao projecto este seria eliminado do sistema e o trabalho removido da fila.

4.7.3 INTERACÇÃO COM A CAMADA DE DADOS

Cada *manager* utiliza a camada de dados para persistir a informação, através de *data access objects* (i.e. *DAO*), que têm uma interface *CRUD*.

A camada de negócio disponibiliza uma interface que simplifica a invocação de operações, tendo em conta a informação recebida no serviço. A camada de acesso a dados, por outro lado, está desenhada para minimizar a quantidade de pedidos feitos ao serviço de dados, estando as suas operações estruturadas para receberem todos os dados necessários. Assim, se na execução de uma operação for necessária outra entidade do serviço, essa entidade tem de ser obtida através de outra operação da camada de dados. Este comportamento permite que a camada de dados oriente a sua correcta utilização à camada de negócio. Por exemplo, a interface das operações de actualização necessitam da entidade a actualizar porque no serviço de dados a inserção de uma entidade é feita com a mesma operação que uma actualização, sendo necessário que os dados sejam lidos antes para haver a certeza que existem. Embora este comportamento tenha de ser feito com uma transacção pela camada de negócio, pois é conseguido através de duas operações diferentes da camada de dados, a interface reforça-o de forma a evitar erros. A camada de dados está também adaptada para as operações da camada de negócio: quando uma operação não precisa que toda a informação relacionada com a entidade seja lida, é disponibilizada uma versão parcialmente carregada da entidade, através de objectos com uma interface *header*.

Cada operação executada na camada de dados tem um comportamento atómico. Deste modo, é da responsabilidade da camada de negócio iniciar uma transacção se for executada mais que uma operação sobre a *DAL* e for necessária a garantia de atomicidade. As transacções são iniciadas através da interface *ITransactionManager*, acessível em todos os *managers* através da classe base *BaseManagerImpl*.

As decisões da camada de dados, que garantem a atomicidade das operações disponíveis e requerem que os dados necessários sejam passados às operações a executar, permitem delegar responsabilidade transaccional para a camada de negócio, sendo apenas iniciadas transacções quando necessárias.

4.8 LÓGICA DE ACESSO A DADOS

A camada de acesso a dados utiliza o serviço de *datastore* para guardar os seus dados. O serviço de *datastore* não é uma base de dados relacional, tendo regras diferentes a ter em conta na concepção da arquitectura do sistema. Estes conceitos são expostos no *Anexo C – Conceitos do serviço Datastore*.

Cada *entity group* definido tem uma granularidade mínima, de forma a promover a concorrência no sistema. Uma entidade utilizador, descrita no Capítulo 4.5, é o *root* de um *entity group* e ancestor de todas as suas dependências, excepto das entidades projectos. Os projectos são partilhados por todos os utilizadores, tendo de pertencer a outro *entity group*, cujo *root* é o cliente do projecto. A granularidade por utilizador deve-se ao facto das funcionalidades de

negócio implementadas requererem a manipulação simultânea das entidades associadas a um utilizador (i.e. *User, ConsultantData, Timeline, Timesheet,* informação *das Timesheets*).

Para acesso ao *datastore* o *Google App Engine* disponibiliza a *low-level API* (33) (34), que fornece uma abstracção das chamadas ao serviço *datastore*. No entanto, é aconselhado o uso de bibliotecas de alto nível, que utilizam a *low-level API* mas que fornecem uma maior abstracção, permitindo a manipulação das entidades através objectos.

4.8.1 BIBLIOTECA DE ACESSO A DADOS

Na fase inicial de implementação do projecto estavam oficialmente disponíveis duas bibliotecas para utilização do *datastore*: *JDO* (i.e. *Java Data Objects*) (35) e *JPA* (i.e. *Java Persistence API*) (36). Ambas são bibliotecas para persistência e modelação de dados, que têm uma interface de alto nível e permitem a adaptação para qualquer sistema de dados, normalmente relacional. A principal diferença entre as duas bibliotecas está na sintaxe das anotações utilizadas, pelo que foi escolhida *JDO uma vez* que disponibiliza apenas a sintaxe para as operações suportadas no *datastore*, tornando-se mais fácil de utilizar.

Durante o desenvolvimento, a utilização da biblioteca *JDO* teve de ser abandonada devido a limitações que dificultam a implementação da lógica de negócio. Os problemas encontrados devem-se ao seu comportamento automático e à falta de documentação e apoio em grupos de discussão, que são enumerados de seguida.

A biblioteca *JDO* disponibiliza operações para inserção, actualização remoção e carregamento de dados. As operações executadas pela biblioteca *JDO* fazem automaticamente a gestão dos *entity groups* de cada entidade, que causa modelações erradas, sendo mostrado um caso específico na Figura 4.9.

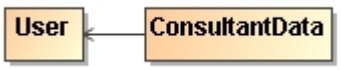

FIGURA 4.9 - RELAÇÃO ENTRE USER E CONSULTANTDATA

A estrutura *ConsultantData* tem uma referência para uma entidade utilizador, previamente criada. A entidade utilizador é o *root* do *entity group,* devendo ser o *ancestor* da entidade *ConsultantData*. No entanto, como *ConsultantData* refere o utilizador em vez de ser referida por este, a biblioteca tentar criar um novo *entity group* para *ConsultantData* e associá-la como *ancestor* do utilizador. Existe uma forma de contornar a situação, não documentada, que implica criar uma *unowned relationship* (conceito do *JDO*, não existente no *datastore*, que indica que a gestão feita sobre a entidade passa a ser manual) com uma anotação indicativa de que a entidade referida é o *parent*. Deste modo, uma *unowned relationship* é construída com uma referência para a chave da entidade.

Outra limitação reside no facto de a eliminação das entidades ser automática, não havendo suporte para duas referências, num objecto, para a mesma entidade. Também não existe suporte directo para polimorfismo, útil para o armazenamento de *spans*, que na arquitectura actual têm o mesmo tipo base. Além disto, não é possível controlar o carregamento dos dados do *datastore*, sendo carregados todos os filhos de uma entidade, o que pode implicar a leitura de uma maior quantidade de informação para além da necessária.

Os objectos têm um comportamento *attatched*, comportando-se como estando permanentemente ligados à base de dados, o que implica que quando é feita uma modificação num campo é feita uma nova inserção à base de dados. Por isto, os objectos têm de ser *dettached* de forma a controlar as modificações, mas este comportamento gera erros na inserção de novas entidades que tenham um *parent*, fazendo com que sejam inseridas em novos *entity groups.* Também não é possível partilhar entidades (necessário partilhar projectos e clientes), sendo criada uma nova entidade em vez de ser associada a entidade já existente.

Devido aos problemas descritos com a biblioteca *JDO* foram analisadas outras bibliotecas que apareceram numa fase posterior, com objectivo de resolver os problemas das bibliotecas de *JDO* e *JPA*. Durante o processo de escolha foram analisadas as bibliotecas *Objectify* (37), *TWiG* (38) e *Slim3* (39) que são desenhadas para a arquitectura do *datastore*, estão bem documentadas e que têm grupos de discussão para esclarecimento de dúvidas. Das três bibliotecas, tanto o *TWiG* como o *Objectify* se mostraram bons candidatos paea solucionar o problema.

A biblioteca *Objectify* usa uma abordagem que obriga a uma gestão manual das chaves e a inserção e eliminação manual de cada entidade. As entidades são referidas através das chaves. A biblioteca *TWiG* permite uma manipulação de mais alto nível, fazendo internamente a gestão das chaves, sendo as entidades manipuladas através dos objectos que as representam. No entanto, ao contrário da biblioteca *JDO*, permite ter controlo total sobre o formato das chaves e sobre gestão de *entity groups,* através de anotações, e a inserção de uma entidade cria automaticamente as entidades que refere. É também suportado polimorfismo, o que implica que as entidades tenham de ser eliminadas manualmente. O carregamento de uma entidade e das suas referências é feito de forma automática, mas é possível controlar a *profundidade* do carregamento.

A biblioteca *TWiG* foi seleccionada por disponibilizar uma *API* com uma maior abstracção que a biblioteca *Objectify*, fornecendo assim uma maior produtividade. A sua utilização é descrita *no Anexo D – Utilização da biblioteca TWiG*.

4.8.2 MODO DE REPLICAÇÃO

O serviço *datastore* usa uma arquitectura distribuída que gere a escalabilidade dos dados de forma automática. Disponibiliza dois modos de replicação: o *high replication datastore* e o

master/slave. O modo de replicação *high replication datastore* reproduz os dados por todos os *datacenters,* através de um custo nas escritas, usando uma variante do algoritmo *Paxos* (40). O modo de replicação *master/slave* reproduz os dados de forma assíncrona mas possui forte consistência, pois os dados são sempre escritos para o mesmo *datacenter* e as leituras são consistentes, através de períodos de tempo com dados indisponíveis. O funcionamento em *master/slave* foi escolhido por ter menores custos e por o sistema implementado dar prioridade a escritas de dados em vez de leituras. Este modo é escolhido durante o processo de registo da aplicação, influenciando o funcionamento de algumas operações e não podendo ser mudado.

4.8.3 ORGANIZAÇÃO

A *data access layer* (i.e. DAL) é a camada de acesso a dados. Toda a sua manipulação é feita através de interfaces que abstraem a sua concretização. Desta forma, se for necessário mudar o formato de armazenamento de dados, a camada de negócio mantém-se inalterada. Na Figura 4.10 pode ser visualizada a organização de *DAL*.

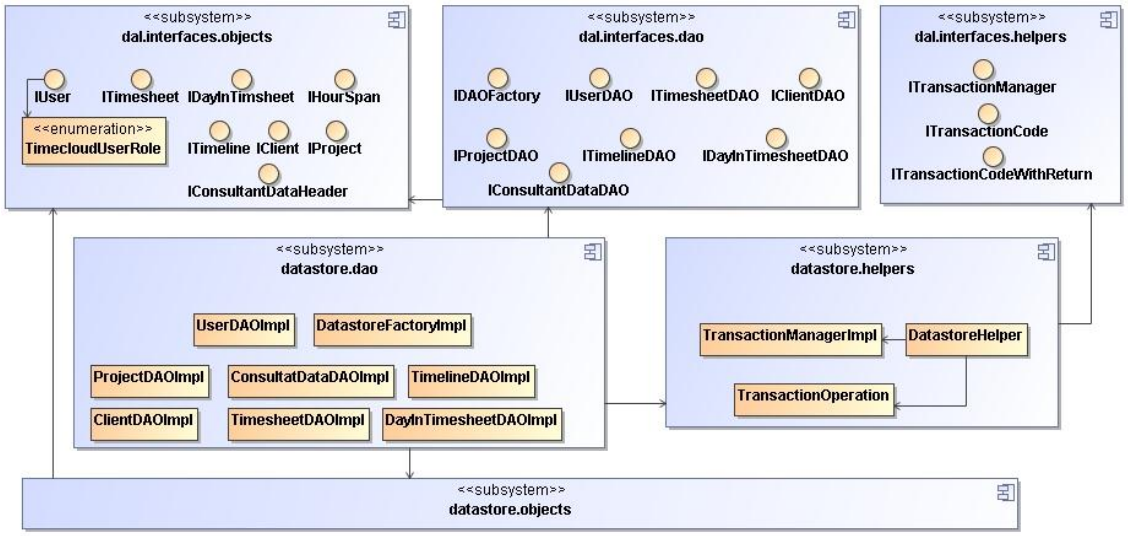

FIGURA 4.10 - CAMADA DE ACESSO A DADOS

A manipulação de dados é feita através de interfaces que que disponibilizam *data access objects* (i.e. *dal.interfaces.dao*) e da sua respectiva implementação (i.e. *datastore.dao*). Os dados são colocados em objectos (i.e. *datastore.objects*) e são acedidos externamente a partir da interface que implementam (i.e. *dal.interfaces.objects*).

Existe um *DAO* para cada entidade que é directamente utilizada, o qual disponibiliza operações *CRUD* (i.e. *create, remove, update e delete*) adaptadas às necessidades de negócio. Operações que não façam sentido no âmbito das necessidades de negócio não são disponibilizadas (existem entidades que não podem ser directamente eliminadas). São também disponibilizadas operações que optimizam a utilização do *datastore*, tais como a obtenção de

um conjunto de entidades em vez de uma de cada vez. Quando uma operação não é sucedida é lançada uma excepção bem definida (esta informação é detalhada no Capítulo 4.9). Uma operação de um *DAO* pede explicitamente um objecto que tem de ser obtido através de outro *DAO*, em vez de o obter internamente. Esta decisão simplifica a gestão dos dados e evita iniciar transacções desnecessariamente.

As operações disponíveis são atómicas de um ponto de vista externo. Cada operação mantém consistência usando o mecanismo transaccional para garantir a atomicidade de todas as suas acções quando é necessário inserir, actualizar e/ou eliminar múltiplas entidades.

4.8.4 ELIMINAÇÃO DE ENTIDADES

Quando uma entidade é eliminada, a implementação do *DAO* que executa a operação é responsável por eliminar todas as suas dependências. De forma a simplificar cada implementação, em cada *DAO* devem ser identificadas as dependências da entidade que manipula. O suporte para a eliminação das suas dependências é disponibilizado através das operações da classe base dos *DAO* mostradas na Listagem 4.3.

```
1  protected <T> void addMyChildren(
2                      HashMap<Class,LinkedList<Object>> map,
3                      Class<T> childClass,
4                      List<T> childs){..}
5
6  protected <T> IChildrenEntitiesRetriever<T> generateChildrenDeletionCode(){…}
7
8  protected abstract <T> void addChildrenToDelete(
9                      HashMap<Class,LinkedList<Object>> map,
10                     T entity);
```

LISTAGEM 4.3 - INTERFACE DE DAOBASEIMPL

A implementação da classe *DaoBaseImpl* permite obter as entidades a serem eliminadas através do método *generateChildrenDeletionCode,* mostrado na linha 6. O método recebe a entidade a eliminar e devolve uma concretização de *IChildrenEntitiesRetriever*, cujo método *getChildrenToDelete* permite obter todas as dependências da entidade, agrupadas por tipo. Esta abordagem é necessária poder obter todas as dependências dentro de uma transacção de forma a garantir atomicidade na eliminação das entidades.

O método *addMyChildren*, mostrado na linha 1, permite o registo de entidades a serem eliminadas num mapa com o formato *HashMap<Class,LinkedList<Object>>* onde as entidades a serem eliminadas são agrupadas por tipo.

A implementação do método *generateChildrenDeletionCode* recorre ao método *addChildrenToDelete*, mostrado na linha 8, para obter as dependências de uma entidade. Este método é implementado em todos os *DAO*, onde são adicionadas dependências da sua

entidade através do método *addMyChildren.* Por cada entidade adicionada é invocado o método *addChildrenToDelete* do *DAO* que a manipula, para que as suas dependências sejam também adicionadas à lista de dependências.

4.8.5 FACTORIZAÇÃO DE CÓDIGO

Existe código utilizado nas operações dos *DAO* que foi possível generalizar, como tratamento de excepções, gestão de transacções ou inserção, remoção e actualização de entidades. Para este fim foi criada uma classe auxiliar que agrega as operações comuns dos *DAO*, chamada *DatastoreHelper*. A classe pode ser visualizada na Figura 4.11.

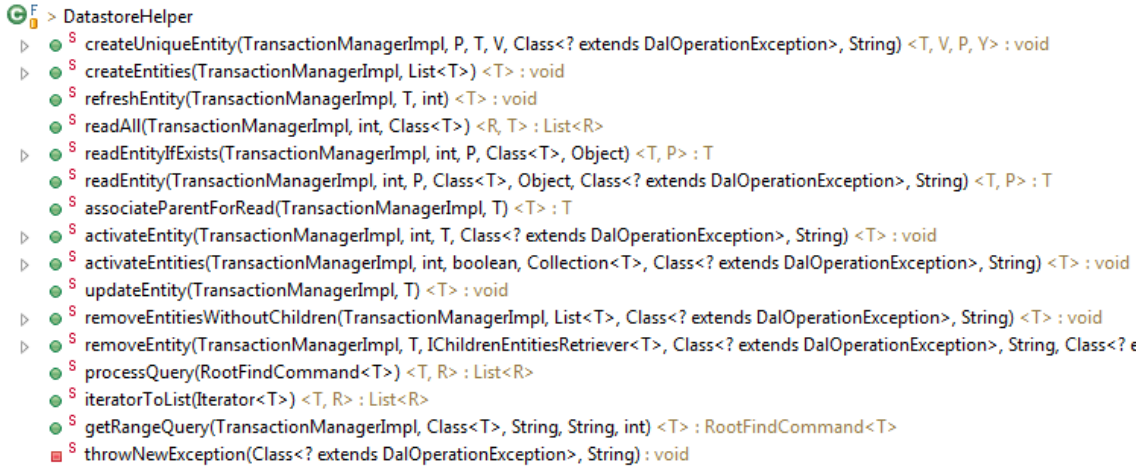

FIGURA 4.11 - ESTRUTURA DA CLASSE DATASTOREHELPER

Esta classe disponibiliza as operações mais comuns aos *DAO* recorrendo ao uso de genéricos de Java. As operações iniciam transacções para as operações, quando necessário, fazem o tratamento de erros e geram as excepções esperadas. Este comportamento é reforçado através da assinatura dos métodos, que recebem sempre o tipo de excepção a gerar em caso de erro. De salientar que a operação para eliminar uma entidade recebe uma implementação da interface *IChildrenEntitiesRetriever* para remoção das suas dependências, conforme descrito no *Capítulo 4.8.4 - Eliminação de entidades*.

4.8.6 EXECUÇÃO DE TRANSACÇÕES

A maioria das operações disponíveis no objecto *DatastoreHelper,* mostrado na Figura 4.11, recebe um objecto chamado *TransactionManagerImpl* por parâmetro, que disponibiliza suporte transaccional. Este objecto, disponibilizado pela interface *ITransactionManager*, fornece uma abstracção da API da *datastore*, permitindo a utilização de transacções na camada de negócio. A Figura 4.12 mostra a estrutura da classe *TransactionManagerImpl*.

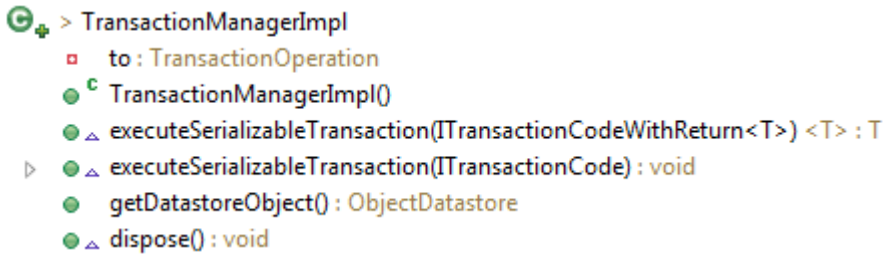

FIGURA 4.12 - OPERAÇÕES DO MÉTODO TRANSACTIONMANAGERIMPL

A classe mostrada tem o objectivo de fornecer um ponto central onde são iniciadas transacções. A partir da interface *ITransactionManager* são disponibilizados métodos para executarem as transacções e um método *dispose*, que deve ser invocado no final de cada pedido processado para garantir que todos os recursos são libertados. Na classe é ainda disponibilizado um método para obter o objecto com a instância do *TWiG*, sobre o qual são executadas as operações do serviço *datastore*.

Através dos métodos *executeSerializableTransaction* é possível serem iniciadas transacções sem se perder o controlo sobre a execução de uma transacção, assegurando que a transacção é concluída. O código deste método é mostrado na Listagem 4.4.

```
1  try{
2      to.begin();
3      result = code.execute();
4      to.commit();
5      ...
6  }
7  finally {
8      ...
9      if (to.isActive())
10         to.rollback();
11     ...
12 }
```

LISTAGEM 4.4 - EXECUÇÃO DE CÓDIGO TRANSACIONAL

O método *executeSerializableTransaction* executa o código que recebe por parâmetro (i.e. linha 3), garantindo que cada transacção executada é concluída com um *commit* ou um *rollback*. O código é executado através do método *execute* de uma classe que implemente a interface *ITransactionCode* ou *ITransactionCodeWithReturn* (se necessitar de retornar dados). Esta abordagem é baseada nos exemplos de boas práticas de uso do *datastore* (41). Um erro ocorrido numa transacção é traduzido para uma excepção do tipo *OperationUnsuccessfulException*. De forma a poderem ser lançadas excepções bem conhecidas na execução da transacção não é feito tratamento de erro para excepções que derivem de *OperationUnsuccessfulException*.

Nem o *datastore* nem a *API* usada têm suporte para transacções aninhadas. No entanto, foi acrescentado suporte para este tipo de transacções através da classe *TransactionOperation,* sendo usada no código de execução de uma transacção (i.e. referência *to*, na linhas 2,4,9 e 10 da Listagem 4.4). Desta forma, as transacções iniciadas são inseridas noutra transacção já anteriormente iniciada.

A arquitectura do sistema tem necessidade de transacções aninhadas pois apesar dos *DAO* garantirem a integridade dos dados (e.g. apagar uma entidade implica apagar todas as entidades dependentes) não garantem integridade do ponto de vista de lógica de negócio (e.g. atribuir um *role* a um utilizador resulta na criação de uma entidade *ConsultantData*). O suporte para estas transacções simplifica a escrita e manutenção de código pois assim as operações podem ser contidas, iniciando uma transacção quando necessitam, não estando dependentes de outras operações. A alternativa à utilização desta abordagem implicaria que as operações da lógica de dados não fossem atómicas de um ponto de vista externo, fazendo com que cada operação da lógica de negócio tivesse de iniciar uma transacção quando fosse necessária numa operação da lógica de dados, complicando a escrita e manutenção do código.

A Figura 4.14 mostra o funcionamento da classe *TransactionOperation*.

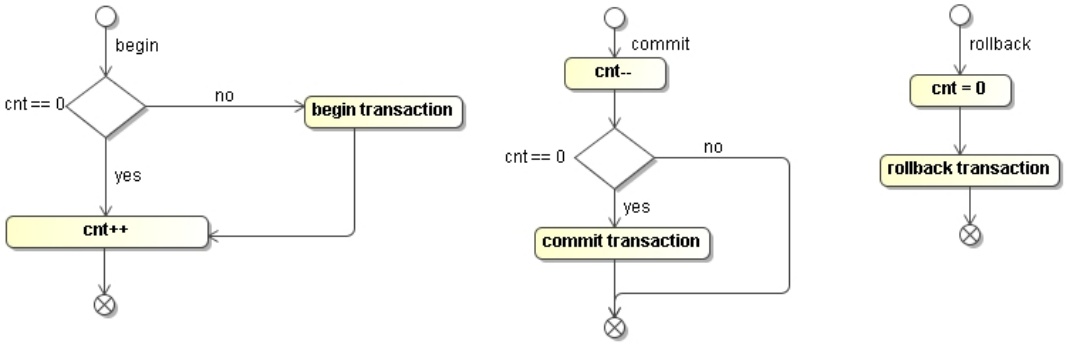

FIGURA 4.13- ALGORITMO DE SUPORTE A TRANSAÇÕES ANINHADAS

O objecto *TransactionOperation* só inicia uma transacção no *datastore* no primeiro pedido de *begin*, sendo só feito o *commit* da transacção quando todas as respectivas transacções internas forem terminadas com sucesso. Caso ocorra um *rollback* numa transacção interna, a transacção não sucede.

4.8.7 MUDANÇA DA CAMADA DE ACESSO A DADOS

Apesar da biblioteca *TWiG* ter resolvido os problemas da biblioteca de *JDO,* não é desprovida de outros problemas, que são enumerados de seguida.

A implementação da *API TWiG* usa uma cache interna, armazenada na instância (i.e. *ObjectDatastore*) sobre a qual são feitas as operações, para não fazer um pedido da mesma

entidade mais que uma vez. Este comportamento pode gerar problemas em transacções: um objecto que seja lido dentro de uma transacção que já tenha sido lido anteriormente (directa ou indirectamente) do *datastore* não será lido uma segunda vez, impossibilitando a garantia de isolamento *repeatable read*, Uma solução para resolver o problema passou pela utilização de uma nova instância de *ObjectDatastore* por cada transacção iniciada sobre o *datastore*. Esta abordagem não se mostrou viável porque a cache interna é usada em todas as operações, tendo de ser adicionadas manualmente à cache todas as entidades lidas fora da transacção para poderem ser usadas. Além disso, o esquecimento de uma associação gera erros, levando à inserção das entidades não associadas. Existem também algumas situações, como na actualização de uma *timesheet*, em que as entidades não são armazenadas no serviço de dados. Devido às razões descritas, esta ideia foi abandonada passando a ser disponibilizado um método nos *DAO,* usado pela camada de negócio, que permite pedir de novo dados anteriormente pedidos ao *datastore*. Devido à API do *TWiG,* esta solução implica ter conhecimento de leituras anteriores, sendo necessário o objecto que representa a entidade para pedir de novo os dados, o que limita a utilização das operações da camada de negócio.

A abordagem que permite que a gestão das chaves das entidades seja interna não é ideal: por exemplo, como quando se obtém uma entidade projecto ou os dados de consultor de um utilizador, a entidade necessária tem uma entidade *parent* com chave conhecida (i.e. através do seu identificador consegue-se gerar a chave). Usando esta *API*, para se poder obter a entidade é necessário obter a sua entidade *parent*, obrigando a leituras não necessárias ao *datastore.* A solução para este problema foi conseguida identificando os sítios com esta ocorrência, sendo instanciado um objecto que representa a entidade *parent* com o seu identificador, que é associado à cache, permitindo ler a entidade do *datastore* sem uma leitura prévia da sua entidade *parent*. No entanto, esta estratégia dificulta a utilização destas operações, tendo de ser usada de forma controlada para que outras operações não obtenham acidentalmente o objecto falso associado à cache. Além disso, é possível que existam situações em que esta ocorrência não esteja identificada, sendo feitas leituras desnecessárias.

Os problemas desta biblioteca não causam ao mau funcionamento do sistema, no entanto, complicam o desenvolvimento de novas funcionalidades e a manutenção de código. Todos os problemas que decorreram na utilização desta biblioteca foram discutidos directamente com o autor do *TWiG*, John Patterson, através dos grupos de discussão (42). Apesar dos erros detectados na biblioteca serem rapidamente corrigidos, não existem pretensões de se resolverem os problemas mencionados neste capítulo. Concluiu-se que o autor da *API* pretende que esta biblioteca seja utilizada para aceder ao serviço *datastore,* tentando simplificar o processo o mais possível, não sendo adequada para uso numa camada de acesso a dados. Além disso, a *API* ainda não é muito madura, havendo funcionalidades cuja sintaxe muda constantemente e não documentadas.

Apesar de se ter tentado mudar novamente a *API* de dados para uma solução mais *low-level*, de forma a colmatar as dificuldades apresentadas, tal não foi possível até à conclusão do projecto, ficando para trabalho futuro o estudo e implementação da biblioteca *Objectify*. Esta biblioteca tem grande apoio pela sua comunidade e, após uma análise superficial, leva a crer que por ser mais baixo nível possibilita a sua utilização na implementação da camada de dados.

4.9 TRATAMENTO DE ERROS

O sistema de tratamento de erros foi planeado com o objectivo de ser o ponto central de tratamento dos erros da plataforma. O sistema é disponibilizado através de um filtro, implementado pela classe *RequestCleaner*. Este filtro tem dois objectivos: processar os erros que ocorram durante o processamento da resposta a um pedido e libertar os recursos resultantes desse processamento. O filtro permite apanhar a excepção lançada por qualquer *servlet* ou filtro executado após a sua execução. Pelo filtro passam tanto pedidos para páginas (i.e. aplicação cliente) como pedidos para o serviço, Por este motivo o pedido tem de ser distinguido de modo a ser enviada uma resposta adequada, enviando uma indicação de erro em *HTML* ou um código de erro adequado na resposta em *HTTP*.

4.9.1 TRADUÇÃO DE ERROS

As excepções lançadas no processo de autorização e nas operações de serviço derivam da classe *TimecloudException*. A sua hierarquia é mostrada na Figura 4.14.

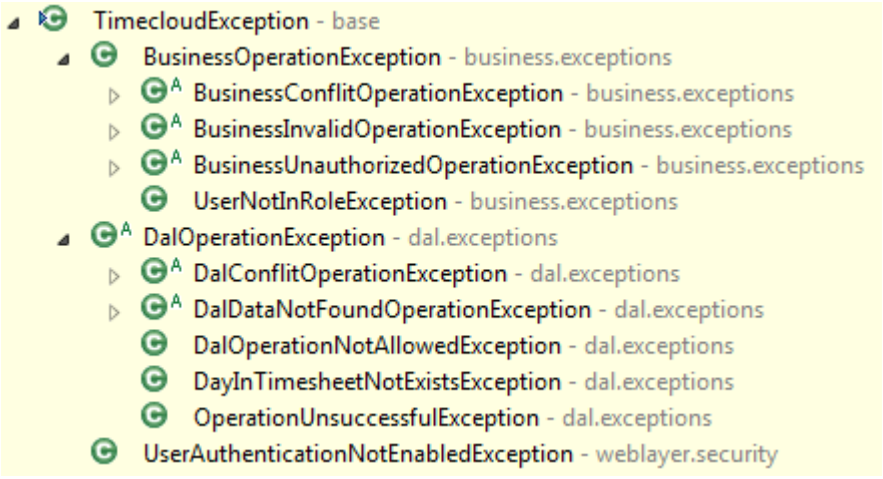

FIGURA 4.14 - HIERARQUIA DE EXCEPÇÕES UTILIZADAS

As classes mostradas dividem-se de acordo com a camada onde são lançadas (i.e. negócio ou camada de acesso a dados). Dentro de cada camada existem excepções que categorizam os tipos de erro que podem ocorrer, de onde derivam as excepções mais especializadas. Esta

categorização permite a tradução de excepções para códigos de erro, detalhados na Tabela 4-1.

Tipo de excepção	Resposta
OperationUnsuccessfulException	Internal Error (500)
DalConflitOperationException	Conflit (409)
DalDataNotFoundOperationException	Not Found (404)
UserNotInRoleException	Forbidden (403)
BusinessConflitOperationException	Conflit (409)
BusinessInvalidOperationException	Bad Request (400)
BusinessUnauthorizedOperationException	Unauthorized (401)

TABELA 4-1 - MAPEAMENTO USADO PARA CÓDIGOS DE ERRO

As excepções que podem ocorrer em tempo de execução devido à operação pedida ser inválida são traduzidas para os códigos indicados. As excepções que não devam ocorrer em tempo de execução geram um erro interno (i.e. *internal error* - código 500).

4.9.2 IMPLEMENTAÇÃO

A abordagem centralizada não é possível de implementar devidos ao uso das tecnologias *Jersey* e *JSP*. Ambas as tecnologias possuem um mecanismo de processamento de erros para um tratamento localizado. Por este motivo, quando ocorre uma excepção não tratada o próprio mecanismo processa a excepção e envia uma resposta por omissão, não permitindo processar o erro no filtro. Assim, nestas tecnologias tem de existir um tratamento de erros local.

O tratamento de erros gerados no pedido da aplicação cliente é feito através do mecanismo *JSP* que permite definir uma página de erro que é mostrada quando ocorre uma excepção. No serviço, por outro lado, tem de ser mostrado o erro adequado à razão pela qual a operação falhou.

O mapeamento de erros em Jersey é feito através de uma classe que implementa *ExceptionMapper<Excepção>*. A classe é injectada na *API* de Jersey e a resposta com o erro é obtida através do método *toResponse*. O tratamento de erros foi centralizado nesta classe, sendo esta classe também usada pelo filtro onde são tratados os erros que ocorram noutros sítios do serviço, como no processo de autorização.

A classe implementa *ExceptionMapper<Throwable>*, processando qualquer excepção que possa ocorrer. Internamente, os códigos de erro estão mapeados de acordo com o tipo da excepção. Para o mapeamento de erros é procurado o código correspondente à excepção

lançada, sendo usada a hierarquia da excepção até ser encontrada uma correspondência. A resposta gerada tem o código de erro e uma mensagem com detalhes do erro ocorrido. Caso não seja encontrada uma correspondência entre a excepção e um código de erro é indicado um erro interno. Sempre que existe um erro interno, seja por não ter mapeamento ou por estar marcado que a excepção gerada não deve ocorrer em tempo de execução, é registada a informação no *log*.

4.10 NOTAS ADICIONAIS

4.10.1 INJECÇÃO DE DEPENDÊNCIAS

Na plataforma é usado um mecanismo de injecção de dependências, de forma a abstrair as três componentes principais umas das outras: a estrutura da plataforma, que autoriza e processa os pedidos, a camada de negócio e a camada de dados. A injecção de dependências é feita com *Guice* (43) (44) (45), por ser uma plataforma simples de usar e menos complexa que outras disponíveis, como *Spring*.

O injector de dependências é instanciado através de uma componente *listener.* Quando a aplicação é iniciada, o *listener* instancia o injector e coloca-o no *ServletContext*, ficando disponível para todos os pedidos. O injector é *thread-safe*, podendo ser usado em cenários de concorrência. É também usado um filtro para utilizar *web-scopes* no injector, que permite que os objectos injectados sejam instanciados por pedido (i.e. *request HTTP*). Os objectos injectados são instâncias de *ManagerFactoryImpl* e de *DAOFactoryImpl* e implementam as interfaces *IManagerFactory* e *IDAOFactory*, respectivamente. Estas interfaces permitem a utilização do padrão de desenho *Factory* na camada de negócio e na camada de acesso a dados. As interfaces são mostradas na Figura 4.15 e na Figura 4.16.

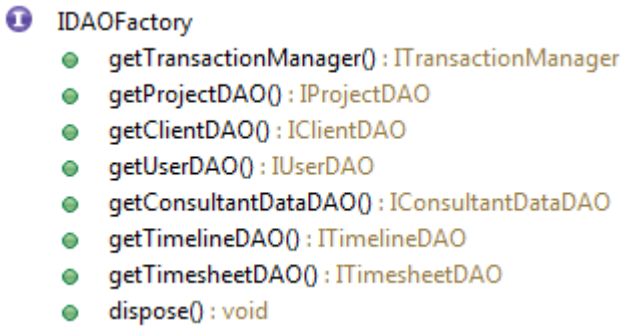

FIGURA 4.15 - INTERFACE DE IDAOFACTORY

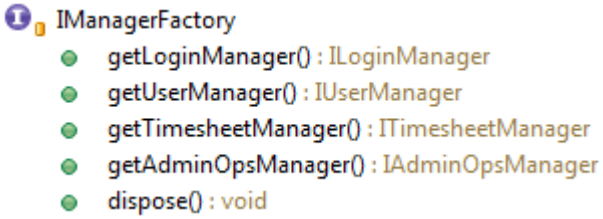

FIGURA 4.16 - INTERFACE DE IMANAGERFACTORY

A interface *IDAOFactory* permite obter os *DAO* do sistema e a implementação de *ITransactionManager*. Os objectos só são instanciados quando necessários, uma única vez por pedido, usando uma técnica de carregamento tardio. É através do método *dispose* de *IDAOFactory* que se libertam os recursos dos objectos utilizados.

A interface *IManagerFactory* permite obter qualquer objecto *manager*. De forma semelhante aos *DAO*, os *managers* só são instanciados quando necessário. A classe *ManagerFactoryImpl* necessita de uma implementação de *IDAOFactory,* injectada no construtor, para a construção dos *managers*. A implementação do método *dispose* liberta os recursos de *IDAOFactory*

Tanto os objectos *manager* como os objectos *DAO* requerem na sua construção a interfaces IDAOFactory e/ou IManagerFactory quando precisam de obter uma implementação de um *DAO*, *ITransactionManager* ou *manager* nas suas operações. Esta abordagem permite tirar proveito do mecanismo que só instancia os objectos quando necessário.

A camada negócio é usada pelo mecanismo de autorização do *Spring-Security* e pelos *endpoints* definidos em *Jersey*. O objecto que implementa *IManagerFactory* é obtido através do objecto *BusinessProvider*, o único objecto que utiliza o injector. A implementação do método *cleanUp* de *BusinessProvider* faz a libertação dos recursos, invocando o método *dispose de IManagerFactory*. Este método é invocado no final do processamento de cada pedido, através do filtro do *ServletContainer* implementado pelo objecto *RequestCleaner,* que é também usado no tratamento de erros.

4.10.2 EDIÇÃO DE TIMESHEETS

A edição das *timesheets* tem um problema com a notação *JSON*, que não permite distinguir os tipos dos objectos. Desta forma, não é possível distinguir os tipos das horas marcadas numa *timesheet* (i.e. horas de trabalho, férias, faltas) quando estão na mesma lista, sendo cada tipo de hora de trabalho identificado através de um campo. O processamento do pedido de actualização de uma *timesheet* é auxiliado no processo de deseriação dos dados recebidos, através da classe a *HourSpanTObjAdapter*, de forma a instanciar criar os tipos adequados. Esta classe deriva de *XmlAdapter* e a sua utilização é indicada através da notação *XmlJavaTypeAdapter* no objecto de actualização onde os dados recebidos são colocados.

4.10.3 SESSÃO

A implementação da *API Spring Security* recorre a um objecto de sessão para autenticar um utilizador que já tenha feito um pedido anteriormente. Como a aplicação está hospedada no *Google App Engine*, podendo ser executada um múltiplas máquinas, a sessão é guardada nos serviços de *datastore* e de *memcache* de forma a ser partilhada por todas as instâncias (46) . A utilização de sessão tem de ser activada explicitamente, acrescentando um elemento *sessions-enabled* com o valor *true* no ficheiro *appengine-web.xml*. A utilização de dados em sessão em vez do uso explícito do *datastore* é mais vantajoso devido à utilização combinada com o serviço *memcache*.

4.10.4 LIGAÇÕES SEGURAS

A decisão de utilização de *HTTPS* deveu-se à necessidade de negócio de ter garantia de confidencialidade e integridade da informação transmitida.O *Google App Engine* tem suporte para *HTTPS*, que é conseguido acrescentando a configuração mostrada na Figura 4.17, extraída do ficheiro *web.xml*.

```
1  <security-constraint>
2      <web-resource-collection>
3          <url-pattern>/*</url-pattern>
4      </web-resource-collection>
5      <user-data-constraint>
6          <transport-guarantee>
7              CONFIDENTIAL
8          </transport-guarantee>
9      </user-data-constraint>
10 </security-constraint>
```

FIGURA 4.17 - CONFIGURAÇÃO DE UMA LIGAÇÃO SEGURA

Esta configuração indica que todos os pedidos feitos para qualquer *URL* (ver o elemento *url-pattern*) têm de ser feitos através de uma ligação segura. Para a ligação é usado um certificado assinado pelo *Google Internet Authority* que é usado em todas as aplicações em domínios *appspot.com* que usam esta configuração.

4.11 TRABALHO FUTURO

Existem implementações que não puderam ser concretizadas, ficando assinaladas como trabalho futuro. Por implementar ficaram as funcionalidades descritas o Capítulo 3 - Linha de projecto – o mecanismo de validação de acções, o sistema de regras e o sistema de avisos. No entanto, chegou a ser feito algum planeamento sobre estas funcionalidades, que é detalhado no Capítulo 6.1 – Desafios futuros. Existem também melhorias que podem ser feitas sobre as funcionalidades já implementadas, as quais são mencionadas de seguida.

O sistema de autenticação pode ser mais flexível. O sistema funciona para casos em que o cliente está no mesmo domínio que o serviço, mas como só é autorizado um domínio, quando os controlos da aplicação cliente estão integrados num portal não é possível partilhar o mecanismo de autenticação com o portal. Uma solução passa pela utilização de *i-frames* dentro do portal, de forma a manter o domínio usado.

A edição de recursos do sistema, como os utilizadores, os clientes e os projectos, pode passar a incluir uma componente de controlo de versões de forma a notificar o utilizador quando os dados que estão a ser alterados mudam. Ainda em relação aos recursos cliente e projecto, estes têm uma interface e comportamento muito semelhante na camada de negócio e na camada de dados. Assim, pode ser analisada uma solução de factorização para operações comuns sobre recursos do sistema, de forma a eliminar código idêntico.

A mudança da *API* de acesso a dados, conforme explicada na lógica de acesso a dados, deve ser mudada. Na lógica de dados pode ainda ser melhorado o sistema de procura, que foi implementado através de filtros de *inequalidade*, que permitem procurar por campos que comecem por uma expressão. No entanto, os filtros não se comportam como o operador *like* existente nas bases de dados relacionais, não permitindo procurar variantes, podendo ser melhorado neste sentido. Além disso, as interrogações são *case-sensitive*, tendo o formato de armazenamento de dados dos campos a serem interrogados ser ajustado para suportarem interrogações que não sejam *case-sensitive*.

Capítulo 5

IMPLEMENTAÇÃO DA APLICAÇÃO CLIENTE

A aplicação cliente foi desenvolvida para validar, por funcionalidade, as operações do serviço. A aplicação consiste num conjunto de páginas *HTML* que comunicam com o serviço fazendo pedidos *AJAX*, permitindo validar os dados (i.e. *JSON* ou *XML*) gerados. O aumento da complexidade do projecto levou à sua restruturação, aumentando a sua usabilidade para permitir que as funcionalidades do serviço sejam validadas em conjunto. Os controlos da aplicação servem como referência acerca da comunicação entre o cliente e o serviço, permitindo sua integração noutras páginas *web* (e.g. integração no portal interno de uma empresa).

5.1 ORGANIZAÇÃO DO SERVIDOR *WEB*

O s*ervlet* que gera o conteúdo *HTML* do cliente e o *servlet* que disponibiliza o serviço estão alojados no mesmo domínio e no mesmo *ServletContainer*. Esta opção permite partilhar o mecanismo de autorização (e consequentemente de autenticação) do serviço e usar a informação utilizada no processo de autorização do utilizador. O conteúdo das páginas enviadas ao cliente é personalizado com as funções do utilizador que as requisita, mas só contém controlos para executarem operações *AJAX* com pedidos ao serviço. Para a geração da página não é executada nenhuma operação disponibilizada pelo serviço.

As páginas são geradas através da tecnologia *JSP*, escolhida pela simplicidade de utilização. De forma a ter um visual consistente por todas as páginas foi utilizada a biblioteca *JSP templates* (28). A biblioteca *JSP templates* permite a definição de um modelo (i.e. *template*) para geração das páginas (ver mais no *Anexo F – JSP Templates*). A Figura 5.1 mostra o formato da página modelo usada.

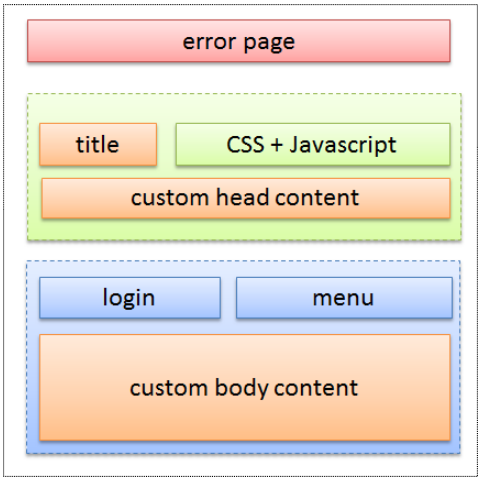

FIGURA 5.1 - MASTER PAGE DA APLICAÇÃO CLIENTE

A página modelo (i.e. *master page*) contém a informação presente em todas as páginas: o controlo de *login*, o *menu* com acesso às páginas disponíveis (sendo o seu conteúdo personalizado), indicação de página em caso de erro, informação de estilo (CSS) e bibliotecas comuns. A *master page* permite injecção de informação no título (i.e. *title)*, *head* e *body*, sendo assim indicada a informação específica de cada página. A Listagem 5.1 mostra o formato de uma página *JSP* que usa a *master page*.

```
1  <%@ taglib uri='/WEB-INF/tlds/template.tld' prefix='template' %>
2
3  <template:insert template='/resources/pages/master.jsp'>
4    <template:put name='title'
5                  content='Timesheets' direct='true'/>
6    <template:put name='head'
7                  content='/resources/content/timesheet/head.jsp'/>
8    <template:put name='body'
9                  content='/resources/content/timesheet/body.jsp' />
10 </template:insert>
```

LISTAGEM 5.1 - UTILIZAÇÃO DE JSP TEMPLATES

A Listagem 5.1 mostra o conteúdo da página que permite a editação do conteúdo de uma *timesheet*. O título da página é directamente indicado enquanto que os restantes pontos de injecção, devido a terem conteúdo mais complexo, são injectados com o conteúdo do ficheiro indicado. Tendo em conta esta estrutura, o servidor *Web* foi organizado como mostrado na Figura 5.2.

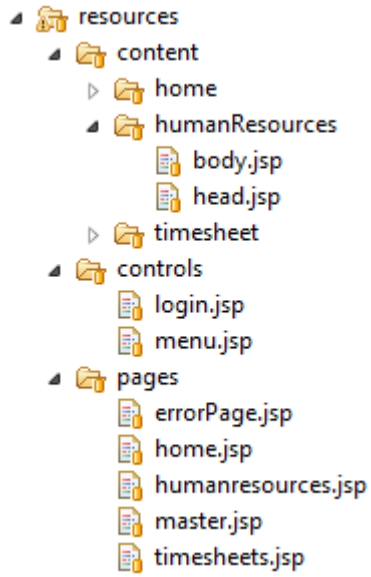

FIGURA 5.2 - ORGANIZAÇÃO DO SERVIDOR WEB

Cada página é colocada na pasta de nome *pages*, sendo o seu *URL* definido através do ficheiro de configuração. O conteúdo de cada página, disponibilizado através de ficheiros com a informação do *head* e do *body*, está colocado numa pasta com o nome da página, dentro da pasta *content*. Existe também uma pasta dedicada aos controlos existentes, chamada *controls*.

5.2 ESTRUTURA

A implementação da aplicação cliente faz a comunicação com o serviço através de pedidos *AJAX* e a actualização a *GUI* através da manipulação de *DOM*. Para aumentar a produtividade no seu desenvolvimento foi usada a biblioteca *JQuery* (29) (30) por ter boa documentação e uma *API* robusta. A biblioteca *JQuery-UI* (31) também foi usada para facilitar o desenvolvimento de uma *GUI* consistente. Esta preferência deveu-se ao facto de ser usada a mesma sintaxe e lógica do *JQuery* permitindo acelerar o processo de aprendizagem. A utilização destas bibliotecas tem também a vantagem de ser gerado código compatível com os browsers mais utilizados.

O cliente está dividido por controlos, existindo um controlo para manipular de cada recurso do serviço. Os controlos têm suporte para mudança de idioma, conseguido através de um objecto dicionário (i.e. *Dictionary*) de onde é retirada a informação para os nomes dos campos dos controlos construídos dinamicamente. Para construção de uma arquitectura orientada a objectos foi usado um mecanismo de suporte para herança, seguindo a abordagem sugerida por *John Resig – Simple JavaScript Inheritance* (5), que permite a construção e herança de classes através de declarações de objectos em notação *JSON*.

Cada controlo é composto por três componentes: *builder, controller* e *proxy*. Esta separação é baseada no padrão de desenho *MVC* (i.e. *model-view-controller*) e tem como objectivo dividir

as responsabilidades de cada controlo em componentes. No componente *builder* é feita a gestão da interface gráfica que interage com o utilizador (i.e. *view*), sendo que o resultado das acções é processado no componente *controller* (i.e. *controller*) que disponibiliza *handlers* à componente *builder* para o processamento das acções do utilizador. O resultado é depois enviado para o serviço (i.e. *model*), cuja interacção é feita através do componente *proxy,* onde são feitos os pedidos e processadas as respostas recebidas.

5.3 ARQUITECTURA

5.3.1 CLASSES PROXY

Cada recurso tem um componente *proxy* associado que deriva da classe base *Proxy*. Esta classe disponibiliza métodos para os pedidos *HTTP* a fazer ao serviço: *request* (i.e. *GET*), *put* (i.e. *PUT*), *post* (i.e. *POST*) e remove (i.e. *DELETE*), que podem ser observados na Listagem 5.2.

```
1  class Proxy{
2
3      Proxy(baseURL);
4
5      void request(resource,success,failure);
6      void put    (resource, data, success,failure);
7      void post   (data, success,failure);
8      void remove (resource,success,failure);
9  }
```

LISTAGEM 5.2 - CLASSE BASE PROXY

Os métodos disponibilizados na classe *Proxy* são um invólucro para pedidos *AJAX* executadas através da *API* do *JQuery*. Os métodos recebem, por parâmetro, a informação necessária ao pedido (e.g o método *put* recebe o identificador do recurso) e duas funções para processamento do resultado do pedido: uma função para ser invocada em caso de sucesso e outra para ser invocada em caso de falha. A classe serve de base a outras classes *proxy*, existindo uma para cada recurso. Na Listagem 5.3 é mostrada a classe *proxy* usada na manipulação de uma *timesheet*.

```
1  class TimesheetProxy
2  {
3      TimesheetProxy();
4
5      void getTimeline(startMonth, startYear, length,success, failure);
6      void getCurrentTimesheet(success, failure);
7
8      void getTimesheet: function(year, month, success, failure){
9          this.request("/" + year + "/" + month, success, failure);
10     }
11
12     void updateTimesheet(year, month, timesheetUpdateData,
13                          success, failure);
14 }
```

LISTAGEM 5.3 - PROXY PARA MANIPULAÇÃO DO RECURSO TIMESHEET

A classe *TimesheetProxy* disponibiliza os métodos de acordo com as operações existentes no serviço. Esta estratégia permite criar objectos *proxy* específicos para cada recurso, sendo feito o utilizados os métodos da classe base para processar as operações disponíveis, como mostrado na linha 9 da Listagem 5.3.

5.3.2 CLASSES BUILDER

Os elementos gráficos são geridos por uma classe *builder*. Existe uma classe por cada recurso existente, pois cada recurso tem dados específicos a serem mostrados. Estas classes não são responsáveis pela lógica do controlo; as acções que necessitem de validação de dados ou comunicação com o servidor são delegadas para o respectivo *handler*, definido quando o objecto é iniciado. Existe um *handler* para cada acção, ao qual é passada a informação necessária para processar o seu processamento.

Os objectos builder disponibilizam operações comuns, que permitem mostrar ao utilizador ecrãs de carregamento de dados ou ecrãs de informação. Estas operações estão reunidas na classe *BaseBuilder*, cujos métodos estão listados na Listagem 5.4.

```
1  class BaseBuilder{
2
3      BaseBuilder();
4
5      void showLoadingScreen(control);
6      void showInfoMessage(control, title, message);
7      void showErrorMessage(control, message, onTryAgain);
8
9      void showControls(control) ;
10     void enable(control);
11
12     void setLoadingInControl(loadingControl);
13     void setTip(control, text);
14
15     boolean validateInputs(inputs, tipContainer);
16 }
```

LISTAGEM 5.4 - CLASSE BASEBUILDER

Todos os objectos *builder* derivam da classe *BaseBuilder.* A classe *BaseBuilder* disponibiliza operações para serem usadas tanto por especializações da classe como externamente, por objecto que faça a sua manipulação (i.e. objectos *controller*). A Listagem 5.4 não mostra todos os métodos da classe *BaseBuilder*, existindo também operações que neutralizam a acção de cada método mostrado. Dos métodos da classe *BaseBuilder* destacam-se os métodos de validação de dados (i.e. linha 15) e os métodos para mensagens de carregamento tardio (i.e. linhas 5, 6 e 7).

A implementação do método de validação de dados faz a verificação do conteúdo das caixas de texto de um formulário. A validação é feita validando o tamanho e o formato do texto introduzido. O formato do texto é validado com a expressão regular definida no atributo *validationRegexp* das caixas de texto, existindo um valor utilizado por omissão. Em caso de erro, o controlo é visualmente destacado e é mostrada uma mensagem de erro, construída com a informação colocada no atributo *validationName*.

Os métodos que permitem mostrar mensagens de carregamento tardio (i.e. *lazy messages*) são utilizados para mostrar mensagens que aparecem regularmente, permitindo indicar carregamento de dados ou mostrar erros de comunicação. Na Listagem 5.5 é mostrado o sistema de gestão de *lazy messages*.

```
_showLazyMessage: function(control, propName,
                           builderFunction, updateViewerFunction){

    if (control.prop(propName) == null)
    {
        //appended by builder function to have a more flexible location
        var lazyControl = builderFunction();
        control.prop(propName, lazyControl);
        lazyControl.addClass(this.LAZY_CONTROL);
    }
    else
    {
        var messageViewer = control.prop(propName);
        if (updateViewerFunction != null)
            updateViewerFunction(messageViewer);
        messageViewer.show();
    }
}
```

LISTAGEM 5.5 - FUNCIONAMENTO DE UMA *LAZYMESSAGE*

A utilização do sistema de gestão de *lazy messages* evita a criação de novo conteúdo cada vez que se quer mostrar uma mensagem num controlo. A implementação do método faz a verificação da propriedade indicada verificando se já existe uma referência para o controlo da mensagem. Se o controlo não existir, é criado e armazenado nessa propriedade; caso contrário a mensagem é actualizada e o conteúdo tornado visível. O método *showControls* e o seu complementar, *hideControls*, não interferem com este tipo de controlos.

De forma a se poder centralizar a manipulação da informação, foi usado um mecanismo de *data-binding*. Este mecanismo permite separar os dados do aspecto gráfico, sendo o aspecto gráfico automaticamente actualizado quando são executadas operações sobre os dados, disponibilizando assim o padrão de desenho *model-view-viewmodel.* Para atingir este fim, é usada a biblioteca *Knockout.js* (6). É ainda usada a *API* de *templates* de *JQuery*, tanto nativamente como em conjunto com a biblioteca *Knockout.js*, que permite a criação de controlos, de forma dinâmica, a partir de *templates* declarados estaticamente. A utilização de *Knockout.js* é analisada no *Anexo G – Notas de knockout.js* e a utilização de *JQuery Templates* no *Anexo H – Notas sobre o uso de JQuery Templates*. Note-se que estas bibliotecas não foram utilizadas desde o início do projecto, pelo que existem controlos que foram adaptados posteriormente ou que não as utilizam. O controlo de gestão de utilizadores, por exemplo, não utiliza o sistema de *templates*.

Para a utilização do mecanismo de *data-binding*, os objectos recebidos do serviço têm de ser modificados pela implementação do objecto *builder*, sendo acrescentados campos aos objectos ou modificados os seus tipos. A modificação destes objectos é gerida pelo objecto *builder*, sendo corrigido o formato dos seus campos quando os objectos têm de ser utilizados fora do componente (e.g. quando invocado um *handler* para processamento).

5.3.3 CLASSES CONTROLLER

A implementação dos objectos *controller* é responsável pela coordenação das acções de manipulação de recursos, recorrendo aos respectivos objectos *builder* e *proxy*. Os objectos *controller* são instanciados quando uma página é carregada, sendo o seu funcionamento iniciado com a invocação do método *start*. As classes *controller* recebem na sua construção as instâncias de *proxy*, permitindo a sua partilha sem haver dependências entre objectos *controller*. Como excepção existe a implementação da classe *controller* da *Timeline*, necessita da referência para o *controller* da *timesheet* para poder fazer o carregamento de uma *timesheet* seleccionada. Esta dependência pode ser removida, se necessário, através de um *handler*.

Os *handlers* registados nas classes *builder* são métodos da classe *controller*, através dos quais é feito o processamento dos pedidos do utilizador e a validação semântica dos dados inseridos (a validação da informação nos controlos é feita na classe *builder*). A Listagem 5.6 mostra uma função onde é feito o processamento de um pedido feito por um utilizador.

```
1  onGetItems: function(){
2      (...)
3      this._startHeaderAction();
4
5      this.proxy.getInterval(this.itemIdx,this.itemSetLength,
6              function(items)
7              {
8                  builder.clearLoadingScreen(body);
9                  if (items == null || items.length == 0){
10                     (...)
11                     builder.showInfoMessage(body,
12                                             dictionary.common.warning,
13                                             strMessage);
14                 }
15                 else{
16                     (...)
17                     for (var i in items)
18                         builder.insertItem(items[i]);
19                     builder.endInsertItems();
20                 }
21                 builder.enable(header);
22             },
23             function(status){
24                 builder.clearLoadingScreen(body);
25                 builder.showErrorMessage(body,
26                                          dictionary.common.connectionFailed,
27                                          function() { outer.onGetItems(); });
28                 builder.enable(header);
29             });
30 },
```

LISTAGEM 5.6 - PROCESSAMENTO DE UMA OPERAÇÃO QUE COMUNICA COM O SERVIÇO

Na Listagem 5.6 é mostrado o método genérico que faz o processamento do pedido de um conjunto de recursos. A implementação do método começa por mostrar uma mensagem de carregamento de informação e bloquear o controlo (i.e. linha 3), para que o utilizador não

interfira enquanto o pedido é feito. O pedido é feito através do objecto *proxy*, a qual é passada a informação a enviar no pedido e as funções que processam o resultado do pedido em caso de sucesso e em caso de falha, necessárias por ser feito de forma assíncrona. As funções, quando invocadas, são responsáveis por remover a mensagem de carregamento de dados e desbloquear o controlo. Quando o pedido é bem-sucedido, é mostrada a informação obtida. Quando pedido falha é mostrada uma mensagem de erro que permite voltar executar a operação.

5.4 DESAFIOS

Esta secção contém informação sobre das decisões de desenho dos controlos do sistema e a explicação sobre a implementação dos algoritmos mais elaborados na aplicação cliente.

5.4.1 RECURSOS DO SISTEMA

A secção de recursos humanos da aplicação permite a gestão dos recursos do sistema. Existem controlos para três recursos: utilizadores, clientes e projectos. Estes controlos têm funcionalidades comuns que permitem a criação, edição e procura de recursos, que se reflecte nas operações do serviço e na *GUI*, estando os elementos comuns das classes *builder* e das classes *controller* factorizados numa classe base. A hierarquia das classes *builder* é mostrada na Figura 5.3.

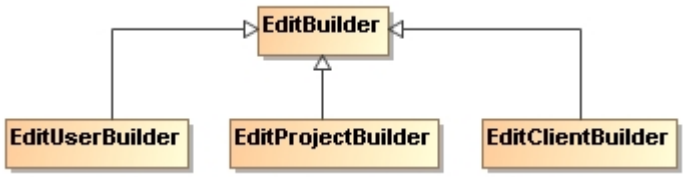

FIGURA 5.3 - CLASSES BUILDER DOS CONTROLOS DOS RECURSOS DO SISTEMA

A implementação da classe *EditBuilder* disponibiliza a lógica de gestão dos controlos gráficos para novos recursos vindos do servidor, exposição e edição dos detalhes e criação de novos recursos. As restantes operações, específicas de cada recurso, são implementadas pelas classes específicas. O mesmo acontece nas classes do tipo *controller*: existe uma classe *EditController*, da qual derivam a classe *UserController* e *ProjectController*. No entanto, não existe uma classe *ClientController* pois uma instância da classe *EditController* com o *builder* e *proxy* adequado tem a implementação de todas as operações necessárias.

A lógica dos controlos que permitem fazer a gestão de um tipo de recurso começa por fazer um pedido da lista dos recursos existentes no servidor. Os recursos podem depois ser visualizados em maior detalhe e editados (para mais informação, consultar o *Anexo E – Aplicação cliente)*. O modo de visualização em detalhe de um recurso permite a edição dos seus dados. Quando é

mostrada a informação detalhada de um recurso é feito um novo pedido ao serviço com a informação básica e a informação detalhada. A informação básica é novamente pedida pois, entretanto, os dados podem ter sido modificados por outro utilizador, o que faz com que a mesma informação possa ser pedida múltiplas vezes. De forma a actualizar facilmente os dados locais com a nova informação vinda do servidor, é utilizada a classe *DataCache*. A Listagem 5.7 mostra a sua interface.

```
1  class DataCache
2  {
3      Item insertItemNoRefCount(newItem);
4      Item insertItem(newItem);
5      void removeItem(item);
6      void clear();
7
8      /*abstract*/
9      ID _getCacheID(item);
10     Item _setUpCacheItem();
11     void _updateCacheItem(currentItem, newItem);
12 }
```

LISTAGEM 5.7 - CLASSE DATACACHE

A implementação da classe *DataCache* disponibiliza métodos para inserção e remoção de dados, onde é feito o armazenando de cada recurso pela sua chave. Se um recurso inserido já existir, os dados são actualizados em vez de serem inseridos. A classe *DataCache* é abstracta, tendo de existir uma classe específica por recurso, que implementa operações específicas ao recurso: a operação *_getCacheID*, que permite obter a chave única do objecto a inserir; a operação *_updateCacheItem*, que permite actualizar a informação de um objecto a partir de um novo objecto recebido; e a operação *_setUpCacheItem* cuja implementação é responsável pela preparação de um objecto para ser inserido na cache (e.g. adicionar novas propriedades necessárias para *data-binding*).

5.4.2 TIMESHEETS

A página de edição de uma *timesheet* é composta por dois controlos: o controlo de edição das *timesheets*, que permite a sua visualização detalhada e a sua edição, e o controlo que mostra a *timeline*, a qual mostra as *timesheets* disponíveis e indica se as suas horas de trabalho ou de férias foram submetidas para aprovação. Apesar de só ser suportada a edição da *timesheet*, não tendo ficado concluído o suporte para outras funcionalidades (e.g. submissão dos dados das *timesheets*), os controlos foram planeados para poderem dar suporte a essas funcionalidades. Assim, como as *timesheets* são partilhadas entre os controlos, é usado um objecto de cache onde é reunida a informação recebida do servidor, permitindo que as modificações feitas sobre as *timesheets* sejam reflectidas nos dois controlos.

Ambos os controlos usam o mesmo *proxy*, pois fazem pedidos ao serviço pelo mesmo tipo de recurso. Na Figura 5.4 é mostrada a arquitectura dos objectos que constituem os dois controlos.

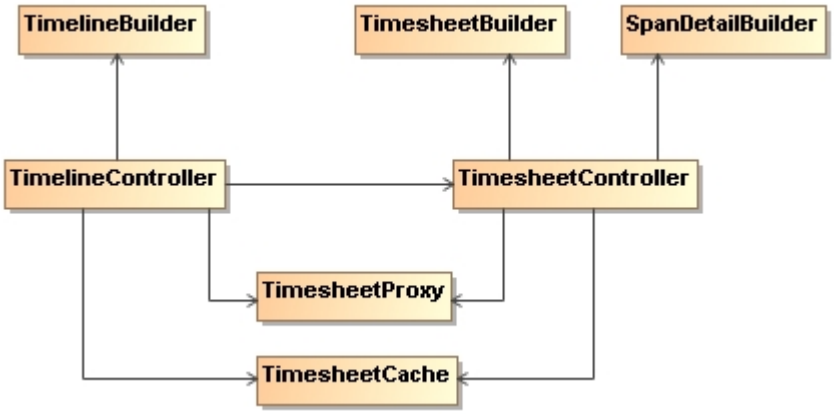

FIGURA 5.4 - RELAÇÃO ENTRE OS COMPONENTES DO CONTROLO DAS *TIMESHEETS*

As classes *TimelineController* e *TimesheetController* partilham o objecto *proxy* e o objecto da cache, que são necessários para a sua construção. A classe *TimesheetController* usa duas classes *builder*: uma para a gestão da edição da *timesheet,* chamada *TimesheetBuilder*, e outra para mostrar o detalhe das horas de trabalho de um dia seleccionado, chamada *SpanDetailBuilder*. Esta separação permite a simplificação da implementação dos controlos, separando as suas responsabilidades. A ligação entre as classes *builder*, ligando o evento de selecção de um dia ao método para mostrar os detalhes é feita pela implementação da classe *TimesheetController*.

5.4.3 CARREGAMENTO DE UMA TIMESHEET

O processo de edição de uma *timesheet* implica o pedido de dois conjuntos de dados: a informação da *timesheet* requisitada e a informação dos projectos associados ao utilizador. Os projectos associados ao utilizador são requisitados sempre que uma nova *timesheet* é carregada de modo a ser possível obter uma lista actualizada. O pedido dos dados de uma *timesheet* e da lista de projectos é efectuado em simultâneo e de forma assíncrona, sendo necessário um algoritmo para coordenar a recepção dos dados pedidos ao servidor. Esta lógica é mostrada na Figura 5.5.

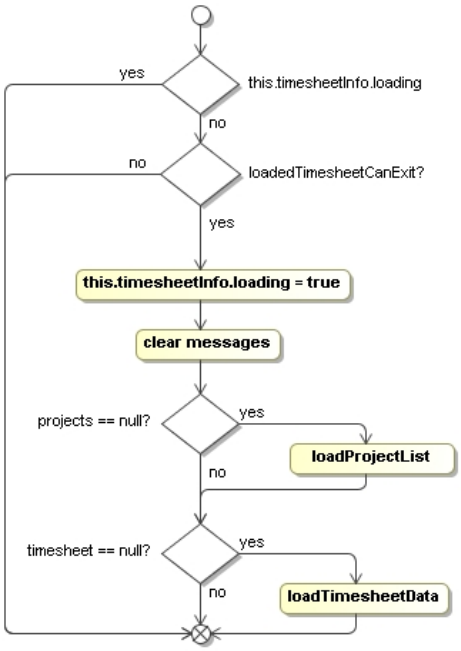

FIGURA 5.5 - PROCESSO DE CARREGAMENTO DE DADOS DE UMA *TIMESHEET*

A lógica do algoritmo impede o pedido simultâneo de várias *timesheets* e verifica o estado de uma *timesheet* que possa estar carregada para edição, através do algoritmo *loadedTimesheetCanExit* (explicado em detalhe no *Capitulo 5.4.4 - Edição de uma timesheet*). Se existirem condições para o carregamento da *timesheet* requisitada é marcado o estado de carregamento e são feitos dois pedidos simultâneo para obter a informação necessária para editar a *timesheet*: os dados da *timesheet* e a lista de projectos. O processamento da resposta a estes pedidos é mostrado na Figura 5.6.

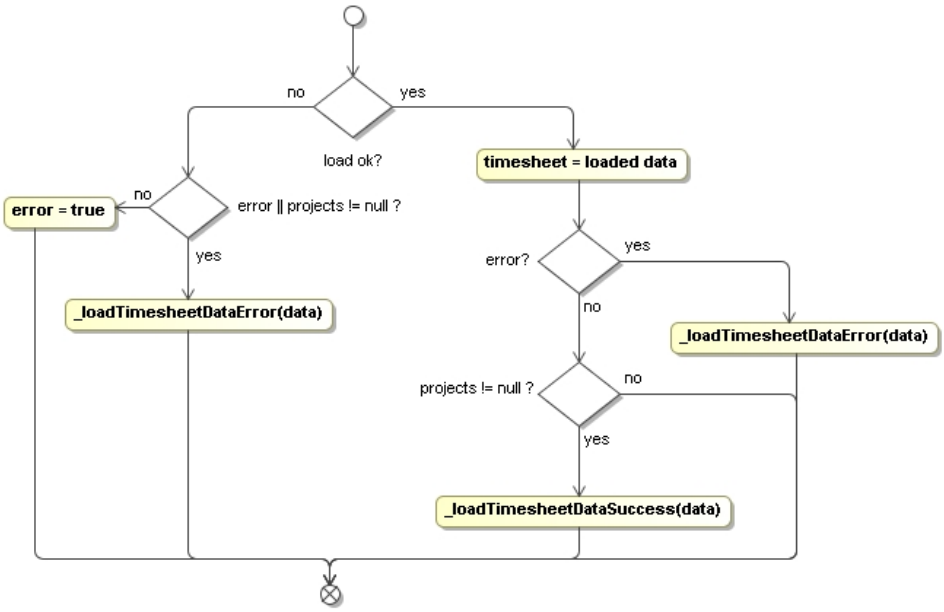

FIGURA 5.6 - PROCESSAMENTO DA RESPOSTA DO PEDIDO AOS DADOS DE UMA *TIMESHEET*

Apesar de não estar exposto na Figura 5.6, a resposta é processada através de funções de *callback*, conforme descrito no *Capítulo 5.3.1 - Classes Proxy*.

Quando um pedido é bem-sucedido, os dados obtidos são guardados, sendo verificada a *flag error,* que indica se o outro pedido que foi feito não foi bem-sucedido. Em caso afirmativo, é invocado o método de processamento de erro (i.e. *_loadTimesheetDataError*), a que são passados os dados obtidos. Caso contrário, é verificado se o outro pedido já foi concluído através dos dados que requisitou. Se os dados existirem, o outro pedido foi concluído e é invocado o método de processamento de sucesso (i.e. *_loadTimesheetDataSuccess*), passando os dados de ambos os pedidos. Se os dados não existirem, significa que o pedido ainda está a ser processado.

Quando um pedido não é bem-sucedido, é verificado se o outro pedido já foi processado, verificando se a *flag error* está sinalizada ou se os dados já existem localmente. Se o outro pedido já tiver sido processado é invocado o método de processamento de erro. Caso contrário, é sinalizada a *flag error* para o processamento do pedido não completado ser notificado que este pedido falhou.

O método *_loadTimesheetDataError*, invocado quando um dos pedidos falha, permite reaproveitar os dados obtidos de um pedido bem-sucedido. Como nos outros controlos, quando o pedido dos dados da *timesheet* não é bem-sucedido é mostrada uma mensagem de erro, que permite voltar a executar a operação de carregamento de dados. No entanto, quando a operação de carregamento de dados volta a ser executada é passada a informação que já foi pedida com sucesso que, como se pode verificar na Figura 5.5, faz com que não seja feito um novo pedido ao serviço por esses dados.

O método *_loadTimesheetDataSuccess,* invocado quando todos os pedidos sucedem, actualiza a interface gráfica e inicia a informação necessária para edição de uma *timesheet*.

5.4.4 EDIÇÃO DE UMA TIMESHEET

Cada *timesheet* permite a inserção e remoção de horas de trabalho, faltas ou férias sobre os seus dias, sendo a sua informação guardada no servidor enquanto é editada, até estar pronta a ser submetida. De forma a aumentar a usabilidade, um dos requisitos de negócio é a possibilidade da informação ir sendo progressivamente guardada no servidor como se fosse localmente preenchida.

A informação de uma *timesheet* é enviada em intervalos regulares através de um objecto *timer*, chamado de *updateTimer*, que é iniciado quando é feita uma modificação sobre a *timesheet*. A informação enviada para o serviço tem o diferencial entre os dados no servidor e as modificações executadas no cliente, sendo registadas na variável *this.updateObject*. As

modificações feitas sobre uma *timesheet* são processadas pelo algoritmo mostrado na Figura 5.7.

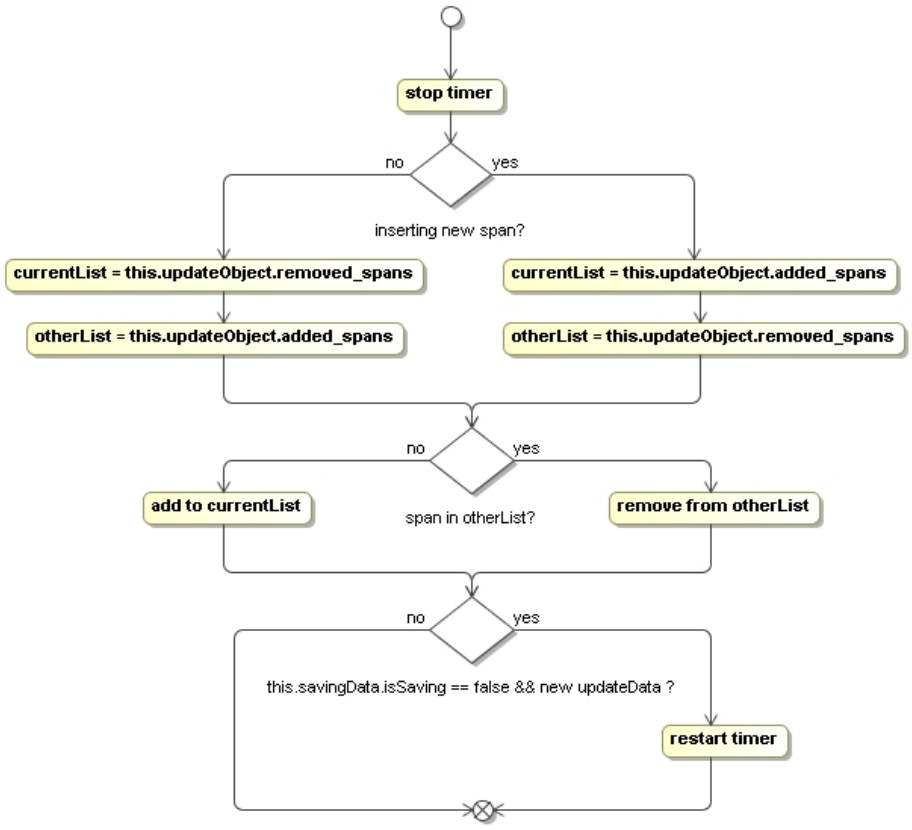

FIGURA 5.7 - ALGORITMO QUE REGISTA AS MODIFICAÇÕES DE UMA TIMESHEET

Sobre uma *timesheet* podem ser adicionadas ou removidas horas (i.e. *span*), que representam trabalho, férias ou faltas. Durante o registo de uma acção o objecto *timer* responsável por enviar as alterações para o serviço é parado. O algoritmo começa por identificar a lista correspondente à acção executada: se a acção for de adição, a variável *currentList* fica com uma referência para a lista de inserções feitas na *timesheet*; caso contrário fica uma referência para a lista de remoções. A lista que não corresponde à acção actual é referenciada pela variável *otherList*. De seguida é procurado por um *span* igual ao da acção executada na *otherList* que, se encontrado, indica que a acção actual está a anular uma acção anterior. Assim, se o *span* existir é removido da *otherList*, caso contrário a acção é registada adicionado o *span* à *currentList*. Por fim, caso não esteja a ser executado um pedido de actualização da *timesheet*, o objecto *updateTimer* é reiniciado se ainda existirem dados registados para actualização.

Os dados registados em *updateObject* são enviados periodicamente, e antes do utilizador deixar de editar a *timesheet*, através do método *_sendTimesheetUpdate*. O envio dos dados de forma periódica só ocorre se houver um período de tempo de inactividade na edição da *timesheet*. Desta forma, há uma maior probabilidade de os dados só serem enviados quando o

utilizador terminou a edição, reduzindo a quantidade de acções que se anulam entre si (e.g. remoção e adição do mesmo objecto *span*). O algoritmo, mostrado na Figura 5.8, faz a gestão do envio de dados e permite a edição da *timesheet* durante o envio dos dados.

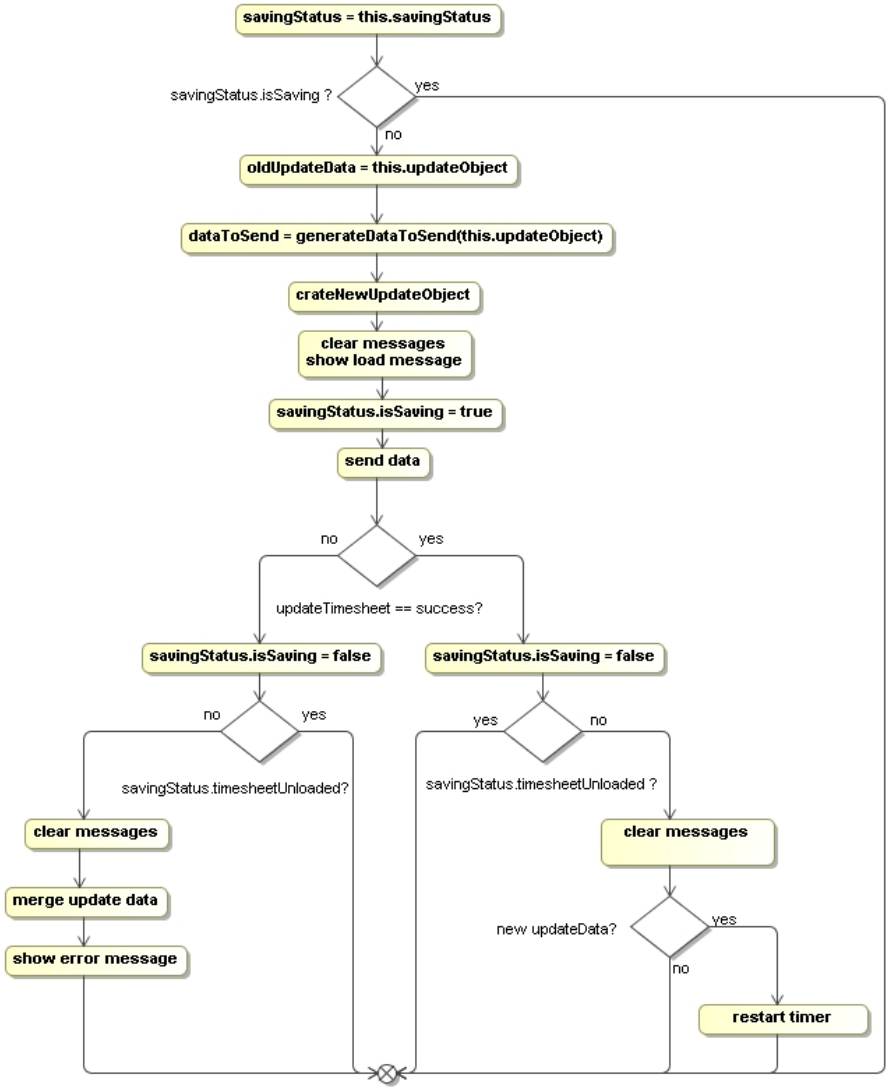

FIGURA 5.8 - ALGORITMO DE ENVIO DA ACTUALIZAÇÃO DE UMA TIMESHEET

Este algoritmo é utilizado em três situações: durante a edição de uma timesheet (periodicamente), por pedido de carregamento de uma nova *timesheet* ou na confirmação da saída da página da *timesheet*.

De forma a garantir que não ocorrem múltiplos pedidos quando uma *timesheet* está a ser guardada, é verificada e sinalizada a *flag isSaving* do objecto *savingStatus*. A referência para este objecto é mudada cada vez que uma nova *timesheet* é pedida, de forma ao envio de dados de *timesheets* diferentes não interferir entre si. Depois, é substituído o objecto *updateObject*, para poderem ser registadas novas modificações feitas à *timesheet* e os dados são enviados.

Apesar de não estar reflectido no diagrama, a resposta é processada de acordo com o padrão referido no *Capítulo 5.3.3 - Classes Controller*. A resposta ao pedido limpa a sinalização da *flag isSaving*. As acções seguintes só ocorrem caso a *timesheet* guardada ainda esteja em edição, verificando a *flag timesheetUnloaded* do objecto *savingStatus,* que é sinalizada no carregamento de uma nova *timesheet*. Caso o pedido tenha sido bem-sucedido e existam novos dados, o *updateTimer* é iniciado. Caso o pedido não tenha sido bem-sucedido, os dados a serem enviados são combinados com novos dados que tenham sido inseridos.

O algoritmo de carregamento de uma *timesheet* tem em conta a natureza assíncrona do processo de persistência de uma *timesheet* no serviço. Por este motivo, o método *loadedTimesheetCanExit* é usado como critério de carregamento de uma nova *timesheet,* cujo algoritmo é mostrado na Figura 5.9.

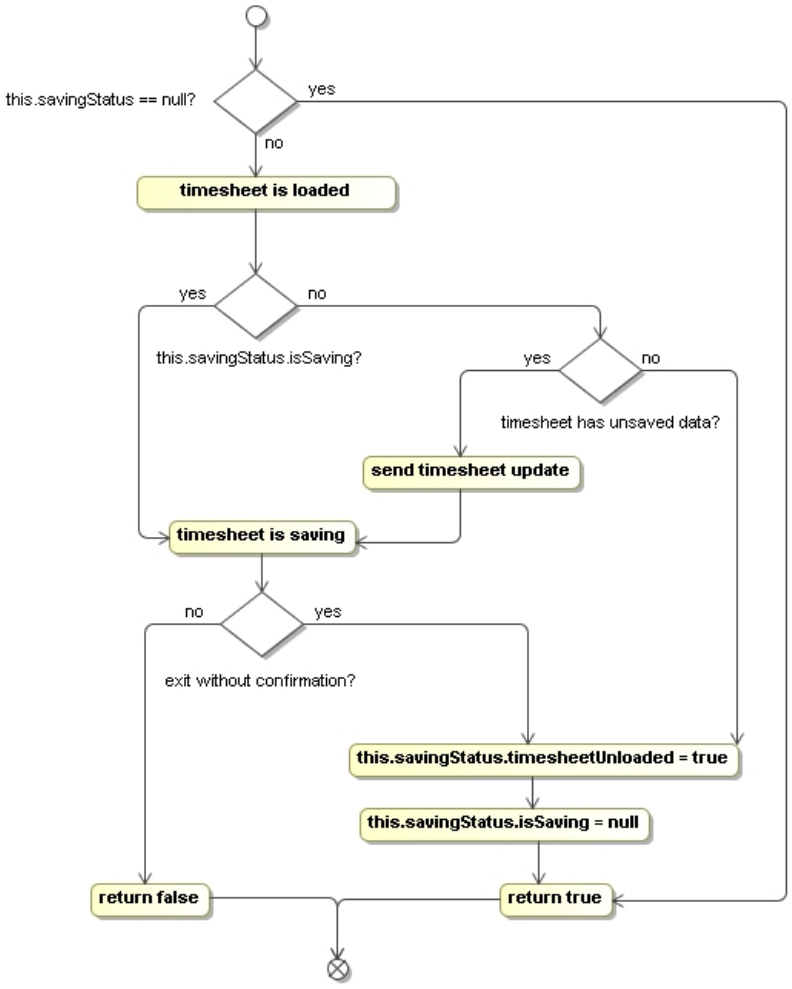

FIGURA 5.9 - PROCESSO DE DECISÃO DE CARREGAMENTO DE UMA TIMESHEET

O algoritmo começa por verificar o campo *savingStatus*, que só tem uma referência para um objecto quando existe uma *timesheet* em edição. Caso exista uma *timesheet* em edição, o seu estado é um, de três possíveis: os dados estão sincronizados com o serviço, está a decorrer

um pedido ainda não concluído para guardar seus os dados ou a *timesheet* tem informação não guardada no serviço. Se os dados estão sincronizados com o serviço é confirmado que a nova *timesheet* pode ser carregada. Caso exista informação não guardada e não esteja a ser feito nenhum pedido, é feito um pedido de actualização de dados ao serviço. Depois, é pedida confirmação ao utilizador se deve ser carregada uma nova *timesheet* sem haver confirmação do serviço que a *timesheet* em edição foi guardada. Caso a nova *timesheet* deva ser carregada para edição, é sinalizado no objecto *savingStatus* que a respectiva *timesheet* não está a ser editada e a referência para *savingStatus* é colocada com *null*.

Parte deste algoritmo está replicado no método *onShouldConfirmPageClose*, utilizado no evento de descarregamento da página das *timesheets,* sendo ligeiramente diferente, pois indica se deve ser confirmado o carregamento de uma *timesheet*. Isto acontece devido a uma limitação do *handler* de saída de uma página, que não permite mostrar janelas de confirmação ao utilizador, só recebe a indicação se deve pedir confirmação de saída.

5.5 TRABALHO FUTURO

A aplicação cliente dá suporte às funcionalidades implementadas no serviço, mas tem vários pontos identificados que podem ser melhorados. Estes consistem em mecanismos que são funcionais, mas que podem ser melhorados para serem mais informativos ou para terem um funcionamento mais complexo.

Os controlos gerados dinamicamente têm suporte para múltiplos idiomas, gerados a partir da informação do ficheiro *dictionary.js*. Como implementação futura, este ficheiro pode passar a ser gerado dinamicamente, por uma *JSP*, conforme a língua escolhida para mostrar informação. Da mesma forma, o conteúdo *HTML* gerado pela *JSP* tem de passar a inserir a informação conforme a língua escolhida.

A gestão dos erros dos pedidos feitos ao serviço também pode ser melhorada. A maioria dos controlos mostra uma mensagem de erro quando um pedido recebe uma resposta com código que não indica que foi bem-sucedido (i.e. código sem ser na gama de 2xx), dando a hipótese de voltar a executar o pedido. Existem situações de erro em que este não é o comportamento correcto, como quando se tentam inserir dados que já existem no servidor, ou no pedido de actualização de uma *timesheet,* que falha por ter sido adicionada uma hora num projecto a que o utilizador já não está associado. Estes erros devem fazer com que o utilizador seja informado do problema ocorrido.

Nos controlos gráficos é usado um mecanismo de *data-binding* cuja utilização faz com que tenham de ser acrescentados campos aos objectos recebidos do serviço. A gestão destes objectos é feita pela componente *builder*, que acrescenta os campos necessários aos objectos e que os remove quando os objectos são enviados para serviço. Na implementação actual, a

transformação destes campos é específica de cada objecto *builder*, não sendo consistente. Existe também um caso, na visualização de *timesheets*, em que dois *builders* usam os mesmos dados. Assim, é possível refactorizar o código de forma a que a transformação dos dados a serem usados para *data-binding* seja feita por uma classe especializada, criando os campos necessários e preparando os dados para serem enviados para o serviço. Esta classe acrescentaria também um mecanismo de transparência aos controlos, pois não teriam de ser modificados se o formato dos dados vindos do serviço mudasse.

Um outro ponto futuro a incluir na aplicação são testes unitários em *javascript* na aplicação cliente. Estes testes permitiram desenvolver novas funcionalidades com maior robustez, dado que esta aplicação tem um grau de complexidade elevado.

A compatibilidade entre os *browsers* é outro ponto a implementar no trabalho futuro. De momento é totalmente funcional nos *browsers Google Chrome* e *Mozzila Firefox*. A utilização do browser *Opera* gera mensagens de erro em alguns pedidos que são concluídos com sucesso. Não existe suporte para *Internet Explorer 9*, por não ser suportado pela versão da biblioteca *JQuery* usada. A aplicação foi também testada no *iPad*, através do browser *Safari*, mas não conseguiu iniciar nenhuma comunicação com o serviço, provavelmente devido à versão do *JQuery* não ser compatível.

Capítulo 6

NOTAS FINAIS

6.1 DESAFIOS FUTUROS

Durante o projecto começou a ser planeado o próximo conjunto de funcionalidades que não chegou a ser implementado. No entanto, o planeamento inicial tem umas ideias que integram com as funcionalidades já implementadas, as quais são expostas de seguida.

A funcionalidade de gestão de utilizadores permite definir uma hierarquia de utilizadores. Um utilizador com a função de gestor pode gerir vários consultores, tendo cada consultor um gestor associado. Esta funcionalidade tem como objectivo permitir que um gestor visualize informação submetida pelo consultor e validar as suas acções. O gestor pode ter simultaneamente a função de consultor podendo, nesse caso, ter outro gestor associado ou ser o seu próprio gestor.

O principal desafio na definição da hierarquia de utilizadores está no formato de dados que permite definir a cadeia de validação. No preenchimento da estrutura deve ser feita uma transacção para garantir que os utilizadores envolvidos (e.g. gestor e consultor) existem. No entanto, a granularidade de cada *entity group* está atribuída por utilizador, e não por empresa, de forma a melhorar a concorrência do sistema, devido ao limite de 5 escritas por segundo no sistema de dados de replicação usado. De modo a manter a granularidade actual foi pensada uma abordagem que consiste em ter uma referência no consultor para o seu gestor e no gestor para os seus consultores. A edição seria feita sobre uma das entidades, dentro de uma transacção, onde também é colocada uma tarefa numa *task queue* com o objectivo de actualizar a outra entidade. Esta tarefa seria depois processada, tentando actualizar a outra entidade ou gerando uma nova mensagem, caso a entidade tivesse sido removida. Esta abordagem precisa de uma análise mais detalhada para definir uma estratégia que garanta o

processamento ordenado das mensagens pois, apesar das *task queues* terem uma filosofia *FIFO,* podem mudar a ordem das mensagens de forma a diminuir a latência.

A implementação de um sistema de regras viabiliza a validação automática de recursos submetidos para aprovação. As regras poderiam ser registadas de forma a validar um tipo de recurso pertencente a um utilizador com determinada função. Estas regras seriam registadas para serem usadas no momento de validação, podendo ser globais, ou seja, validadas antes de o recurso ser submetido para aprovação ou no momento de validação de determinado utilizador, permitindo que cada utilizador possa ter regras especificas associadas. As regras teriam dados associados, registando os dados aprovados de forma a manterem histórico para validação de dados futuros. Na submissão das horas de trabalho de uma *timesheet*, por exemplo, poderiam ser registadas globalmente um limite de faltas por trabalhador, e por gestor poderia ser registada uma regra que alerta para a existência de utilizadores em excesso de horas de trabalho. O processo de validação de regras permitiria notificar o utilizador e/ou fazer com que o processo de validação falhe.

O sistema de validação de acções teve um planeamento detalhado. Através desta funcionalidade é possível definir uma cadeia de utilizadores para validar acções do sistema, validando as regras existentes. A submissão das horas de trabalho de uma *timesheet*, por exemplo, pode ser traduzida numa acção que tem de ser validada por uma cadeia de utilizadores, o utilizador com a função de gestor e o utilizador com a função de recursos humanos, ambos associados ao consultor. O funcionamento a implementar para o processo de submissão é mostrado na Figura 6.1.

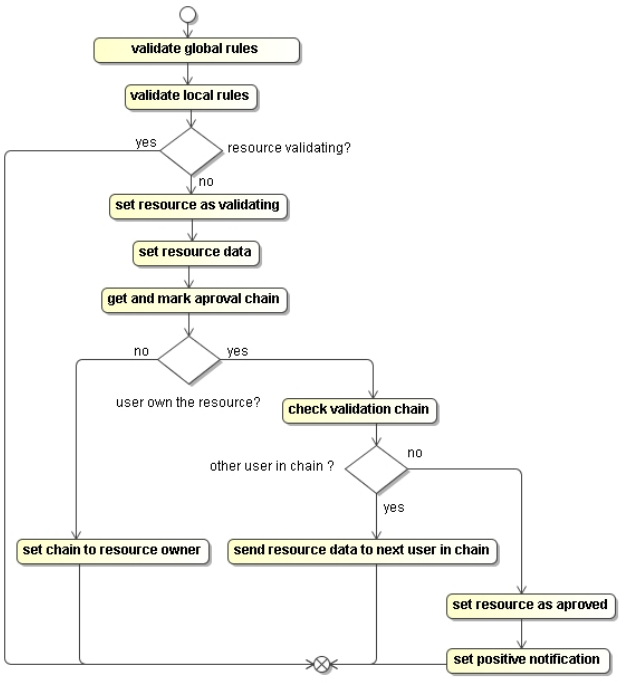

FIGURA 6.1 - PROCESSO DE SUBMISSÃO DE UM RECURSO PARA APROVAÇÃO

O processo de submissão de um recurso começa por validar as regras globais e locais, que estejam registadas, e marca-o como em validação, não permitindo a sua edição até a validação estar concluída. Os dados necessários para a validação são depois inseridos e a cadeia de aprovação a ser utilizada é registada. Este passo é necessário para ser usada a mesma cadeia de aprovação durante o processo, uma vez que pode ser editada externamente. O utilizador que submete os dados para aprovação não tem de ser o utilizador associado ao recurso; uma *timesheet*, por exemplo, pode ser editada e submetida por um gestor. Se a submissão for feita pelo utilizador associado ao recurso, o próximo utilizador da cadeia é notificado para continuar o processo de aprovação; caso contrário o utilizador associado ao recurso é notificado, começando o processo de aprovação nesse utilizador. Neste caso, apesar de o utilizador que submeteu os dados estar na cadeia de aprovação, não vai ser mais notificado até o processo terminar. As acções que não têm uma cadeia de validação definida são automaticamente aprovadas, havendo necessidade de um sistema de controlo de versões nesses recursos.

Como cada utilizador tem um entity *group* associado, o processo de validação recorre ao mecanismo de *task queue* para executar a tarefa de validação sobre o próximo utilizador da cadeia. A Figura 6.2 mostra o processo de validação de um recurso.

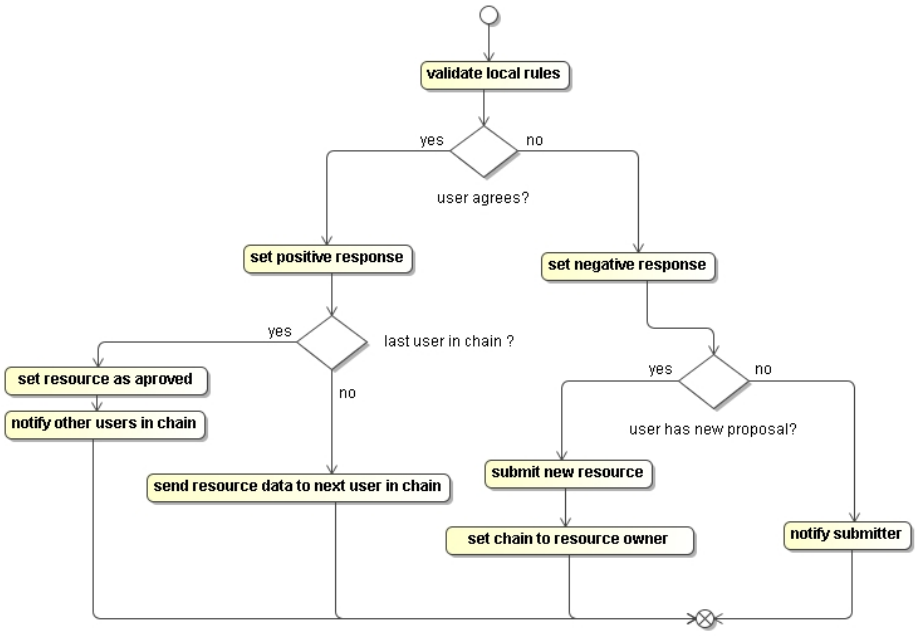

FIGURA 6.2 - PROCESSO DE VALIDAÇÃO DE UM RECURSO

Quando é recebida uma notificação para aprovação de um recurso, são validadas as regras locais e é registado o parecer do utilizador. Quando é registado um parecer negativo, o utilizador pode propor uma alteração que é enviada para o dono do recurso, sendo o processo de validação reiniciado. Caso contrário, o processo termina e os restantes utilizadores são notificados. Se o utilizador registar um parecer positivo e existirem mais utilizadores para fazer a validação, é notificado o próximo utilizador. No caso de ser o último utilizador na cadeia de

validação, o recurso é marcado como aprovado e são notificados os outros utilizadores. O utilizador a quem pertence o recurso é responsável por actualizar os dados com base no resultado do processo de validação. A Figura 6.3 tem a estrutura do algoritmo que notifica os utilizadores que um recurso foi aprovado.

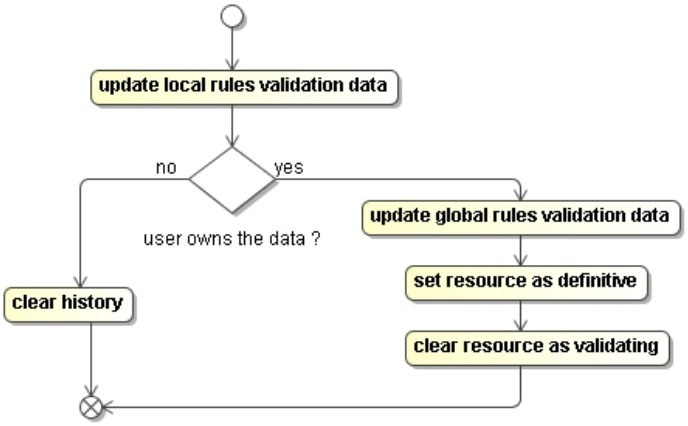

FIGURA 6.3 - ALGORITMO DE APROVAÇÃO DE UM RECURSO

Quando uma notificação de aprovação de recurso é recebida, o algoritmo das regras de cada utilizador faz o registo da nova informação necessária para validações futuras. Quando os dados aprovados estão associados a um utilizador, é ainda feito o registo de informação para as regras globais, é actualizado o recurso com base na informação aprovada e é mudado o seu estado de validação para que se lhe possam fazer novas edições.

O próximo passo deste projecto consistiria na implementação da hierarquia de utilizadores e do algoritmo de aprovação de acções, ficando o sistema de regras para mais tarde. Esta área ficou pela fase de planeamento, estando também identificada como trabalho futuro.

6.2 CONCLUSÕES

O presente projecto foi iniciado com o objectivo de implementar uma plataforma de serviços para gestão e contabilização de tempo remunerável. Para concretizar este objectivo foi feito um levantamento de requisitos funcionais, que foram traduzidos em funcionalidades, e uma análise do formato e das plataformas mais adequadas para a sua implementação, tendo sido escolhido o *Google App Engine*.

Para a implementação, foi necessário estudar o funcionamento dos serviços da plataforma e analisar as tecnologias que podem ser utilizadas, tendo em conta as restrições da plataforma. Além do cuidado inerente à escolha das tecnologias, a dificuldade foi acrescida por existirem operações que funcionam no simulador local mas que não funcionam quando executadas na *cloud*. Esta fase teve como objectivo definir uma arquitectura que fornecesse suporte à implementação de funcionalidades através de um serviço. A arquitectura foi estruturada em

camadas, de forma a separar o processamento do serviço, a lógica de negócio e lógica de acesso a dados.

Para a camada de processamento do serviço foram estudados mecanismos para recepção de pedidos, tendo sido disponibilizada uma interface *REST*. Nesta camada está implementada a lógica de autenticação do utilizador e de autorização de pedidos, é feita a validação da informação recebida e efectuado o tratamento de erros de forma centralizada. A biblioteca usada para implementar o serviço permite adicionar novos *endpoints* e definir dados a enviar, em formato *JSON* ou *XML*, de uma forma flexível. Além disso, os tipos de resposta enviados pelas operações do serviço têm um comportamento coerente e bem definido. As diferentes camadas, a plataforma, a lógica de negócio e a lógica de acesso a dados estão separadas, havendo pouco acoplamento e sendo fáceis de mudar.

A implementação da camada de dados implicou o estudo de um novo formato de armazenamento de dados, influenciando a estrutura e complexidade das funcionalidades implementadas. A utilização do serviço *datastore* acabou por ser ainda mais desafiante do que o esperado, devido às limitações das bibliotecas escolhidas, estando identificada como trabalho futuro a utilização de um biblioteca diferente. Na camada de dados foi ainda adicionado suporte para transacções aninhadas, permitindo que as transacções iniciadas nas operações disponibilizadas possam ser incluídas em transacções externas, iniciadas pela camada de negócio. Esta estratégia permitiu delegar responsabilidade transaccional para a camada de negócio, garantindo que as transacções só são iniciadas quando necessário.

Todas as operações implementadas pela camada de negócio estão testadas através de testes unitários, garantindo a qualidade do código e facilitando a implementação de novas funcionalidades. As operações do serviço são validadas pela aplicação cliente.

O desenvolvimento da aplicação cliente passou por várias fases. O seu formato inicial tinha como objectivo fazer uma validação rápida do serviço, testando as operações *REST*. Com o evoluir do serviço a aplicação cliente foi evoluindo também, passando a ter um funcionamento e um aspecto gráfico mais complexo: a manipulação de recursos passou a ser feita por controlos, que estavam divididos em componentes, havendo uma componente para comunicação, uma para implementar a lógica e uma para tratamento gráfico. A utilização de novas bibliotecas foi introduzida conforme a necessidade de melhorar o processo de implementação das funcionalidades implementadas foi surgindo. Por este motivo, existem componentes que tiveram uma implementação progressiva, não reflectindo a utilização das tecnologias disponíveis. No entanto, sempre que se justificou, o código foi alterado para utilizar as novas bibliotecas, como é o caso do uso do mecanismo de *data binding*.

O projecto passou por várias fases, tendo cada uma os seus desafios que foram ultrapassados através do estudo de soluções. Apesar das dificuldades, que fez com que as funcionalidades

inicialmente propostas não tenham sido implementadas, foi possível construir uma solução que cumpre a base da proposta e que disponibiliza uma arquitectura que viabiliza e auxilia a implementação das restantes funcionalidades.

REFERÊNCIAS

1. **Timereporting**, 03 dez. 2010. Disponivel em: <http://www.timereporting.com/>.

2. **Toggl**, 03 dez. 2010. Disponivel em: <http://www.toggl.com/public/tour>.

3. **Cube**, 03 dez. 2010. Disponivel em: <http://cube.bitrzr.com/>.

4. **Google App Engine**, 12 jan. 2011. Disponivel em: <http://code.google.com/appengine/>.

5. **Windows Azure**, 12 jan. 2011. Disponivel em: <http://www.microsoft.com/windowsazure/windowsazure/>.

6. **Amazon Web Services**, 12 jan. 2011. Disponivel em: <http://aws.amazon.com/>.

7. **Joyent**, 12 jan. 2011. Disponivel em: <http://www.joyent.com/>.

8. **VmWare vFabric cloud application platform**, 12 jan. 2011. Disponivel em: <http://www.springsource.com/products/cloud-application-platform>.

9. **Python in GAE**. Disponivel em: <http://code.google.com/appengine/docs/python/gettingstarted/>. Acesso em: 05 set. 2011.

10. **Java in GAE**. Disponivel em: <http://code.google.com/appengine/docs/java/gettingstarted/>. Acesso em: 05 set. 2011.

11. **Windows Azure Interoperability**. Disponivel em: <http://www.microsoft.com/windowsazure/interop/>. Acesso em: 25 set. 2011.

12. **Java on Windows Azure**. Disponivel em: <http://msdn.microsoft.com/en-us/windowsazure/ee941631>. Acesso em: 25 set. 2011.

13. **AllThingsDistributed - Eventual consistent**. Disponivel em: <http://www.allthingsdistributed.com/2008/12/eventually_consistent.html>. Acesso em: 25 set. 2011.

14. **Linked-In**. Disponivel em: <http://pt.linkedin.com/>. Acesso em: 05 set. 2011.

15. **THQ**. Disponivel em: <http://www.thq.com/uk>. Acesso em: 05 set. 2011.

16. **Kabam**. Disponivel em: <http://www.kabam.com/>. Acesso em: 05 set. 2011.

17. **Spring**. Disponivel em: <http://www.springsource.org/>. Acesso em: 05 set. 2011.

18. **Scrum**, 26 jan. 2011. Disponivel em: <http://scrummethodology.com/>.

19. **GIT**. Disponivel em: <http://git-scm.com/>. Acesso em: 14 jun. 2011.

20. **Unfuddle**. Disponivel em: <http://unfuddle.com/>. Acesso em: 14 jun. 2011.

21. **Why we switched from Google App Engine to EC2**, 14 jun. 2011. Disponivel em: <http://activelearningblog.com/2010/10/why-we-switched-from-google-app-engine-to-ec2/>.

22. **Amazon's Trouble Raises Cloud Computing Doubts**. Disponivel em: <http://www.nytimes.com/2011/04/23/technology/23cloud.html?_r=1>. Acesso em: 14 jun. 2011.

23. **Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region**. Disponivel em: <http://aws.amazon.com/message/65648/>. Acesso em: 14 jun. 2011.

24 BURKE, B. **RESTFul Java with JAX-RS**. 1ª edição. ed. [S.l.]: O'Reilly, 2010.

25 RICHARDSON, L.; RUBY, S. **RESTFull Web Services**. [S.l.]: O'Reilly, 2007.

26. **How the Servlet Container Invokes Filters**. Disponivel em: <http://download.oracle.com/docs/cd/B14099_19/web.1012/b14017/filters.htm#i1000654>. Acesso em: 15 set. 2011.

27. **The JRE Class White List**. Disponivel em: <http://code.google.com/appengine/docs/java/jrewhitelist.html>. Acesso em: 14 jun. 2011.

28. **Will I tPlay In Java**. Disponivel em: <http://code.google.com/p/googleappengine/wiki/WillItPlayInJava>. Acesso em: 14 jun. 2011.

29. **The Users Java API**. Disponivel em: <http://code.google.com/appengine/docs/java/users/>. Acesso em: 14 jun. 2011.

30 MULARIEN, P. **Spring Security 3**. [S.l.]: PACKT Publishing, 2010.

31. **Spring**. Disponivel em: <http://static.springsource.org/spring/docs/1.2.x/reference/beans.html>. Acesso em: 14 jun. 2011.

32. **Spring Security**. Disponivel em: <http://static.springsource.org/spring-security/site/docs/3.1.x/reference/technical-overview.html>. Acesso em: 14 jun. 2011.

33. **Package com.google.appengine.api.datastore**. Disponivel em: <http://code.google.com/appengine/docs/java/javadoc/com/google/appengine/api/datastore/package-summary.html>. Acesso em: 14 jun. 2011.

34. **GAE - Exemplos com a low level API**. Disponivel em: <http://code.google.com/appengine/docs/java/datastore/>. Acesso em: 14 jun. 2011.

35. **JDO**. Disponivel em: <http://code.google.com/appengine/docs/java/datastore/jdo/>. Acesso em: 14 jun. 2011.

36. **JPA**. Disponivel em: <http://code.google.com/appengine/docs/java/datastore/jpa/>. Acesso em: 14 jun. 2011.

37. **Objectify**, 14 jun. 2011. Disponivel em: <http://code.google.com/p/objectify-appengine/>.

38. **TWiG Persist**. Disponivel em: <http://code.google.com/p/twig-persist/>. Acesso em: 14 jun. 2011.

39. **Slim 3**. Disponivel em: <http://code.google.com/p/slim3/>. Acesso em: 14 jun. 2011.

40. **Paxos Algorithm**. Disponivel em: <http://labs.google.com/papers/paxos_made_live.html>. Acesso em: 15 jan. 2011.

41. **Datastore transactions**. Disponivel em: <http://code.google.com/appengine/docs/java/datastore/transactions.html#Using_Transactions>. Acesso em: 10 set. 2011.

42. **TWiG Persist - Google Groups**. Disponivel em: <http://groups.google.com/group/twig-persist?pli=1>. Acesso em: 14 jun. 2011.

43. **Guice**. Disponivel em: <http://code.google.com/p/google-guice/>. Acesso em: 14 jun. 2011.

44 PRASANNA, D. R. **Dependency Injection**. [S.l.]: Manning, 2009.

45 VANBRABANT, R. **Google Guice**. [S.l.]: Apress.

46. **Google App Engine - Enabling Sessions**. Disponivel em: <http://code.google.com/appengine/docs/java/config/appconfig.html#Enabling_Sessions>. Acesso em: 14 jun. 2011.

47 CIURANA, E. **Developing with Google App Engine**. [S.l.]: Apress, 2008.

48 BIBEAULT, B.; KATZ, Y. **jQuery In Action**. 2ª edição. ed. [S.l.]: Manning, 2010.

49. **Jquery-UI**. Disponivel em: <http://jqueryui.com/>. Acesso em: 17 jun. 2011.

50. **JSP Templates**. Disponivel em: <http://java.sun.com/developer/technicalArticles/javaserverpages/jsp_templates/>. Acesso em: 17 jun. 2011.

51. **Wikinvest**, 03 dez. 2010. Disponivel em: <http://www.wikinvest.com/concept/Software_as_a_Service>.

52. **TestNG**. Disponivel em: <http://testng.org/doc/index.html>. Acesso em: 14 jun. 2011.

53. **JSP Tags**. Disponivel em: <http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPTags.html>. Acesso em: 05 set. 2011.

54. **HTTP Codes**. Disponivel em: <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>. Acesso em: 20 set. 2011.

55. **Knockout JS**. Disponivel em: <http://knockoutjs.com/>. Acesso em: 05 set. 2011.

GLOSSÁRIO

Termos	Definições
AJAX	Asynchronous JavaScript and XML – conjunto de tecnologias usadas para comunicações assíncronas entre aplicações web (client-side)
AMI	Amazon Machine Instance
API	Application programming interface
Cloud computing	Modelo de utilização da internet e de um conjunto de servidores remotos que fornecem serviços de hospedagem de aplicações e de armazenamento de dados
Commit	Palavra-chave usada numa transacção para indicar que a transacção terminou com sucesso
CRUD	Create, Read, Update and Delete – Conjunto de operações normalmente existentes em armazenamento de dados
CSS	Cascading Style Sheets
DAL	Data Access Layer
DAO	Data Access Object
DOM	Domain Object Model - Convenção para representação de objectos HTML
FIFO	First In First Out - Descreve o comportamento de uma fila quando são inseridos e removidos dados
GUI	Graphical User Interface - Aspecto gráfico de uma aplicação
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol – implementação do protocolo http
HTTPS	Protocolo HTTP que usa uma camada de segurança para proteger os dados transmitidos
IDE	Integrated Development Environment - Aplicação que facilita o desenvolvimento de software, fornecendo ferramentas como destaque de instruções de código especiais
JAXB	Java Architecture for XML Binding – Permite mapear objectos de dados para XML, para seriação ou deseriação
JAX-RS	Java API for RESTful Web Services – API que permite mapear classes Java para webservices REST
JDO	Java Data Objects – modelo Java, standard, para abstracção do modelo de persistência usado
JPA	Java Persistence API – framework para manipulação de bases de dados relacionais
JSON	JavaScript Object Notation
JSP	Java Server Pages
KO	Refere a biblioteca Knockout.js
MVC	Padrão de desenho Model-View-Controller
POJO	Plain Old Java Objects – Notação usada para referir qualquer objecto java
REST	Representational State Transfer

Termos	Definições
RESTFul	Serviço web que implementa os princípios Rest
Role	Papel / função associado a um utilizador, que lhe dá permissão para execução determinada operação
Rollback	Palavra-chave usada numa transacção para indicar que a transacção não sucedeu
SaaS	Software As a Service – modelo de negócio cuja aplicação não reside na máquina de um cliente mas na entidade vendedora, acedida através de uma ligação à internet
schema	Linguagem que descreve um formato de dados rígido, normalmente usada em bases de dados ou xml
Servlet	Modelo de programação Java usado para estender as capacidades de servidores web
SGBD	Sistema de gestão de base de dados
Single Sign-on	Ponto de entrada único, comum entre várias aplicações, onde um utilizador se autentica
SOAP	Simple Object Access Protocol – especificação para troca de informação estruturada entre serviços web, através de XML
Span	Conjunto de horas de trabalho associadas a um dia de uma timesheet
TCP	Protocolo de transmissão de dados
Timesheet	Folha marcação de horas de trabalho, mensal
URI	Uniform Resource Identifier – Especifica o nome de um recurso
URL	Uniform Resource Locator – Especifica a localização de um recurso e o mecanismo usado para o obter
WADL	Web Application Description Language – Formato XML usado em Web Services, tipicamente REST, que descreve o formato das operações. Nem todos os serviços REST disponibilizam
Workflow	Processo de aprovação de uma operação, como aprovar uma timesheet
WSDL	Web Services Description Language – Formato XML para descrição de web services SOAP. Normalmente disponibilizado devido à complexidade do formato das operações destes serviços
XML	Extensible Markup Language – Conjunto de regras para definição de documentos

Anexo A

COMPARAÇÃO DE PLATAFORMAS CLOUD

Este anexo foi criado com o objectivo de fornecer um meio de comparação entre as três principais plataformas *cloud* estudadas, de forma a concluir qual a melhor plataforma para implementar o projecto proposto. Para as conclusões, foram analisadas duas vertentes: as características das máquinas onde a aplicação é executada e os serviços oferecidos. A comparação é mostrada na Tabela A-1 e na Tabela A-2, respectivamente.

A.1 EXECUÇÃO DA APLICAÇÃO

	Google App Engine	***Windows Azure***	***Amazon Web Services***
Características da máquina	Não se aplica.	Máquina virtualizada, capacidade varia em 4 configurações.	Máquinas virtuais, capacidade comprada por unidades de computação (*EC2U*).
Formato	Aplicação em *Java* (*Java 6*) ou *Python* (*Python 2.5*).	Um role por instância (*Web Role* ou *Worker Role)*	Imagem com sistema operativo e aplicações.
Tecnologias	API autorizadas no ambiente de execução.	*C#, Visual Basic, C++* ou *Java*; *Visual Studio, ASP.NET, WCF, ADO.NET PHP, Oracle*.	Qualquer tecnologia.
Escalabilidade	Distribuição automática por vários servidores.	Instâncias adicionadas dinamicamente; permite definir zonas geográficas e grupos de dados que devem estar no mesmo *data center.*	Instâncias adicionadas dinamicamente; permite definir zonas geográficas.

	Google App Engine	***Windows Azure***	***Amazon Web Services***
Fiabilidade	Dados sempre replicados.	Dados sempre replicados 3 vezes.	Dados sempre replicados.
Publicação	*Staging* e publicação quando pronta, mantendo o endereço.	*Staging* e publicação quando pronta, mantendo o endereço.	Manual.
Monitoração	*API appstats* identifica pedidos redundantes (*datastore* e mecanismo de *caching)*.	Sistema *Fabric* recolhe informação de performance, carga do CPU e *crash dumps*.	*Cloudwatch*.recolhe informações sobre o processamento, acessos de I/O e acessos à rede.
Teste local	Servidor local simula o GAE e os serviços.	Simulação local da aplicação com as funcionalidades da *cloud*.	Não tem devido à filosofia do serviço.

TABELA A-1 – COMPARAÇÃO DO FORMATO DAS PLATAFORMAS CLOUD

	Google App Engine	***Windows Azure***	***Amazon Web Services***
Persistência	*Blobstore*, Binário, cada entrada pode ter 2 GB; Datastore, armazenamento estruturado	*Binary large objects (blobs)*, cada entrada pode ter 1 TB; *Table, armazenamento estruturado.* Podem ser acedidos de com *ADO.NET Data services* ou *Linq*.	*Simple Storage Service, O*bjectos de dados até 5GB; *Amazon Elastic Block Storage, volumes entre 1GB e 1 TB;* *Amazon Simple DB,* armazenamento estruturado
Suporte de SQL	*Hosted SQL*;	*SQL Azure;*	*Relational Database Service* Instalação de SGBD em máquina EC2.
Comunicação Assíncrona	*Task Queues, tarefas* eventualmente executadas; *Scheduled Tasks, tarefas* executadas em intervalos regulares	*Queue, fila de mensagens.*	*Simple Queue Service,* filas de mensagens.
Acesso Autenticação	*APIs* de autenticação através do *Google Accounts.*	*AppFabric:* *- Service Bus, para* disponibilização de *endpoints* *- Access Control* para identificação e permissões de acesso.	*AWS Identity and Access Management*, gestão de permissões

	Google App Engine	***Windows Azure***	***Amazon Web Services***
API	*Bulk loader tool*; *Memcahe;* Envio de *email*; Manipulação de imagens; *URLFetch;* XMPP *para stream* e ligações longas.	Fornecidas por tecnologias windows	*Elastic Map Reduce*; *CloudFront*.

TABELA A-2 - SERVIÇOS DAS PLATAFORMAS CLOUD

Anexo B

UTILIZAÇÃO DA TECNOLOGIA JERSEY

A biblioteca *Jersey* é usada através de uma *servlet*. A sua integração no *Servlet Container* é feita através do ficheiro *web.xml*, sendo mostrada na Listagem B.1.

```
1  <servlet>
2      <servlet-name>Timecloud WS</servlet-name>
3      <servlet-class>
4          com.sun.jersey.spi.container.servlet.ServletContainer
5      </servlet-class>
6      ...
7      <init-param>
8          <param-name>com.sun.jersey.config.property.packages</param-name>
9          <param-value>weblayer.endpoints</param-value>
10     </init-param>
11 </servlet>
12
13 <servlet-mapping>
14     <servlet-name>Timecloud WS</servlet-name>
15     <url-pattern>ws/*</url-pattern>
16 </servlet-mapping>
```

LISTAGEM B.1 - INTEGRAÇÃO DE JERSEY NO SERVLETCONTAINER

A configuração apresentada indica que deve ser usada a *Servlet* Jersey. O parâmetro de iniciação (i.e. *param-value,* na linha 9) indica o *package* onde estão as classes que vão ser usadas pelo *Servlet*, que contém a definição dos endpoints ou outro objecto tipo auxiliar para transformação de dados. No elemento *servlet-mapping* é indicada a base dos *URL* processados, sendo neste caso começados por *ws/*.

Na Listagem B.2 é mostrado um exemplo da implementação de um *endpoint* que utiliza a tecnologia.

```
1  @Path("/")
2  public class Resources {
3      @GET
4      @Path("helloworld")
5      @Produces("text/plain")
6      public String getHelloWorldMessage() {
7          return "Hello world";
8      }
9  }
```

LISTAGEM B.2 - IMPLEMENTAÇÃO DE UM ENDPOINT USANDO JERSEY

Cada *endpoint* é definido através da anotação *@Path*, que indica o *URL* associado. A anotação tem de ser colocada no método que processa o *endpoint* e pode também ser colocada na classe de forma a indicar a base do *URL* para todos os métodos. Também deve ser usada uma anotação que indica o verbo *HTTP* (e.g. *@GET*) que o método processa e o tipo de resposta produzida (i.e. *@Produces*). Com esta configuração o método *getHelloWorldMessage* será invocado quando for feito um pedido *HTTP* com o verbo *Get* ao *URL ws/helloworld* que requisite uma resposta em texto. Todos os pedidos no serviço seguem esta lógica.

No presente projecto os pedidos com necessidade de resposta *XML* ou *JSON* utilizam a tecnologia *JAXB*, suportada pela biblioteca *Jersey*. As respostas que necessitem de enviar informação utilizam um objecto com anotações *JAXB,* sendo gerada a resposta adequada *a*o formato pedido.

Anexo C

CONCEITOS DO SERVIÇO DATASTORE

O serviço *Datastore* permite armazenar objectos de dados, chamados de entidades. Uma entidade pode ter várias propriedades, armazenadas por pares com o formato nome – valor. Cada entidade é identificada por uma chave única. A chave é constituída por três partes: um tipo (i.e. *String*), um identificador (i.e. nome indicado ou valor numérico gerado pelo *datastore*) e, opcionalmente, uma outra chave que identifica o *parent* da entidade.

Cada entidade pertence a um conjunto de entidades chamado *entity group*. Os *entity groups* são o critério usado pelo *datastore* para guardar entidades no mesmo nó (i.e. *node*). Se na chave da entidade for indicada a chave de outra entidade, o seu *parent*, o serviço *datastore* coloca-os no mesmo *entity group*. Caso contrário, a entidade é a *root* (i.e. base) do seu *entity group*. O *entity group* de uma entidade não pode ser mudado. As entidades numa cadeia de *parents* são designadas de *ancestors*.

As operações no *datastore* são feitas através das operações *put* (i.e. inserir), *get* (i.e. obter) e *delete* (i.e. apagar), que usam a chave para identificar a entidade. Uma actualização é feita com uma operação *put,* indicando a chave da entidade a actualizar e os dados a inserir. Uma inserção descuidada não só esmaga dados existentes, como pode inserir outro formato de dados no lugar dos dados anteriores, pois não há *schema*.

Todas as operações ocorrem numa transacção. Uma transacção iniciada explicitamente tem um nível de isolamento *serializable*; uma operação executada sem transacção iniciada tem um nível de isolamento equivalente a *read commited* (a inserção de cada entidade é atómica). Uma transacção iniciada só pode operar sobre dados no mesmo *entity group*.

Dentro das transacções as operações têm garantia de operarem sobre dados no mesmo estado do início da transacção. Operações executadas dentro de transacções não são visíveis. O mecanismo transaccional é optimista, sucedendo apenas se a última modificação feita ao

grupo foi feita antes do início da transacção. Modificações sobre a mesma entidade fazem que outras transacções falhem no *commit*, havendo por isso a limitação de 10 operações (escritas) concorrentes por segundo por *entity group*.

O *datastore* permite fazer um pedido para um conjunto de operações de um tipo, em vez de um pedido por cada operação (i.e. execução em *batch*). Estas operações são mais eficientes por terem menos *overhead* por chamada e por permitirem o *datastore* executar operações em paralelo, agrupando-as por *entity group*.

Anexo D

UTILIZAÇÃO DA BIBLIOTECA TWIG

A biblioteca *TWiG* permite armazenar qualquer objecto *Java* (i.e. *POJO*) no serviço *datastore*. A indicação de como uma entidade deve ser armazenada é feita através de anotações, usadas para indicar campos que não devem ser persistidos, o identificador de uma entidade, um *parent* ou entidades que são *child*. A sua sintaxe pode ser visualizada na Listagem D.1.

```
1  @Entity(allocateIdsBy=10)//needed to create id after the creation of timeline
2  public class Timesheet implements ITimesheet
3  {
4      @Parent private Timeline parent;
5      @Id private int id;  //timesheet id;
6      private int version; //should be incremented in each update
7
8      private int year;
9      private int month;
10
11     @Child private DayInTimesheet [] days;
12 }
```

LISTAGEM D.1 - POJO USADO PARA ARMAZENAR UMA TIMESHEET NO DATASTORE

Na Listagem D.1 a entidade *timeline* é identificada como *parent* da entidade *timesheet*; esta anotação é uma indicação para criação da chave da entidade, não sendo a chave da *timeline* referida através num campo da entidade. Da mesma forma, a anotação *@Id* é usada para indicar o campo com o identificador da chave; a sua omissão indica que o identificador é automaticamente gerado pelo *datastore*. Por fim, a anotação *@Child* indica que as entidades *DayInTimesheet* contidas, quando criadas, devem ter como *parent* a entidade *timesheet* a que estão associadas.

As entidades são manipuladas através de instâncias que implementam *ObjectDatastore,* onde são executadas operações de *store*, *update*, *load* e *delete*. De forma a suportar múltiplas referências para a mesma entidade, existe um mecanismo interno de cache que verifica quais

as entidades já foram lidas do *datastore*, pedindo só as que ainda não existem localmente. Assim, é garantido que não existe localmente mais de uma instância que representa uma entidade. Nenhuma entidade é obtida pela chave: as entidades são sempre obtidas por indicação do tipo (i.e. objecto *Class*), o seu identificador e, caso exista, a referência para a instância da sua entidade *parent*.

Anexo E

APLICAÇÃO CLIENTE

E.1 NOTAS

De seguida estão enumerados várias notas sobre os princípios usados no desenvolvimento da aplicação cliente, mas que não foram incluídos no resto do relatório.

O aspecto gráfico da aplicação é manipulado através das implementações dos componentes *builder*, que utilizam as bibliotecas *JQuery-UI* e *Knockout.js*. Os controlos mostrados são modificações de controlos existentes, como o controlo para edição de recursos, ou foram criados com o auxílio das funcionalidades destas bibliotecas, como é o controlo de edição das *timesheets*. A manipulação de objectos utiliza os mesmos princípios da biblioteca *JQuery-UI*: o estilo é definido através de CSS e os elementos são associados a classes específicas para serem aplicados certos estilos ou identificados certos tipos de controlo, de forma a serem facilmente obtidos através de um selector *JQuery*.

Toda a aplicação cliente é orientada a objectos, sendo usada como convenção o carácter "_" para indicar os métodos privados.

O ficheiro *utilities.js* disponibiliza a informação necessária aos controlos que não tenha nenhum sítio específico onde possa ser inserida. Neste ficheiro estão várias declarações necessárias, como códigos de respostas *HTTP*, constantes necessárias, funções utilitárias e *custom-binds* complementares à biblioteca *kncokout.js*.

E.2 ASPECTO DA APLICAÇÃO

De seguida são mostrados vários ecrãs com o aspecto da aplicação cliente.

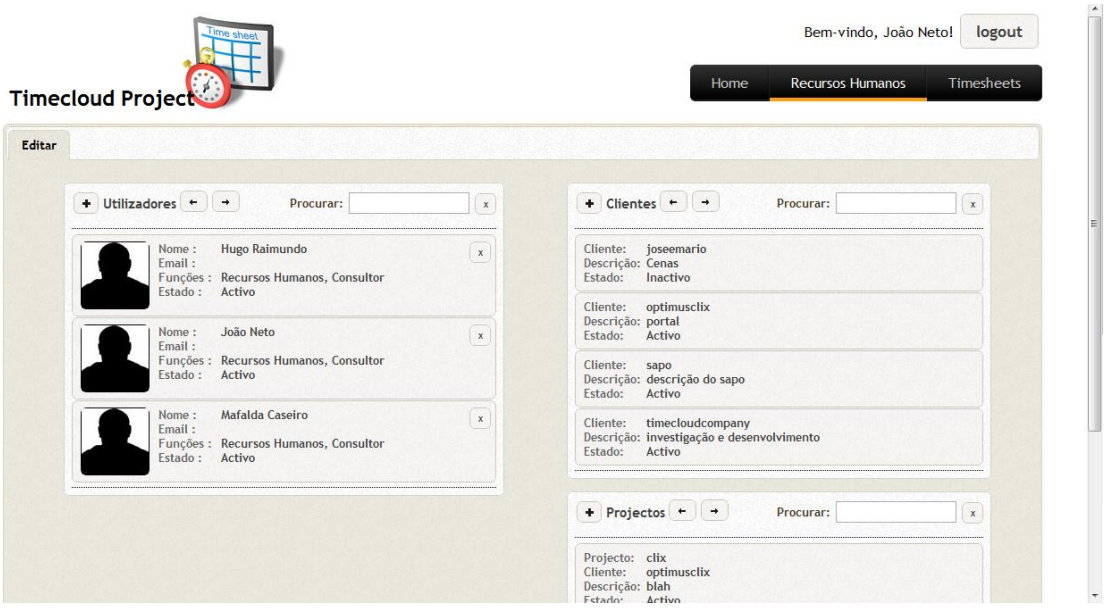

FIGURA E.1 - ECRÃ DE GESTÃO DE RECURSOS

FIGURA E.2 – CONTROLO EM MODO DE EDIÇÃO DE UM UTILIZADOR

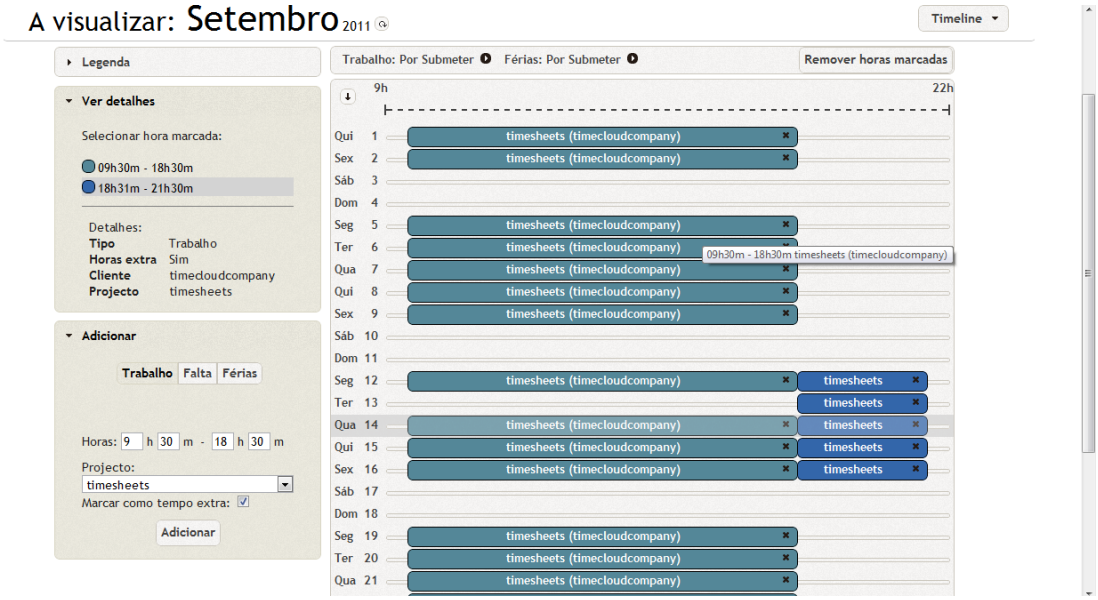

FIGURA E.3 - ECRÃ DE EDIÇÃO DE UMA *TIMESHEET*

Para demonstração a aplicação cliente foi publicada no domínio timecloudproject.apppspot.com, sendo no entanto necessário o utilizador ser autorizado para o utilizar. Para este fim foi autorizado o *utilizador* timeclouduser@gmail.com, cuja *password* é tcproject.

Anexo F

JSP TEMPLATES

Os *JSP templates* utilizam a tecnologia *JavaServer Pages Standard Tag Library (JSTL)(53)* para disponibilizar três elementos: *get*, *put*, e *insert*. Estes elementos permitem injecção de código numa página modelo (i.e. *template*), permitindo definir uma página com os elementos comuns entre as páginas de um *web site*, para que cada página só tenha de disponibilizar o seu conteúdo específico. Um exemplo de uma página *template* é mostrado na Listagem F.1.

```
1  <%@ taglib uri='/WEB-INF/tlds/template.tld' prefix='template' %>
2  <%@ page errorPage="/resources/pages/errorPage.jsp" %>
3  <html>
4      <head>
5          <title> <template:get name='title'/> </title>
6          <style> (...)  </style>
7          <script type="text/javascript"
8                  src="resources/static/js/jquery-1.6.1.js"></script>
9          (...)
10         <template:get name='head'/>
11     </head>
12
13     <body>
14         <table id="header">
15         <div>
16             <jsp:include page="/resources/controls/login.jsp" />
17             <jsp:include page="/resources/controls/menu.jsp" />
18         </div>
19         <p></p>
20         <div id="bodyContainer"> <template:get name='body'/> </div>
21     </body>
22 </html>
```

LISTAGEM F.1 - EXEMPLO DE UMA PÁGINA *TEMPLATE* USANDO *JSP TEMPLATES*

Na Listagem F.1 podem-se verificar pontos de injecção, definidos através do uso do atributo *get*, nas linhas 5, 10 e 20. O elemento *get* tem um atributo *name* que o identifica. Estes pontos

de injecção são depois usados por uma *JSP* que indica como o *template* deve ser preenchido. A Listagem F.2 mostra uma JSP que utiliza o *template* definido.

```
1  <%@ taglib uri='/WEB-INF/tlds/template.tld' prefix='template' %>
2  <template:insert template='/resources/pages/master.jsp'>
3    <template:put name='title' content='Timesheets' direct='true'/>
4    <template:put name='head'
5                  content='/resources/content/timesheet/head.jsp'/>
6    <template:put name='body'
7                  content='/resources/content/timesheet/body.jsp' />
8  </template:insert>
```

LISTAGEM F.2 - JSP QUE UTILIZA O TEMPLATE MASTER.JSP

A utilização da página *template* é feita com o elemento *insert*, mostrado na linha 2. O ficheiro com o modelo a ser usado é identificado com o atributo *template*. A injecção do código no *template* é feita através de elementos *put*, onde deve ser indicado o atributo *name* com o identificador do elemento *get* e o atributo *content* com a localização do ficheiro com o código que substitui o elemento *get*. No elemento *put* pode também ser indicado o atributo *direct* com o valor *true*, caso o conteúdo do atributo *content* deva substituir o elemento *get (*e.g. inserção do elemento *title* na linha 3).

Anexo G

NOTAS SOBRE O USO DA BIBLIOTECA KNOCKOUT.JS

O código cliente usa a biblioteca *knckout.js* (55) para fazer a ligação entre a interface gráfica aos dados recebidos do serviço. Esta *API* permite ligar os campos de um objecto aos controlos gráficos através de um *binding*. A quantidade de *bindings* disponíveis é diversificada, permitindo definir texto de elementos, atributos, classes de *css*, conteúdo de campos de formulários, entre outros. A declaração de um *binding* é mostrada na *Listagem G*.1.

```
1  <div id="clientHeader">
2      <div>
3          <span class="edit-header-label"> cliente: </span>
4          <span data-bind='text: ID'></span>
5      </div>
6      <div>
7          <span class="edit-header-label"> descrição: </span>
8          <span data-bind='text: description'></span>
9      </div>
10 </div>
```

LISTAGEM G.1 - DECLARAÇÃO DE UM BIND EM KNOCKOUT.JS

Na linha 4 e na linha 8 da Listagem G.1 estão declarados dois *bindings,* que são declarados através do atributo *data-bind*. O valor deste atributo tem um formato composto por um conjunto de pares chave valor, em que cada par identifica o *binding* a ser utilizado e o seu valor, respectivamente. Na linha 4, por exemplo, é usado um *binding* que faz a ligação do valor do elemento *span*, em texto, com o campo ID do *view model* a que for associado. A associação de um elemento *HTML* a um *view model,* chamada de aplicação de *binding,* é feita através de *javascript.* A Listagem G.2 mostra um exemplo dessa associação.

```
1  var item = {
2                  ID: "ISEL",
3                  description: "descrição de exemplo"
4              };
5  ko.applyBindings(item, $('#clientHeader').get(0)); //local
6  ko.applyBindings(item);                             //global
```

LISTAGEM G.2 - APLICAÇÃO DE UM BINDING EM KNOCKOUT.JS

Na linha 1 da Listagem G.2 é instanciado um objecto, chamado *item*, com os campos correspondentes aos *bindings* declarados na Listagem G.1. Um *binding* pode ser aplicado sobre toda a hierarquia *DOM* (i.e. linha 5) ou a um elemento específico (i.e. linha 6). A aplicação de um *binding* a um elemento específico é utilizada neste projecto, mediante a chegada dos elementos do servidor, sendo os *bindings* aplicados com os dados recebidos. Na declaração de *bindings* é também possível a utilização de expressões com os campos do objecto, como mostrado na Listagem G.3

```
1  <div class="ts-span ts-span-month ui-corner-all" style="top: 0px;"
2       data-bind="attr:  { title: extra.title },
3                  style: { width: extra.width() + 'px',
4                           left:  extra.left() + 'px'},
5                  css:   { 'ts-span-work':       span_type == 'WORK_SPAN' ?
6                                                 !is_extra_time : false,
7                           'ts-span-work-extra': span_type == 'WORK_SPAN' ?
8                                                 is_extra_time : false"
9      >
10     <a class="ts-span-remove" href="#" data-bind="click: function() {
11                                                    extra.removeClick(); }">
12        <div class="ui-icon ui-icon-close"></div>
13     </a>
14 </div>
```

LISTAGEM G.3 – DECALARAÇÃO DE BINDINGS COM EXPRESSÕES

A Listagem G.3 mostra o exemplo da utilização de vários *bindings* e expressões utilizadas no projecto. Na linha 3 e na linha 4 está declarado um *binding* que define atributos de estilo, cujo valor é o resultado da concatenação de um campo com a *string* '*px*'. A linha 5 e a linha 7 mostram a utilização de um *binding* que associa ou remove uma classe *css* a um elemento conforme o valor booleano associado; o seu valor é o resultado de uma expressão que usa dois campos, o *span_type* e o *is_extra_time*. Na linha 10 é mostrado um *binding* feito à função *click* de um elemento, com a função de processamento declarada *in-place*.

Para a *GUI* reflectir as modificações dos dados que têm o binding aplicados é necessária a utilização de tipos especificos: *observable, dependent observable e observable array*. O tipo *observable* é usado para campos com tipos primitivos (*integer, string*) associados; o tipo *dependent observable* é usado para definir uma função cujo valor é composto por outros campos do tipo *observable*; por fim, o tipo *observable array* é usado para reflectir inserções e remoções feitas numa lista. Os campos destes tipos passam depois a ser acedidos como se

fossem uma função (e.g. ver linha 3 da Listagem G.3). A conversão dos dados recebidos para estes tipos é auxiliada com o método *ko.mapping.fromJS,* disponibilizada pela *plugin mapping*.

Existem casos em que os *bindings* disponibilizados não são suficientes (i.e. não disponibilizam informação necessária), tendo sido escritos *bindings* específicos, designados *custom-bindings*. Os *custom-bindings* usados no projecto estão declarados no ficheiro *utilities.js*.

Anexo H

NOTAS SOBRE O USO DA BIBLIOTECA JQUERY TEMPLATES

As classes *builder* escritas usam a extensão *JQuery Templates*, que possibilitam uma escrita de código através do uso de *templates*. Estes *templates* são declarados estaticamente, em *HTML*. Na Listagem H.1 é mostrado um exemplo de um *template*.

```
1  <script id="client-headerTemplate" type="text/x-jquery-tmpl">
2      <div class="edit-header-data">
3          <div>
4              <span class="edit-header-label"> ${client}: </span>
5              <span data-bind='text: ID'></span>
6          </div>
7          <div>
8              <span class="edit-header-label"> ${description}: </span>
9              <span data-bind='text: description'></span>
10         </div>
11         <div>
12             <span class="edit-header-label"> ${status}: </span>
13             <span data-bind='text: statusStr'></span>
14         </div>
15     </div>
16 </script>
17
18 $( "#client-headerTemplate" ).tmpl(dictionary.edit).appendTo(container);
```

LISTAGEM H.1 - TEMPLATE QUE UTILIZA A EXTENSÃO JQUERY TEMPLATES

Na Listagem H.1 é mostrado o *template* de um elemento que mostra informação de um objecto cliente. A informação a ser inserida é identificada através do símbolo ${nome do campo}, sendo utilizado nas linhas 4, 8 e 12. Neste exemplo só a informação relativa às *labels* é inserida com este mecanismo, sendo os dados do cliente inseridos através do mecanismo de *data-binding*. A utilização de um *template* é mostrada na linha 18: o tipo de *template* é obtido através do atributo *id* e é instanciado com o método *tmpl*, que recebe um objecto com os campos com os

nomes identificados no *template*. Neste caso específico, é utilizado o dicionário do sistema, fornecendo suporte de para várias linguas ao controlo.

O sistema de *templates* é também usado na biblioteca *knockout.js*, permitindo utilizar o mecanismo de *data-binding* sobre listas de objectos. A Listagem H.2 mostra um exemplo da sua utilização.

```
1  <div id="ts-master" data-bind='template: { name: "spanItem",
2                                              foreach: spanList }'>
3  </div>
4
5  <script id="spanItem" type="text/x-jquery-tmpl">
6      <div class="ts-master-hour" data-bind='customClick: extra.spanSelected'
7
8      >
9          <div class="ts-span-detail-ui ts-span ui-corner-all" data-bind="css: {
10             'ts-span-work':       span_type == 'WORK_SPAN' ?
11                                                !is_extra_time : false,
12             'ts-span-work-extra': span_type == 'WORK_SPAN' ?
13                                                 is_extra_time : false
14         }">
15         </div>
16         <div class="ts-span-detail" data-bind="text: extra.hourStr"></div>
17     </div>
18 </script>
19
20 ko.applyBindings({  spanList: data }, toolbarContainer);
```

LISTAGEM H.2 - USO DE JQUERY TEMPLATES EM *KNOCKOUT.JS*

Na Listagem H.2 está declarado um elemento *div* cujo conteúdo é gerado com os elementos de uma lista. O elemento *div* usa um *binding* chamado *template* cuja implementação faz a geração dos elementos gráficos da lista através do *template* identificado pelo atributo *name*. Neste caso, o mecanismo de *JQuery template* é usado directamente pela biblioteca *Knockout.js* bastando aplicar os *bindings* sobre o elemento, como mostrado na linha 20. Este mecanismo é usado para listar as horas de trabalhos marcadas num dia seleccionado de uma *timesheet,* não tendo sido usado noutros controlos por só ter sido descoberto após o seu desenvolvimento.