Tábula : uma framework para o desenvolvimento de aplicações REST

Guilherme Gustavo Ramos Gomes

(Licenciado)

Tese Submetida à Universidade da Madeira para a Obtenção do Grau de Mestre em Engenharia Informática

Funchal – Portugal

Novembro 2008

Orientador:

Professor Doutor Paulo Nazareno Maia Sampaio

Professor Auxiliar do Departamento de Matemática e Engenharias

Universidade da Madeira

Abstract

The development of web applications is in our days an important area due to the dissemination of Internet access and the use of the browser as an universal client application. The typical HTML based web application is being challenged by technologies allowing richer interfaces that communicate to a back end via services. The existing tools for creating web applications haven't widely adopted this approach yet and the existing support is lacking in offering developers dynamic and easy to use tools. Traditional SOAP-based web services and Representational State Transfer (REST) are alternatives for communication between the frontend and the backend. REST has recently gained visibility and its lower entry barrier seems adequate for many applications. However, the lack of standards leads developers to implement REST services in many different ways even on the same platform, and there is no common agreement on how to describe them. In this dissertation, we propose a new framework for the rapid development of REST based applications. For that purpose, specifications and administration services are presented in order to solve the issues of configuring a REST-based service layer. Additionally, a new approach is presented to process the requests, allowing also the application of AOP principles.

Finally, the implementation of a framework that supports these specifications is presented. In this scope, we aim at exposing the potentialities of this dynamic framework, aimed at maximizing developer productivity in the creation of REST based applications, minimizing server restart and supporting dynamically a broad range of changes.

Keywords

REST

Specification

Services

SOFEA

URI

XML

Resumo

O desenvolvimento de aplicações web nos nossos dias é uma área importante, devido à disseminação do acesso à Internet e à utilização do *browser* como uma aplicação cliente universal. A aplicação web típica em HTML está sendo desafiada por tecnologias que permitem interfaces ricas que comunicam com um *backend* através de serviços. As ferramentas existentes para criar aplicações web ainda não adoptaram esta abordagem e o suporte existente é omissa em oferecer programadores ferramentas dinâmicas e fáceis de usar. *Web Services* tradicionais baseados em SOAP e *Representacional State Transfer* (REST) são alternativas para a comunicação entre o *frontend* e no *backend*. O REST ganhou recentemente visibilidade e a sua menor dificuldade de aprendizagem parece adequada para muitas aplicações. No entanto, a falta de normas leva a implementar serviços REST de muitas formas diferentes, mesmo na mesma plataforma, não existindo um acordo comum sobre como descrevê-los. Nesta dissertação, propomos uma nova framework para o desenvolvimento rápido de aplicações baseadas em REST. Para esse efeito, especificações e serviços de administração são apresentadas a fim de resolver os problemas de configuração de uma camada de serviços baseada em REST. Além disso, é apresentada uma nova abordagem para processar os pedidos, permitindo também a aplicação dos princípios AOP.
Finalmente, é apresentada a implementação de uma framework que suporte estas especificações. Neste âmbito, temos por objectivo expor as potencialidades dinâmicas desta framework, tendo em vista a maximização da produtividade do programador na criação de aplicações baseadas em REST, minimizando o reinício do servidor e suportando dinamicamente uma ampla gama de mudanças.

Palavras-chave

REST

Especificação

Serviços

SOFEA

URI

XML

À Cátia

À Tia conceição

E em especial aos meus pais

Agradecimentos

O meu agradecimento é dirigido ao Professor Doutor Paulo Nazareno Maia Sampaio, por ter aceite ser meu orientador neste trabalho, pela confiança depositada e críticas efectuadas.

Ao meus amigos André, Joana, Jorge Cardoso, Ricardo Velas e Ricardo Pestana pela presença e companheirismo.

Um agradecimento muito especial à Cátia, pela sua constante presença e suporte ao longo do período de elaboração deste trabalho.

À minha família, em especial à Tia Conceição e aos meus pais, pelo seu apoio incondicional ao longos destes anos sem o qual não teria sido possível este meu percurso.

Índice

Índice de Figuras

Índice de Tabelas

1 Introdução

As aplicações Web assumem actualmente uma importância sem precedentes em todas as áreas da sociedade, não só pelo acesso cada vez mais comum à Internet, mas também pela crescente importância de partilhar e processar informação diversa sobre vários aspectos da nossa vida.

Este cenário actual é o culminar de décadas de investigação e implementação de estratégias diversas à criação de aplicações Web, onde muitas tecnologias promissoras não singraram neste mercado altamente competitivo e centrado na produtividade.

O desenvolvimento de aplicações Web é um campo rico em soluções variadas, sob constante mudança face ao desenvolvimento de novas tecnologias de persistência, apresentação e serviços remotos.

Desenvolveram-se frameworks[61], estruturas de suporte ou esqueletos genéricos contendo em si as bases essenciais e forçando boas práticas na implementação de aplicações, reduzindo o código e configuração necessários. Surgiram iniciativas de uniformizar o processo de implementação, bem como à definição de APIs específicas a diferentes aspectos necessários a uma aplicação, como a persistência e Web Services[199] entre outros.

Como seria de esperar de um mercado saturado de soluções, surgem novas soluções como o Ruby on Rails[148] forçando as frameworks clássicas a reconheceram a inadequação das suas estratégias, adaptando-se e alterando os processos de desenvolvimento, o que comprova que existe espaço para inovação.

Mas no actual quadro de novas tecnologias para *Rich Internet applications* (RIA)[145] tais como Flex[58], Silverlight[158] e JavaFX[94], o crescimento de computação *cloud*[30] e a divulgação de serviços remotos via SOAP[164] e REST[140], as soluções actuais poderão não ser as mais adequadas.

Este trabalho visa estudar o cenário actual, delinear as tendências futuras e finalmente propôr uma arquitectura mais adaptada a essa realidade.

1.1 Motivação

Esta dissertação na área de aplicações web tem por fundamento os seguintes factores:

A) As recentes tecnologias de RIAs permitem um comportamento muito semelhante ao das aplicações desktop clássicas, tornando obsoletas as estratégias complexas utilizadas pelas frameworks na geração e controle de HTML[81] dinâmico. Com estas tecnologias, a interface gráfica é desenvolvida como uma aplicação individual executada no cliente, não sendo gerida pelo servidor.

Simultaneamente, estas tecnologias permitem que a alteração da interface de forma transparente ao utilizador. Ao entrar na aplicação, o utilizador acede a versão mais recente da interface. Esta solução combina os melhores aspectos dos *thin clients*[178]:

- Facilidade de acesso;
- Instalação não requerida, e;
- Updates transparentes ao utilizador.

Com os melhores aspectos dos *fat clients*[57]:

- Comportamento semelhante ao desktop;
- Riqueza de controlos, e;
- Bons tempos de resposta.

As aplicações web têm vindo assim a adoptar a separação entre a aplicação servidor (*backend*[20]) e a aplicação da interface (*frontend*[64]) a correr no browser do cliente, recorrendo a serviços como forma de comunicação.

B) Por outro lado, as dificuldades inerentes à definição e configuração da camada de serviços de uma aplicação, bem como a sua implementação estão a ser alvo de vários esforços, especialmente na utilização emergente da arquitectura REST. No entanto não há um esforço conjunto portável, isto é, multi plataforma, restringindo soluções a plataformas específicas.

C) Finalmente, a escalabilidade e disponibilidade de aplicações Web continuam a ser um problema considerável, mas a comercialização da computação em cloud permite às empresas que dominam este sector oferecer as suas estratégias a um preço competitivo, com acordos de garantias de qualidade e disponibilidade de serviço. Tornam-se possíveis bases de dados massivamente escaláveis, instanciação de servidores a pedido, todo um conjunto de diferentes formas de escalar uma aplicação até agora inacessíveis a pequenas empresas e indivíduos.

A disponibilização destes recursos é porque até então cada empresa desenvolveria as suas próprias soluções e meios para resolver a questão de escalar a sua infraestrutura. O mercado altamente competitivo criou uma grande resistência à divulgação das estratégias criadas, levando ao desenvolvimento repetitivo das mesmas soluções, um exercício redundante feito pela não partilha de informação.

O abrir deste novo mercado da oferta de infraestruturas que garantem a escalabilidade e disponibilidade veio mudar este cenário. A Amazon[8] foi a empresa pioneira, e o seu sucesso levou a IBM[85], Microsoft[114] e SUN[174] a anunciar as suas próprias iniciativas na área da computação cloud.

D) Este trabalho incide sobre o desenvolvimento de aplicações web na plataforma Java[92]. Esta plataforma é interessante pelas suas capacidades de portabilidade, sendo muito utilizada em contextos industriais e empresariais para aplicações de grande dimensão. O Java é

popularizado como uma plataforma difícil de utilizar na construção de aplicações web, muito embora haja uma grande diversidade de frameworks disponíveis.

Existem portanto vários movimentos de mudança aos processos tradicionais de implementação de uma aplicação web. Exige-se um trabalho de estudo que esclareça o cenário actual, os factores de mudança e os padrões futuros, que proponha e desenvolvida ainda uma ferramenta adaptada ao novos processos.

1.2 Contribuição

A principal contribuição desta tese refere-se à proposta e implementação de uma framework de desenvolvimento de aplicações web orientada para serviços REST. No decorrer deste processo, esta dissertação também apresenta as seguintes contribuições:

- Realização de um estado da arte sobre as frameworks existentes e as suas características;
- Identificação das actuais tendências no desenvolvimento e instalação de aplicações Web;
- Identificação do problemas actuais no desenvolvimento de aplicações Web;
- Caracterização de uma framework adaptada aos desafios futuros;
- Definição de uma especificação para configuração e administração dinâmica de serviços REST;
- Implementação de uma framework para o desenvolvimento de aplicações web com suporte à especificação de configuração e administração REST;
- Implementação de um servidor para desenvolvimento de aplicações web centradas em serviços com grandes capacidades de alterações dinâmicas, e;
- Implementação de Ferramentas gráficas para configuração dos serviços REST, controladores e *Aspect Oriented Programming*(AOP)[13].

1.3 Organização

Esta dissertação encontra-se organizada da seguinte forma:

O capítulo 2 “Estado da Arte” apresenta uma análise a várias frameworks de desenvolvimento de aplicações web, identificando as tendências de Também são analisadas novas tecnologias e abordagens com grande impacto nesta área.

O capítulo 3 “Serviços” apresenta um levantamento das diversas abordagens para a implementação de serviços remotos via Internet.

O capítulo 4 “Proposta” apresenta as características esperadas de uma framework para o desenvolvimento de aplicações web adaptadas ao cenário futuro centrado em serviços.

O capítulo 5 “Especificações” apresenta descrições de abordagens específicas por forma a solucionar as questões abertas no capítulo 4.

O capítulo 6 “Implementação” descreve o desenvolvimento da framework Tábula por forma a implementar as especificações introduzidas pelo capítulo 5.

O capítulo 7 “Testes” apresenta um estudo à performance da framework, iniciando pelas optimizações aplicadas, depois depois a uma comparação contra um conjunto de plataformas e finalmente procedendo à avaliação do impacto de escolhas de tecnologias comuns.

O capítulo 8 “Conclusões e Desenvolvimentos Futuros” apresenta as conclusões desta dissertação bem como algumas propostas de trabalho futuro a desenvolver.

2 Estado da Arte

2.1 Introdução

Neste capítulo é feita uma análise da área de desenvolvimento de aplicações web. Essa análise passa por descrever as frameworks mais relevantes, e também a identificação e caracterização de novas tecnologias e abordagens com grande impacto nesta área. Procura-se criar uma imagem das condições e tendências actuais com vista a determinar como serão as aplicações web num futuro próximo.

Este capítulo está estruturado da seguinte forma: Em primeiro lugar descreve-se brevemente a história do desenvolvimento de aplicações web, seguido da caracterização das frameworks existentes e apresentação de uma tabela comparativa. A partir do estudo das frameworks, é possível então efectuar um levantamento das tendências e dificuldades na construção de aplicações web, bem como a identificação de abordagens e soluções que se destacam. Finalmente descrevem-se novas metodologias, arquitecturas e tecnologias com grande impacto nesta área e apresenta-se um exemplo como forma de demonstração da utilização destas abordagens.

2.2 História das aplicações web

A arquitectura cliente-servidor[29] introduziu uma forma de criar um sistema distribuído constituído por software cliente e servidor, com funcionalidades distintas. O cliente efectua pedidos e o servidor aceita, processa e responde a pedidos. Esta abordagem tornou-se central no conceito de redes, sendo adoptada pela grande maioria das aplicações.

Inicialmente, as aplicações requeriam que no cliente fosse instalado um programa específico para utilização como interface gráfica, denominado de *fat clients*. Este tipo de solução torna a evolução e manutenção das aplicações muito dispendiosa, já que qualquer alteração no servidor leva à criação de uma nova versão do programa cliente e ao update de todos os clientes instalados.

O desenvolvimento de novas tecnologias no *Centre d'Etudes et de Recherche Nucléaires* (CERN) [27], nomeadamente, o padrão HTML e o protocolo HTTP[82], bem como o primeiro servidor web[83]. Com o desenvolvimento do primeiro browser[118] criou-se uma alternativa à abordagem de *fat clients.*

Inicialmente, cada página web consistia num documento HTML estático, renderizado pelo browser. Rapidamente tornou-se evidente a necessidade de algum mecanismo que permitisse enviar informação para o servidores.

O NCSA desenvolveu então a *Common Gateway Interface* (CGI) [28], um protocolo que permite a interligação entre servidores de páginas HTML e aplicações externas por forma a gerar informação dinâmica. O CGI permitiu a implementação de aplicações em C[25], mas em breve deu-se a

adopção generalizada de linguagens de natureza interpretada mais adaptadas à criação de documentos dinâmicos, tais como o Perl[133] e o PHP[134].

A popularização da Internet, e a integração dos browsers como software básico nos sistemas operativos transformou o *browser* numa aplicação cliente universal. A adopção desta plataforma como alternativa a *fat-clients* específicos, possui o benefício de não ser necessária nenhuma instalação e a modificação da aplicação ser transparente ao utilizador. O aumento de produtividade e redução de custos associados a esta solução determinou a sua popularização.

A importância crescente das aplicações web como alternativa ao modelo clássico de *fat client* captou a atenção de uma companhia interessada em divulgar a sua própria plataforma. Era o ano de 1997 quando a SUN lançou a especificação Servlet 1.0[157], dando início a uma metodologia oficial para criar aplicações WEB com Java. Esta solução tem vários pontos positivos tais como portabilidade, acesso às várias APIs do Java, independência do servidor e performance.

Nesta altura já se havia desenvolvido um conjunto de abordagens padrão à implementação de aplicações web, controle de transacções, ciclos de vida de objectos, entre outros aspectos necessários a uma boa aplicação web. Mas como não existia uma solução padrão, cada empresa desenvolvia a sua própria versão das abordagens.

A Sun lançou então o *Java 2 Platform Enterprise Edition* (J2EE)[90], declarando APIs para essas abordagens, com o objectivo de retirar aos programadores a necessidade de desenvolver soluções próprias. No entanto, embora a plataforma J2EE seja muito flexível, é também complexa, sobretudo a nível de configuração, mesmo para a mais simples aplicação.

Este enorme esforço associado à configuração levou a que muitos entusiastas de Java criassem a suas próprias soluções, melhor adaptadas às suas necessidades. Surgiram assim dezenas de frameworks com o objectivo de facilitar o desenvolvimento de aplicações web, embora apenas algumas destacaram-se o suficiente para criar uma comunidade à sua volta.

Esta variedade significa que não houve um domínio absoluto deste espaço por nenhuma framework em particular, embora algumas se destaquem significativamente, a tal ponto que na versão mais recente do Java EE 5[91], a SUN adoptou várias metodologias de outras ferramentas, tais como a Injecção de Dependências do Spring[168] ou o modelo de persistência do Hibernate[76].

Embora se possa pensar que este é um campo estagnado e saturado de soluções, a rápida popularidade de frameworks recentes como o Ruby on Rails e Grails[72] mostram que ainda há espaço para inovação.

Um aspecto negativo da grande quantidade de frameworks existentes actualmente, é que o processo de escolha de uma framework se torna difícil e moroso. A experiência nestas frameworks é uma mais-valia com grande valor no mercado de aplicações empresariais, pelo que as pessoas com estes

conhecimento não o partilham gratuitamente. Como resultado, existem muito poucos recursos disponíveis que facilitem essa escolha.

O desenvolvimento em Java ainda é caracterizado como complexo, o que aliado às poucas soluções de *hosting* e ao custo destas em relações a alojamento para PHP, tornam a adopção do Java para pequenas aplicações muito difícil. As aplicações web em Java são usadas frequentemente em sistemas empresariais de grande dimensão, geralmente em redes fechadas.

A utilização do do Java passa assim despercebida pelo o público em geral e pela maioria dos programadores de pequenas aplicações. Como plataforma para aplicações web, o Java já demonstrou a sua capacidade de escalonamento para grandes aplicações, mas não é adaptado para pequenas aplicações.

Para o âmbito desta tese, é essencial estudar as frameworks que se destacam quer pela popularidade, quer pela inovação, bem como pelas tendências futuras nesta área.

A caracterização das frameworks requer a introdução, na secção seguinte, de algumas tecnologias e abordagens referidas extensivamente ao longo deste trabalho.

2.3 Tecnologias e Soluções Relevantes

Algumas tecnologias e abordagens tornaram-se padrões *de facto* no desenvolvimento de aplicações em geral, provocando também um grande impacto nas frameworks para construção de aplicações web. Esta secção efectua uma breve apresentação de cada uma delas.

2.3.1 Model View Controller

A arquitectura Model-View-Controller(MVC)[119] foi descrita por Trygve Reenskaug em 1979 enquanto trabalhava no Smalltalk[160] na Xerox PARC[132]. O objectivo desta abordagem é a separação de uma aplicação em camadas distintas, cada uma com uma função específica. Reconheceu-se a existência de três camadas na generalidade das aplicações : modelo, controle e vista. (Figura 1)

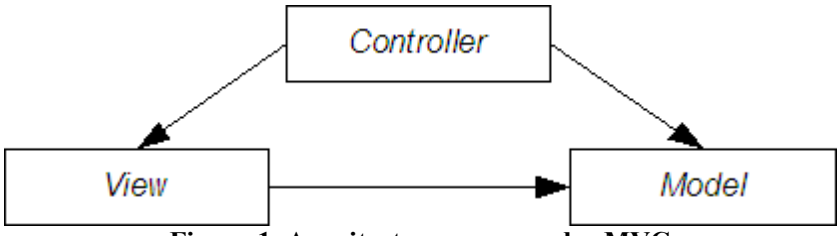

Figura 1: Arquitectura em camadas MVC

A camada de modelo representa os dados da aplicação, a vista representa a interface gráfica dos modelos e finalmente o camada de controle representa o código que efectua a gestão das acções do

utilizador pela interface, aplicando alterações ao modelo e à vista quando necessário.

Esta abordagem permite a separação de responsabilidades por cada camada, o que aumenta a organização, modularidade, e reduz o acoplamento dos componentes. Por exemplo, a camada de vista pode ser alterada sem introduzir modificações na camada de modelo.

O benefícios do MVC levaram à sua utilização em aplicações web, onde o modelo representa geralmente conteúdo guardado numa base de dados e regras de negócio associadas, a vista representa as páginas HTML geradas e o controle é o código que processa os dados retornados pela vista actual. Este processamento pode incluir aplicar alterações ao modelo e geração da próxima vista.

No entanto, as necessidades de escalabilidade das aplicações web levam a que, em muitas situações, cada camada seja distribuída por diferentes máquinas. Com hardware dedicado à funcionalidade específica de cada camada, é possível aplicar optimizações de forma isolada e consegue-se uma melhor performance. Esta abordagem requer no entanto uma adaptação da arquitectura MVC, como apresentado na Figura 2.

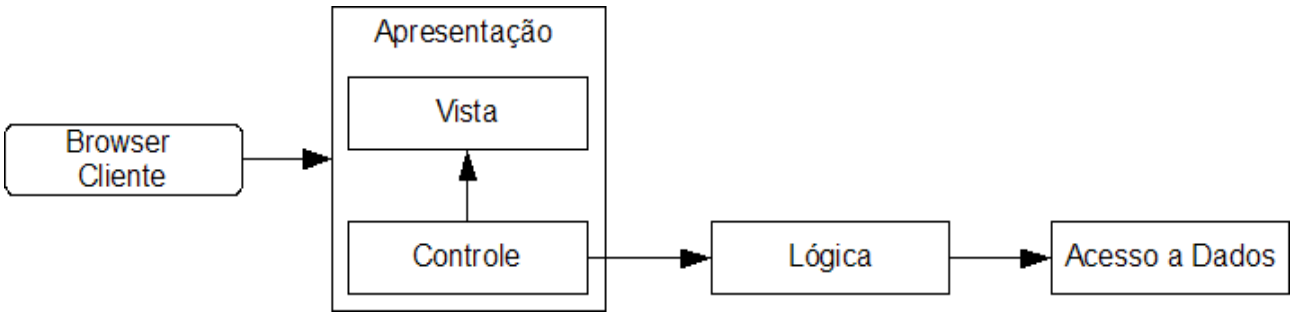

Figura 2: Arquitectura MVC adaptada a aplicações Web

A arquitectura MVC inspirou a separação da aplicação em camadas de acesso a dados, apresentação e lógica de negócio que possam ser fisicamente alocadas em diferentes máquinas. A apresentação por sua vez estrutura-se segundo uma arquitectura MVC simplificada, apenas de vista e controle.

A vista representa a tecnologia específica de interface gráfica, o que regra geral representa páginas HTML. O controle por sua vez representa o código de geração da interface, bem como o processamento das acções do utilizador. Este processamento implica o acesso à camada de lógica que implementa às regras de funcionalidade de aplicação e abstrai o acesso à base de dados da apresentação.

Com ou sem adaptações, a arquitectura MVC teve um grande impacto na organização de aplicações de todo o tipo, e é considerado um estilo padrão. Em relação às aplicações web, os benefícios da arquitectura MVC determinaram a sua adopção extensiva por todas as frameworks.

2.3.2 Anotações

As anotações são um mecanismo disponibilizado a partir do Java 1.5, cujo objectivo é decorar ou anotar o código Java com meta-dados, ou seja informação sobre as classes, que possa ser útil para todo o tipo de ferramentas. Usando a API de *Reflection* [93] do Java, para obter informação sobre a classe, é possível saber que anotações foram usadas numa classe, seus métodos e atributos.

Este mecanismo veio reduzir ou retirar a necessidade de ficheiros de configuração em XML[213], mantendo os dados de configuração na mesma classe a que se aplicam. Este é um mecanismo muito popular em várias frameworks e bibliotecas.

2.3.3 Object Relational Mapping

Um dos problemas na construção de aplicações é a persistência, ou seja, a salvaguarda de dados para posterior uso. Actualmente os sistemas gestores de bases de dados relacionais estão fortemente difundidos, e existem várias soluções comerciais e *Open Source*, bem como inúmeras ferramentas disponíveis.

Actualmente existe dificuldade em utilizar esses sistemas gestores em aplicações programadas em linguagens orientadas a objectos, como o Java. No entanto, torna-se difícil reconciliar o uso de SQL com as estruturas de dados manipulados pela aplicação. O resultado foi o aparecimento de várias soluções padronizadas e bem documentadas como o *Data Access Objects* (DAO)[38] que encapsulam o SQL necessário para a aplicação. Ainda assim, estas soluções têm de ser desenvolvidas para cada caso específico, requerendo um grande esforço.

Surgiram então tentativas de mapear automaticamente classes para tabelas e objectos da classe para tuplos da mesma tabela. A ideia é retirar ao programador a construção das expressões SQL necessárias para criar as tabelas, adicionar dados, retirar ou editar, usando ao invés uma API orientada a objectos. Este é o conceito de *Object Relational Mapping* (ORM)[131].

O exemplo mais conhecido de uma biblioteca ORM é o Hibernate, pois foi um dos projectos pioneiros que teve sucesso nesta iniciativa, ganhando popularidade rapidamente. Embora originalmente a configuração fosse feita em XML, posteriormente, passou-se a usar anotações, o que veio facilitar imenso o esforço e o tempo despendido.

Alguns exemplos de configuração são os mapeamentos entre classes e tabelas, ou entre cada atributo da classe e a respectiva coluna da tabela, que atributo é a chave primária, que estratégia usar para mapear a herança de classes, entre vários outros aspectos.

O Hibernate tornou-se tão relevante, que a SUN ao desenvolver a plataforma Java EE 5, tomou o exemplo e criou a especificação *Java Persistence API* (JPA) [106]. Esta solução tinha por objectivo criar uma API única para persistência, abstraindo a implementação específica da ORM. O Hibernate

e TopLink [181]são exemplos de implementações possíveis.

As operações são executadas através de um *EntityManager*, tais como *persist*, *merge* e *remove*. Para as operações mais complexas, sobretudo para *queries*, existe uma *Java Persistence Query Language* (JPQL)[107], semelhante ao SQL, com algumas noções de orientação por objectos.

O principal benefício do JPA é permitir a abstracção da implementação da especificação, bem como da base de dados relacional usada, permitindo que estes elementos sejam mudados sem alterar o código da aplicação.

2.3.4 Spring e Injecção de Dependências

Como no contexto deste estudo é feita uma exposição do SpringMVC[168], que é uma framework completa para construção de aplicações web, nesta secção é apresentado sucintamente a maior contribuição do Spring: a Injecção de Dependências - *Dependency Injection* (DI)[42] - ou Inversão de Controle - *Inversion of Control* (IoC) [89].

A framework SpringMVC é constituída por vários componentes, sendo um deles o gestor de *Dependency Injection*. Esse componente levou o Spring a ganhar popularidade, sobretudo porque pode ser usado fora da framework, em aplicações desktop por exemplo.

A injecção de dependências é actualmente considerada uma boa prática essencial que resolve o problema de múltiplas dependências externas. Por exemplo, uma aplicação escrita numa linguagem orientada a objectos geralmente possuirá código utilizando uma determinada API para inicializar e configurar objectos, que são recursos necessários para a execução do código. O problema desta aproximação clássica é que o código fica dependente dessa API.

A ideia da injecção de dependências consiste em efectuar o processo contrário. O objecto não pede o recurso, o recurso é injectado no objecto, já completamente configurado. O Spring, por exemplo, usa ficheiros XML e/ou anotações para configurar que recursos são necessários para uma classe. Logo que um objecto seja instanciado dessa classe, o Spring injecta no objecto, quer pelo constructor, quer pelos métodos setter, os recursos necessários. É o que se chama de princípio de Hollywood : *"Don't call us, We'll call you"*[78].

O Spring tem por objectivo ser o componente da aplicação que se responsabiliza por fornecer aos objectos os recursos necessários, retirando do código as dependências das implementações e configuração dos objectos.

A centralização da configuração dos recursos em ficheiros XML melhora muito a qualidade, simultaneamente baixando o custo de manutenção do código, já que permite retirar quase todas as dependências directas. Praticamente todos os componentes da aplicação podem ser configurados e injectados noutros de forma transparente e simples, o que melhora substancialmente a flexibilidade

do desenvolvimento. A mudança de alguma implementação ou configuração, implica apenas editar um ou outro ficheiro XML, não sendo necessárias alterações ao código. É esta características da DI que permite o fraco acoplamento dos componentes, pois estes não dependem da implementação uns dos outros, mas sim das interfaces.

A DI também pode ser usada para injectar implementações falsas ou *stubs* para testes, enquanto são desenvolvidas as implementações reais. Esta metodologia de configuração teve um grande impacto na comunidade e hoje em dia é usado ou suportado por quase todas as frameworks.

Foi apresentada nesta secção uma introdução muito sucinta sobre o IoC/DI e a biblioteca mais conhecida para a sua implementação, o Spring. Note que o Spring possui muito mais capacidades de configuração do que aqui foi comentado, propriamente documentadas no site oficial do projecto[168].

O processo de DI é visto como uma aproximação uniforme à integração dos diferentes componentes de uma aplicação, sendo hoje em dia uma tecnologia de facto. Existem várias outras bibliotecas disponíveis para IoC/DI para além do Spring, como o Guice[75], Picocontainer[135] ou o Hivemind[77].

2.3.5 Aspect Oriented Programming

O conceito de *Aspect Oriented Programming* (AOP)[222] foi introduzido por Gregor Kiczales em 1996, tendo por objectivo facilitar a modularização e reutilização de código. O desenvolvimento do projecto AspectJ[18] em 2001 veio fundamentar e popularizar esta abordagem como complementar à programação orientada a objectos (POO)[129].

A função de AOP é fornecer mecanismos que permitam adicionar funcionalidade transversal aos vários componentes da aplicação, sem os alterar ou criar dependências. É prática comum um projecto iniciar com um determinado conjunto de requisitos primários e posteriormente, os requisitos secundários levam a adicionar novas capacidades à aplicação, sobretudo no que toca a funcionalidades de *logging*, tratamento de excepções e segurança.

Por exemplo, no caso do *logging*, não é possível centralizar essa funcionalidade num módulo, pois o objectivo é na verdade efectuar o *logging* de toda a aplicação. Portanto, existem chamadas às APIs de *logging* nos vários componentes da aplicação. Este processo reduz a modularidade e capacidade de reutilização. Esta não é uma solução ideal, e levou à criação do *Aspect Oriented Programming*.

No AOP a funcionalidade adicional é transparente aos componentes da aplicação. Isto é conseguido definindo pontos de intersecção onde o fluxo de execução é alterado, adicionado um ou vários métodos que implementam as funções desejadas.

Esses pontos de intersecção geralmente são definidos com base nos métodos dos componentes. como demonstrado na Figura 3:

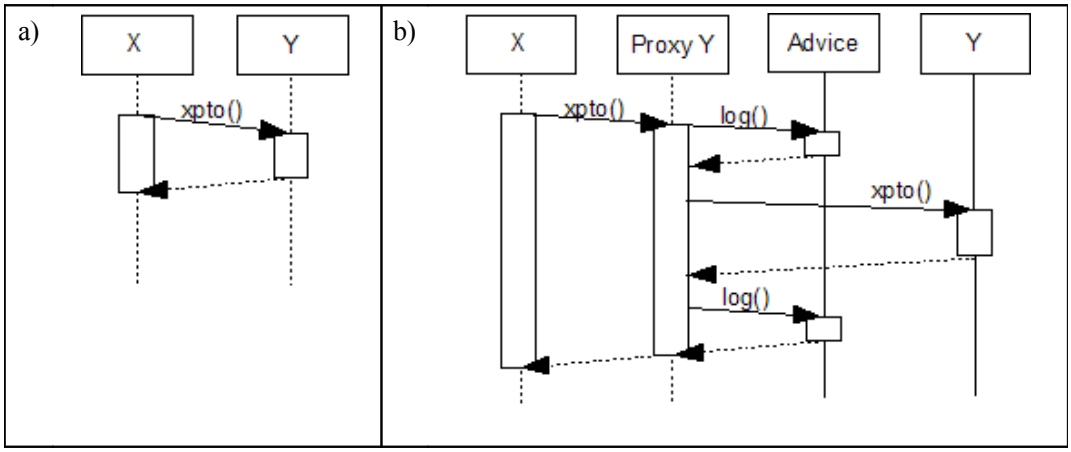

Figura 3: Chamada de método sem e com AOP

A Figura 3(a) demonstra o caso sem AOP com um componente X a invocar o método *xpto* do componente Y. Na Figura 3(b), com recurso a AOP, o componente X invoca o método *xpto* num *proxy* que substitui o componente Y. Este *proxy* é colocado em X de forma transparente pela framework de AOP, pelo que X não distingue o *proxy* do verdadeiro componente Y.

O *proxy* por sua vez, invoca um método de *log* ao componente *Advice*, que realmente implementa a funcionalidade adicional, neste caso de *logging*. Só depois é que o *proxy* efectua o método *xpto* do componente Y, após o qual novamente efectua uma acção de *log*. De seguida a execução retorna a X e continua normalmente.

Como visto, foi adicionada funcionalidade à aplicação, sendo que as chamadas adicionais foram feitas sem alteração de código nos componentes X ou Y. Esta abordagem constitui uma solução para funcionalidade transversal de aplicações sem comprometer a modularidade e fraco acoplamento dos componentes.

O AOP possui uma nomenclatura associada, que pode variar nalguns detalhes conforme a solução de implementação. No entanto, os termos mais relevantes são:

- ***Advice***: consiste no código a adicionar à aplicação. Por exemplo, *logging*, segurança ou tratamento de excepções, entre outros.
- ***Pointcut***: denomina o ponto de intersecção, ou seja os pontos durante a execução em que o código adicional necessita ser executado. Tipicamente são definidos antes e depois da execução de um método de um componente da aplicação.
- ***Aspect***: é combinação de *pointcuts* e *advice*. O *aspect* representa não só o código adicional como as regras de aplicação que definem os *pointcuts*.
- ***Weaver***: é o componente da framework de AOP que aplica os aspects aos componentes da aplicação.

O papel do Weaver é representado na Figura 4:

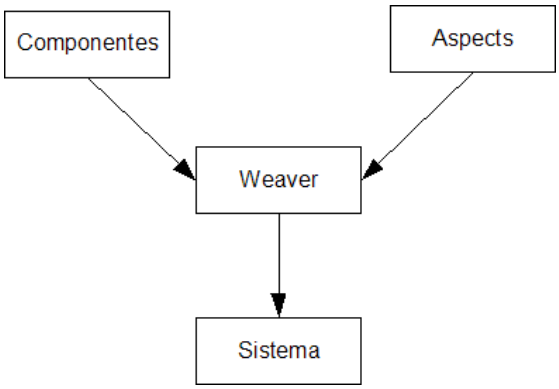

Figura 4: Integração de Componentes e Aspects

A utilização de AOP tem benefícios evidentes o que levou à sua popularização. Existem actualmente várias bibliotecas que fornecem soluções de AOP em diferentes plataformas. No entanto cada solução possui uma forma específica de declarar e configurar os *aspects*, quer por código ou por XML.

Adicionalmente algumas soluções requerem passos adicionais de compilação, antes da execução da aplicação. O AspectJ no entanto possui um modo de funcionamento em *runtime*, que aplica os *aspects* sem necessitar compilação adicional. Ainda assim, qualquer alteração na definição e configuração dos *aspects* leva ao reinício da aplicação, pelo que as soluções existentes não são dinâmicas.

O AOP fornece uma solução adaptada ao problema da funcionalidade transversal numa aplicação, e é considerada actualmente uma boa prática no desenvolvimento de aplicações.

Na secção seguinte inicia-se o estudo das frameworks de construção de aplicações web.

2.4 Estudo de Frameworks

2.4.1 Introdução

Nesta secção pretende-se apresentar o estudo efectuado às frameworks mais conhecidas para construção de aplicações web. Em primeiro lugar, indica-se as frameworks a analisar. De seguida são introduzidos os atributos considerados à caracterização geral de cada uma das frameworks, seguido de um quadro comparativo com os resultados da análise. Finalmente apresentam-se as conclusões do estudo, com destaque de algumas abordagens utilizadas pelas frameworks, identificação das tendências gerais na evolução das frameworks e dificuldades comuns ao desenvolvimento de aplicações web.

2.4.2 Frameworks

As frameworks estudadas foram escolhidas com base na sua popularidade, e por na sua maioria serem soluções para a plataforma Java, pois esse é o âmbito deste trabalho. Destacaram-se então as seguintes frameworks: Struts 1.1[172], SpringMVC[168], Java EE 5[91], Seam[154], Struts 2[173], Ruby on Rails[148], Grails[72], Tapestry[177], Wicket[196], Stripes[171] e Rife[146].

Classificação

Cada uma destas frameworks oferece abordagens específicas e únicas à criação de aplicações web, mas de forma geral, essas abordagens pertencentes a três grupos distintos de soluções:

- **Frameworks orientadas à criação de aplicações de grande dimensão** - apresentam soluções à criação e gestão flexível de componentes das diversas camadas da aplicação, sobretudo a nível de lógica de negócio e acesso a dados;
    - Struts 1.1;
    - Spring MVC;
    - Java EE 5;
    - Seam, e:
    - Struts 2.
- **Frameworks orientadas à geração automática de aplicações** – são ferramentas centradas em DDD[40], com capacidade de gerar automaticamente uma aplicação com funcionalidade CRUD[36] e interface HTML seguindo definições comuns. Têm por objectivo aumentar a produtividade do programador para criação de aplicações de pequena e média dimensão;
    - Ruby on Rails, e;
    - Grails.
- **Frameworks orientadas à apresentação** – estas ferramentas tentam simplificar a gestão da camada de apresentação, nomeadamente as interfaces em HTML e seu controle. Como tal geralmente não oferecem soluções para as camadas de acesso a dados e lógica de negócio.
    - Tapestry;
    - Wicket;
    - Stripes, e;
    - Rife.

Cronologia

É relevante visualizar a ordem cronológica do lançamento da primeira versão de cada framework, como descrito na Tabela 1. De notar que adicionamos, por uma questão de referência, a primeira versão da especificação Servlet, bem como a especificação J2EE 1.2 e J2EE 1.4. Estas especificações são a base, bem como o motivo, da construção de muitas das frameworks estudadas.

Tabela 1: Frameworks ordenadas pela data de lançamento da primeira versão

Framework	Data
Servlet 1.0	***06-1997***
J2EE 1.2	***12-1999***
Struts 1.0	07-2001
RIFE	12-2001
J2EE 1.4	***12-2002***
Tapestry	01-2003
Spring MVC	03-2004
Ruby On Rails	07-2004
Wicket	06-2005
Stripes	09-2005
Java EE 5	05-2006
SEAM	06-2006
Struts 2.0	02-2007
Grails 1.0	02-2008

2.4.3 Caracterização

Para a caracterização das frameworks foi considerado o processo que um arquitecto de software efectuaria antes de implementar uma aplicação, nomeadamente o estudo das ferramentas existentes.

Uma abordagem possível consistiria no desenvolvimento da mesma aplicação web em cada uma das frameworks. Na prática empresarial, só é viável no máximo a implementação de protótipos limitados a duas ou três frameworks previamente seleccionadas através de estudo, consultadoria ou de experiência prática anterior dos arquitectos da aplicação. As pessoas com experiência em várias frameworks raramente disponibilizam publicamente e de forma gratuita o seu conhecimento, pelo que torna-se difícil obter dados objectivos.

Assim sendo, este estudo baseia-se nas informações disponíveis para cada framework no site oficial do projecto ou em sites populares com recursos relevantes ao estudo, tais como documentação, tutoriais, *plugins*, ferramentas, fóruns e outros.

O estudo visa a obtenção de uma ideia concreta do funcionamento de cada framework, no que resulta uma descrição geral, seguida da descrição da organização e implementação das camadas MVC usando as tecnologias específicas disponíveis, e finalmente uma caracterização em termos de vários atributos. A descrição geral evidencia a arquitectura da framework, a lógica que a originou, como são implementados os componentes MVC, caso existam, e como é feita a relação entre eles.

A caracterização é feita em relação a alguns atributos relevantes no âmbito desta tese, tais como:

- **Suporte**: Que recursos existem para a aprendizagem e uso da framework. Depende da documentação, tutoriais, exemplos, fóruns, sites, livros, interesse e actividade da comunidade em geral.

- **Maturidade**: Se o código da framework é estável, usável e se há uso prático desta ferramenta, dando provas da sua maturidade.
- **Modularidade**: Capacidade de construir uma aplicação em módulos, ou secções isoladamente, que possam então ser integradas.
- **Configuração**: Quais são os pontos essenciais para a configuração de uma aplicação construída na framework. Nomeadamente, que ficheiros são usados e para que fim.
- **Extensibilidade**: Capacidade da arquitectura de framework em ser alterada e/ou adicionada funcionalidade.
- **Flexibilidade**: Capacidade de utilizar a framework em conjunto com outras tecnologias já existentes, bem como a possibilidade de usar camadas da framework fora do contexto da mesma.
- **Localização/Internacionalização**: Que procedimentos são usados tipicamente para providenciar localização.
- **Validação**: Que mecanismos estão disponíveis para efectuar validação de dados do utilizador.
- **Logging**: Que capacidades de criação e manutenção de logs de actividade e erro estão integradas na framework.
- **Instalação**: Quais os requisitos mínimos a nível de servidor de aplicação, suporte de API's Java EE e versões de Java.
- **Ferramentas**: Que ferramentas estão disponíveis para a framework em questão, nomeadamente a nível de IDEs como o NetBeans[124] e o Eclipse[48].
- **Facilidade**: indica a facilidade em utilizar a framework, as suas ferramentas e bibliotecas bem como a complexidade do código que o programador tem de escrever.
- **Aprendizagem**: Traduz a maior ou menor dificuldade na aprendizagem da utilização da framework.
- **Escalabilidade**: reflecte a capacidade de crescimento da**s** aplicações desenvolvidas na framework.
- **Dinamismo**: O suporte a mudanças *on-the-fly* da aplicação, desde alterações de configuração, ao código, bases de dados, adição dinâmica de bibliotecas, entre outros.

O estudo completo das frameworks recorreu a vários excertos de código de forma a melhor elucidar o funcionamento de frameworks indicadas. Este trabalho foi apresentado no relatório intermédio desta dissertação, e devido à sua extensão, foi decidido resumir a caracterização de cada framework disponível no Anexo 1 a uma descrição geral das suas características relevantes e organização das camadas MVC.

2.4.4 Comparação de frameworks

Após o estudo efectuado seria interessante poder avaliar cada framework em relação às outras em termos das suas características. Essa comparação no entanto é limitada pela falta de processos de construção de medidas empíricas para comparação das características referidas.

É de referir que existem métricas de software capazes de analisar alguns aspectos de um produto de software, como as linhas de código (SLOC)[159], Complexidade ciclomática[37], e acoplamento [35]. Existem também processos automáticos com alguma capacidade de análise, sendo hoje possível com recurso a ferramentas comerciais obter um perfil de código que indique não só uma multitude de bugs comuns bem como propostas de soluções básicas conhecidas por *quick fixs*.

No entanto, não estão disponíveis ainda ferramentas que permitam automaticamente analisar, diagnosticar e comparar objectivamente atributos tais como facilidade de configuração de uma aplicação, suporte a mudanças *on-the-fly* entre outros.

Medir a modularidade de uma framework, ou traduzir a arquitectura de validação de dados do utilizador por um número são dois exemplos de situações não suportadas pelas soluções existentes actualmente. A própria noção de boas práticas de software evolui com o tempo, o que implica uma renovação periódica dos processos de medida, invalidando os resultados anteriores.

A questão da produtividade em cada framework também é subjectiva à produtividade própria do programador, à experiência prévia e conhecimento em cada framework e no historial de desenvolvimento de outras aplicações, entre outros e vários factores.

Portanto, este estudo não é, nem pode ser, totalmente objectivo e imparcial, e ao invés, reflecte a opinião do autor em relação a cada framework após o estudo de cada uma delas a partir dos recursos disponíveis.

Embora não estejam disponíveis métricas para os atributos indicados, este estudo não estaria completo sem uma avaliação geral dos mesmos. Os objectivos deste trabalho não abrangem a elaboração de um sistema de classificação detalhado, mas sim, pretendem oferecer uma indicação, dentro do mais objectivo possível, da avaliação geral de cada atributo.

Para este fim, é utilizada uma escala de três pontos, onde, por exemplo, em relação à documentação, um ponto indica documentação insuficiente, dois pontos indicam documentação razoavelmente completa, de qualidade mediana e três pontos reflectem uma documentação completa e bem estruturada.

Esta abordagem é aplicada de forma similar aos atributos de Suporte, Maturidade, Modularidade, Configuração, Extensibilidade, Flexibilidade, Facilidade e Aprendizagem. Em relação à Localização, Validação, Logging e Instalação, a classificação precisa ser esclarecida:

- **Localização/Internacionalização:** reflecte a facilidade de utilização dos procedimentos para providenciar localização das aplicações.
- **Validação:** indica a qualidade das abordagens para efectuar validação de dados inseridos pelo utilizador.
- **Logging:** a classificação é indicativo da qualidade de integração de soluções ou bibliotecas de logging, suas capacidades e facilidade de alteração.
- **Instalação**: reflecte a oferta em termos de servidores que satisfazem os requisitos de instalação, ou seja, que podem alojar as aplicações desenvolvidas nas frameworks.

Quadro Comparativo

Assim, após esta exposição das várias frameworks, é apresentada a Tabela 2 como quadro comparativo que resume a caracterização feita:

Tabela 2: Comparação de Frameworks

	Struts 1.1	Spring MVC	Java EE 5	Seam	Struts 2	RoR	Grails	Tapestry	Wicket	Stripes	RIFE
Suporte	***	***	***	***	*	**	**	**	**	*	*
Maturidade	***	***	***	**	**	*	*	**	*	*	*
Modularidade	***	***	***	**	***	**	***	***	***	***	***
Configuração	*	**	**	*	*	***	***	*	**	**	*
Extensabilidade	***	***	***	***	***	***	***	***	***	***	**
Flexibilidade	**	***	***	***	***	**	**	***	***	***	***
L8n/i10n	**	**	**	**	**	*	**	**	**	**	**
Validação	***	***	**	**	**	*	*	**	***	*	*
Logging	*	**	***	**	**	*	**	***	*	***	***
Instalação	***	***	**	***	***	*	***	***	***	***	***
Ferramentas	**	**	***	**	**	*	*	**	*	*	*
Facilidade	**	*	*	*	**	***	***	**	**	***	*
Aprendizagem	**	*	*	*	**	***	***	**	***	**	*
Escalabilidade	***	***	***	***	***	*	**	***	***	***	**
Dinamismo	*	*	*	*	*	**	**	*	*	*	*

2.4.5 Resultados

Nesta secção apresentamos o resumo do estudo efectuado, nomeadamente, que abordagens se destacam nas várias frameworks, as metodologias e arquitecturas que têm vindo a ser adoptadas ao longo do tempo, e finalmente as dificuldades comuns à construção de aplicações web.

Estratégias

Ao longo deste estudo destacam-se algumas metodologias, estratégias e soluções desenvolvidas nestas ferramentas:

- **Struts 2 -** a estruturação da parte de controle, com a possibilidade de adicionar interceptores às acções, dá origem a uma arquitectura flexível de controle com suporte a AOP;
- **Spring -** a configuração por *dependency injection* torna o código coeso, mas não acoplado, ou seja, não dependente um do outro. Aumentando muito a capacidade de manutenção do código, a sua modularidade, flexibilidade bem como a facilidade de utilização em testes;
- **Ruby -** são retirados conceitos importantes:
    - É possível programar em MVC facilmente - Os utilizadores desta plataforma assimilam rapidamente o conceito de MVC, reflectido em toda a framework;
    - É possível ter boa produtividade aplicando boas metodologias como o MVC, e;
    - *Convention over Configuration*[33] – a maioria das aplicações seguem estratégias de configuração e estruturação semelhantes, pelo que por definição, a framework deve reconhecer sem necessitar nenhum esforço adicional;
- **Ruby on Rails -** Baixa dificuldade de aprendizagem (*low entry level)* ou seja, o nível requerido de experiência em programação para construir uma aplicação é baixo, não só pelos conceitos relativamente básicos como pela linguagem de *scripting* que permite um ciclo relativamente rápido de implementação e teste;
- **Grails -** é possível implementar a metodologia RoR na plataforma Java, com os mesmo benefícios da facilidade de desenvolvimento, e simultaneamente a utilização de bibliotecas Java de grande qualidade, tais como Spring e Hibernate;
- **Tapestry/Wicket/Stripes -** Estas três frameworks, embora distintas entre si, mostram que frameworks mais simples, orientadas à apresentação são uma solução possível para a criação de aplicações de pequena e média dimensão:
    - São simples de aprender;
    - São flexíveis porque permitem a utilização de vários tipos de soluções de persistência e controle de métodos de negócio;
    - A tecnologia JSP[110] não é essencial – Só o Stripes usa JSP, o Tapestry e o Wicket usam HTML com tags próprias, e;
    - O uso de HTML permite uma *preview* da interface sem ser necessário correr a

aplicação, facilitando a separação de esforços entre *designers* e programadores.

- **RIFE -** A flexibilidade extrema pode não compensar. As possibilidades de integração de componentes do RIFE é contrabalançada pela quantidade de XML necessário à sua configuração.

Evolução

Ao longo deste estudo foi possível identificar as linhas de orientação das frameworks actuais através da adopção de certas estratégias. É possível verificar uma evolução natural destas ferramentas, não só no surgimento de frameworks mais recentes, mas também na construção de novas versões de frameworks já estabelecidas.

Vistas as características de cada framework, a sua ordem de desenvolvimento, bem como os objectivos para as versões futuras é possível identificar algumas tendências :

- **Menos XML -** a diminuição do uso de ficheiros XML para configurar a aplicação é conseguida usando as seguintes estratégias:
  - **Convention over Configuration -** suporte automático aos cenários mais comuns de utilização, sendo a configuração manual só requerida para os casos de excepção, e;
  - **Anotações -** uso das anotações para adicionar meta-dados usados pelas frameworks para configurar a aplicação.
- **Dinamismo –** adição ao suporte a alterações *on-the-fly* da aplicação, sem ser necessário reiniciar o servidor. Recorre às seguintes abordagens:
  - **Hot Swap[79]-** permite alterar a implementação de um método;
  - **Class reloading -** permite redefinir uma classe, e;
  - **Scripting -** Uso de linguagens de *scripting* como Groovy e Ruby.
- **RIA –** implementação de interfaces ricas com recurso a AJAX[6];
- **Geração CRUD -** criação automática de uma aplicação CRUD a partir dos modelos de domínio;
- **Componentes POJOS[137] -** Não forçar a herança de uma classe ou implementação de uma interface da framework permite maior facilidade nos testes, flexibilidade e reutilização dos componentes;
- **Injecção de Dependências -** permite o fraco acoplamento e coesão forte dos componentes, como verificado pelo uso de Spring ou de outras bibliotecas de DI em várias frameworks;
- **Aspect Oriented Programming -** permite a adição de funcionalidade transversal aos vários componentes da aplicação, mantendo sua modularidade e fraco acoplamento, e;
- **ORM -** constitui numa abordagem orientada a objectos para persistência, mapeando classes a tabelas em sistemas de bases de dados relacionais, aumentando a facilidade de utilização de bases de dados relacionais em plataformas OO.

No geral verifica-se um maior foco na facilidade e simplicidade de utilização das frameworks. A demonstração de que este é um factor importante nas frameworks é o RoR. Esta ferramenta foi criada sem arquitectura para localização de aplicações, tem limitações de *multithreading*, o que limita a escalabilidade, e mesmo assim, ganhou imensa popularidade, produzindo grande impacto na área de aplicações web. *Convention over Configuration* tornou-se uma metodologia obrigatória.

Dificuldades

Ao longo deste estudo, foram identificados vários pontos comuns a várias frameworks que tornam difíceis a criação de aplicações ricas para a web em Java:

- **Refactoring da base de dados -** usando as implementações de JPA, a mais pequena alteração geralmente resulta em destruir todas as tabelas e recriá-las, o que é dispendioso, ou então manualmente efectuar as alterações, o que pode não ser óbvio nem fácil. O RoR suporta *migrations*, onde o programador pode indicar pequenas alterações ao esquema, mas a sua criação é manual. No Grails o GORM com base no Hibernate consegue suportar várias alterações básicas sem ser necessário recriar a base de dados, mas alterações mais extensas ou complexas requerem esse processo.
- **Re-iniciar a aplicação devido a pequenas alterações - a**lterar o código da implementação, os ficheiros de configuração de XML ou anotações relacionadas com o controle, injecção de dependências ou persistência implica reiniciar a aplicação para aplicar as mudanças. Este processo consome tempo e reduz a frequência dos ciclos de implementação e teste.
- **Complexidade de Configuração** - na maioria das frameworks é requerido um grande esforço de configuração dos vários componentes da aplicação, muitas vezes utilizando diversas técnicas na mesma aplicação, o que regra geral se reflecte no uso de anotações e ficheiros XML.
- **Conceitos avançados -** *Dependency injection*, gestão de vida de componentes, arquitectura orientada a serviços, são exemplos de conceitos valiosos e essenciais à produtividade em grandes aplicações, mas que no âmbito de pequenas e médias aplicações não estão comprovados e afastam os principiantes nesta área.

2.4.6 Conclusão

Este estudo de frameworks caracterizou as ferramentas mais conhecidas, possibilitando a sua comparação e levantamento de várias conclusões. Foi possível reconhecer as tendências na evolução das frameworks, as dificuldades associadas à construção de aplicações na plataformas Java e as estratégias que se destacaram em várias frameworks.

Estes dados permitem construir uma imagem do ambiente das ferramentas disponíveis actualmente. Esta imagem é essencial para analisar a adaptabilidade das soluções existentes com o cenário futuro determinado pelas novas tendências nesta área. Esse cenário é introduzido na próxima secção, com a apresentação de novas abordagens à construção de aplicações web.

2.5 Tendências

O progresso determina uma alteração gradual dos processos de desenvolvimento, na forma de novas arquitecturas, metodologias e tecnologias. É importante reconhecer que forças estão actualmente a determinar a evolução da construção de aplicações web. Só com a identificação e caracterização desses factores se pode avaliar como serão desenvolvidas as aplicações num futuro próximo.

Nesta secção são então expostas as abordagens inovadoras mais relevantes ao processo de implementação e instalação de aplicações Web.

2.5.1 RIA

As *Rich Internet Applications* (RIAs)[145] são aplicações web que demonstram o mesmo comportamento e funcionalidade das aplicações *desktop* tradicionais. Assim, permitem os benefícios de tempos de resposta e controlos de um *fat client* com a facilidade de manutenção e modificação de um *thin client*.

A tecnologia de Applets[17] da SUN permite a criação de aplicações SWING embebidas no *browser*, mas a sua aceitação foi muita fraca devido às limitações e problemas de performance das versões do Java disponíveis na altura.

A tecnologia que levou à maior difusão do conceito de RIA foi o *Asynchronous JavaScript and XML* (AJAX) [6] que consiste na obtenção assíncrona de dados do servidor a partir da interface web residente no *browser* cliente. Na prática, utiliza o Javascript e o objecto *XMLHttpRequest [215]* para dinamicamente trocar mensagens em XML entre cliente e servidor que permitem alterar dinamicamente a página através de manipulação do seu *Document Object Model* (DOM)[43].

As Interfaces em AJAX podem ser desenvolvidas em todas as plataformas, com recurso a tecnologias como JSP ou JSF[108], PHP ou RoR. O uso de AJAX está actualmente muito difundido, existindo várias bibliotecas que agrupam funções comuns e facilitam o desenvolvimento de aplicações.

Ainda assim, o AJAX possui várias limitações a nível de funcionalidade. Esta é uma realidade que companhias como a Microsoft, Adobe[4] e a SUN perceberam, levando-as a considerar a necessidade de desenvolver novas alternativas, pois existe também obviamente uma oportunidade de mercado.

Esta é uma área sob forte desenvolvimento nos últimos dois anos, tendo a Microsoft lançado o Silverlight[158], a Adobe o Flex[58] e a SUN lançará o JavaFX[94] ainda este ano. Nesta secção vão ser avaliadas estas tecnologias.

Tipicamente estas aplicações necessitam da adição de uma biblioteca ao *browser*, geralmente sob a

forma de *plugin* ou instalação independente. Esta biblioteca fornece o motor da aplicação cliente que reside no *browser* ou em máquina virtual.

Os benefícios desta solução são vários, nomeadamente:

- Não é necessário instalar a aplicação, apenas a biblioteca requerida;
- A aplicação pode ser alterada sem grandes inconvenientes ao utilizador;
- A disponibilidade da aplicação está dependente apenas de acesso à Internet, e um *browser* com os *plugins* necessários, e;
- Apresentação uniforme nas várias plataformas.

Em contrapartida, as aplicações tornam-se dependentes das características da máquina cliente, podendo surgir limitações de velocidade ou até várias complicações relativas a permissões de segurança e requisitos da aplicação.

Adicionalmente, embora a aplicação seja executada na máquina cliente, esta pode utilizar serviços remotos disponíveis em servidores. A funcionalidade destes serviços é variada, conforme o objectivo da aplicação. Podem ser disponibilizada gestão de entidades via acções CRUD, facturação, workflows, agendas, pesquisas entre vários outros exemplos. Este aspectos dos serviços será discutido em pormenor em secções posteriores.

O aspecto relevante neste momento é que com esta tecnologia RIA que é desenvolvida a aplicação gráfica - o *frontend* - que serve de interface aos serviços disponíveis remotamente - o *backend*.

➢ Silverlight

O Silverlight é uma iniciativa da Microsoft que consiste num um plugin para browsers que permite o desenvolvimento de aplicações web com animação, gráficos vectoriais e médias dinâmicos como som e vídeo.

Foi baseado no *Windows Presentation Foundation* (WPF) [198] e como tal, usa *eXtensible Application Markup Language* (XAML)[211] podendo ser programado em JavaScript. A versão 2.0 adiciona o suporte às linguagens .NET[225].

➢ Flex

A Adobe lançou o Flex, que se pode considerar como um Flash para programadores. As aplicações são construídas com documentos MXML[120] e Actionscript 3.0[1]. É de referir também o Adobe Integrated Runtime (Adobe AIR)[5] que consiste num ambiente multi plataforma que permite correr aplicações desenvolvidas com Flash, Flex, HTML e AJAX. O objectivo é permitir desenvolver aplicações ricas em desktop da mesma forma que se construiria aplicações web.

Esta tecnologia mostra-se madura, existindo um grande interesse por parte da comunidade, possui

uma grande base instalada pois a maioria dos browsers possuem instalado o plugin Flash, e mostra consistência da apresentação em qualquer plataforma.

➢ JavaFX

Este projecto da SUN utiliza uma nova linguagem de denominada JavaFX, cujo objectivo é facilitar o desenvolvimento de aplicações com SWING adoptando uma estrutura declarativa, em que o código se assemelha à estrutura final da interface. Adicionalmente, reúne características dos applets e da tecnologia *WebStart*[194], estando dependente da versão Java 6u10, conhecida por JRE Cliente.

Esta última versão é mais pequena que as anteriores, pois inclui apenas o essencial para aplicações básicas. As bibliotecas adicionais são obtidas online à medida da necessidade da aplicação. Foram resolvidos vários problemas inerentes aos applets como a performance e o facto de correrem no mesmo processo que o browser, e adicionadas APIs para melhor integração da aplicação com a página HTML e Javascript existente.

Uma capacidade interessante desta tecnologia é que uma aplicação JavaFX a correr no browser pode ser arrastada para o desktop, passando a residir na máquina cliente. A aplicação fica então instalada e disponível para execução sem a necessidade do browser.

A SUN anunciou este projecto em maio de 2007, estando actualmente disponível uma versão inicial para aprendizagem das APIs básicas. A versão oficial deverá ser lançada no final de 2008.

➢ Considerações

Caso a SUN tivesse sido bem sucedida nos Applets em 1995, hoje em dia não estaríamos provavelmente a discutir as várias alternativas de criação de aplicações web ricas.

Os applets tiveram uma péssima recepção devido à baixa performance do Java na altura, bem como tempos de espera e bloqueios no browser. Estas experiências dos utilizadores sobre os applets constituem um dos factores contra o sucesso da iniciativa JavaFX.

Neste momento o Flex, e o Adobe AIR detêm um grande suporte. Das novas tecnologias RIA, o FLEX é a mais madura, pois está na sua terceira versão e foi lançada a versão 1 do Adobe AIR. O plugin Flash permite um aspecto idêntico da aplicação em várias plataformas, ao invés das soluções em AJAX, onde é necessário esforço adicional para garantir o funcionamento e aparência consistente nos diferentes browsers.

O Silverlight tem tido uma fraca adopção, não só pelo comportamento habitual da Microsoft de não seguir standards, mas também por não suportar à partida plataformas como o Linux – excepto através do projecto Moonlight[117]. No entanto, nos últimos meses têm sido lançadas aplicações de relevo utilizando esta tecnologia, o que pode reflectir uma mudança na atitude de adopção desta

tecnologia por parte da indústria.

O JavaFX ainda não foi lançado oficialmente, pelo que está fora de consideração para uma aplicação de momento.

- **Impacto**

As novas tecnologias RIA permitem resolver muitos problemas das aplicações Web actuais. No entanto criam novas metodologias de desenvolvimento, quebrando o paradigma anterior da aplicação Web como uma série de páginas.

O estudo das frameworks evidenciou que a maioria destas surgiu como tentativa de solucionar os problemas específicos de gerar e gerir o HTML bem como o fluxo da aplicação. A criação de tecnologias como o Silverlight, Flex e JavaFX invalida o uso dessas soluções para as novas plataformas RIA.

Sem dúvida que as frameworks actuais serão usadas ainda durante vários anos, não só pela base instalada mas como também pela dificuldade actual em termos de processamento de suportar estas tecnologias RIA em clientes portáteis como telemóveis.

A apresentação e lógica do fluxo da interface constituem o ponto central no caso de frameworks como o Struts, Tapestry, Wicket, Stripes e RIFE. Como resultado, estas frameworks tornam-se practicamente irrelevantes no uso destas tecnologias RIA. Noutras frameworks como J5EE, Seam, Spring MVC, RoR e Grails é possível verificar que uma grande parte das suas capacidades e funcionalidades são desnecessárias, podendo ainda ser utilizada a camada de controlo se exposta via serviços remotos.

Estas novas tecnologias constituem uma mudança drástica na aproximação ao desenvolvimento de aplicações Web, e requerem por isso soluções diferentes das actuais.

2.5.2 SOFEA/SOUI

O desenho de aplicações web em geral baseia-se no padrão arquitectural MVC, e em muitos casos, todas as camadas MVC são implementadas na mesma aplicação.

Uma questão relevante sobre esta abordagem diz respeito à concepção de uma interface do usuário (UI)[186] para a Internet, que não é trivial uma vez que não podemos aplicar a mesma abordagem que na concepção das aplicações desktop.

Os Applets da SUN foram criados para superar esta limitação, permitindo que programadores de aplicações web implementassem de forma semelhante que na concepção das aplicações desktop. No entanto, Applets nunca foram amplamente utilizado devido a problemas de desempenho e recursos.

Como resultado, muitas frameworks web têm sido propostas como soluções diferentes para lidar com o desenho de aplicações baseadas em camadas MVC, e em particular com as questões relacionadas com a concepção de IU baseadas em Web.

Com o desenvolvimento de Web Services [199], as frameworks existentes começaram a adoptar este novo paradigma na sua implementação. Assim, as aplicações web ganharam a responsabilidade adicional em fornecer serviços além de lidar com a persistência, lógica de negócio, e ainda inclui a apresentação localização, validação e fluxo da aplicação, entre outras questões.

Uma solução alternativa para o desenvolvimento de aplicações web baseadas em Web Services surgiu com a proposta de uma nova arquitectura, chamada *Service-Oriented Front-End Architecture* (SOFEA) [166]. O conceito de SOFEA permite remover do servidor a responsabilidade da interface da aplicação.

O utilizador utiliza o sistema por meio de outra aplicação, quer um programa cliente instalado no desktop que acede os serviços, ou por outra aplicação web, que é responsável apenas pela interface do usuário, utilizando os serviços remotos conforme necessário. Este conceito simplifica consideravelmente a criação e manutenção do servidor de aplicação, colocando a interface de utilizador fora das suas responsabilidades.

Conforme surgir a necessidade de suportar tecnologias adicionais de apresentação, outra aplicação é desenvolvida e implementada para essa tecnologia específica de interface, sem ser necessário aplicar alterações ao servidor da aplicação.

A arquitectura SOFEA também é conhecida como *Service Oriented UI* (SOUI) [167]. A principal diferença entre as duas é a utilização de XML no SOFEA como formato de transporte de informação, enquanto que no SOUI recorre-se à *JavaScript Object Notation* (JSON)[109].

Apresenta-se outra designação possível, denominada *Model-Controller-Service* (MCS), baseada no MVC, onde a camada de vista é substituída por uma camada de serviços como a interface entre o servidor e os clientes.

As vantagens são :

- **Desacoplamento entre servidor e aplicação cliente** - a única ligação entre os dois são os serviços. A implementação da interface é invisível para o servidor e a implementação dos serviços é invisível para o cliente;
- **Abstracção** - A utilização de serviços é uma abstracção das tecnologias de implementação dos mesmos, podendo a linguagem ou plataforma de implementação dos serviços no servidor ser alterada sem modificar a aplicação cliente;
- **Simplificação da arquitectura** - Como a gestão da interface de utilizador é removida do

servidor, todo o código e bibliotecas necessárias para esse fim também é retirado. A variedade de soluções das frameworks estudadas para gerir a apresentação e o fluxo de páginas demonstra que a implementação do servidor será bastante mais simples. As responsabilidades do servidor passam então a estar focadas à disponibilidade de serviços, lógica de negócio, persistência, segurança e escalabilidade, e;

- **Criação dos clientes usando serviços *stub*** - torna-se possível criar a aplicação cliente, desenvolvendo toda a interface de utilizador sem recorrer aos serviços reais, mas sim a serviços falsos que emulam o comportamento esperado. Isto permite a prototipagem rápida do cliente, pois o acesso aos serviços é transparente, logo é trivial substituir pelos serviços reais.

2.5.3 SOA

A *Service Oriented Architecture* (SOA)[163] pode ser definida como um estilo arquitectural para criar aplicações baseadas em serviços fracamente acoplados para fins específicos, disponibilizados por fornecedores.

As necessidades da aplicação são definidas baseado em que serviços serão utilizados, que fornecedores serão escolhidos, e, finalmente, a aplicação é montada de acordo com esta informação. No entanto, esta abordagem ainda é encarada com relutância, uma vez que existe a necessidade de controlar a aplicação, e a fiabilidade desses serviços externos ainda pode ser questionada.

Uma analogia pode ser feita em relação ao progresso das aldeias onde cada família é responsável pela recolha de água e madeira, e a realidade actual das cidades com água e gestão centralizada de energia e gás. A responsabilidade da disponibilidade contínua desses serviços foi colocada em mãos de terceiros que cobram o seu uso.

As aplicações Web de hoje enfrentam as mesmas mudanças. Para gerar a aplicação, os prestadores de serviços serão escolhidos, como um contrato é efectuado com uma empresa fornecedora de água, gás ou electricidade.

A própria aplicação pode ser comparada à arquitectura de alimentação de água, gás e energia de um edifício. O seu objectivo é providenciar acesso a serviços externos e torná-los utilizáveis. A torneira da água ou rede eléctrica é colocada num edifício, de acordo com as preferências e necessidades do proprietário. Da mesma forma, o código é construído para lidar com esses serviços na aplicação. Afinal, torneiras e tomadas eléctricas não são nada mais do que interfaces cuidadosamente colocadas aos serviços pagos de água e electricidade.

Actualmente o acesso à Internet é tão omnipresente como a electricidade e a água, e as empresas fornecedores de serviços competirão não só na variedade de serviços, capacidade e flexibilidade mas também sobre a disponibilidade e escalabilidade dos mesmos.

Um exemplo de um serviço como comodidade é o da *Amazon Elastic Compute Cloud* (Amazon EC2)[9] onde os utilizadores pagam a utilização dos recursos disponíveis para as suas aplicações. Após o upload de uma *Amazon Machine Image*(AMI)[12], e a configuração do ambiente, um servidor virtual privado corre a aplicação. Através de Web Services, a própria aplicação pode obter instâncias de servidores EC2 adicionais, o que na prática constitui uma *server farm*[156] *on demand*[128], tornando possível a auto-regulação de aplicações web conforme a carga dos pedidos.

Outro exemplo é a *Amazon Simple Storage Service* (Amazon S3)[10] um serviço de armazenamento on-line da web que através de uma interface relativamente simples por Web Services oferece armazenamento de dados sob pedido. A Amazon por sua vez cobre taxas para os dados armazenados e para a largura de banda utilizada para o envio e recepção de dados.

Existem também soluções mais simples e específicas como por exemplo a gama de serviços disponibilizados pela Cdyne[26], que inclui, por exemplo, um serviço que permite verificar endereços postais, outro serviço que obtém a localização geográfica do utilizador a partir do IP da máquina, um serviço que permite o envio de SMS a telemóveis e outro que permite o envio de mensagens telefónicas por voz, um dos aspectos essenciais de qualquer *call center*, seja para aviso prévio de marcações ou campanhas de marketing.

A relutância de utilização de serviços assenta na introdução de vários possíveis pontos de falha, dependências externas e latência numa aplicação. No entanto, fortes argumentos podem ser utilizados em favor de tais aplicações, tais como uma melhor escalabilidade, disponibilidade, backup e sistemas de redundância.

Na verdade, estes argumentos serão utilizados como pontos-chave na concorrência entre os fornecedores de serviços, deixando aos programadores a tarefa de escolher os melhores, de acordo com os requisitos da aplicação. É o domínio do *utility computing* [189] e *software as a service* (SaaS) [152].

No domínio da linguagem popular, são utilizados os termos "*mashups*"[112] e "Web 2.0"[193] para denotar a utilização de diferentes serviços de fornecedores distintos, com finalidades específicas para criar uma aplicação.

2.5.4 Cloud Computing

O Cloud computing[30] assenta na instalação de aplicativos numa grelha computacional, em vez de uma máquina específica ou componente de hardware, permitindo a alocação dinâmica de recursos a partir de um conjunto de recursos computacionais partilhados. A Amazon EC2, o Windows Azure[19] da Microsoft, a Blue Cloud da IBM[23] e o Google App Engine [69], são três exemplos de produtos para cloud computing.

O Cloud computing como uma comodidade abre uma via inteiramente nova para aplicações online.

Com a utilização de serviços externos, o que resta construir é o código de integração que irá obter esses serviços, e utilizá-los de acordo com um conjunto de regras para fornecer a funcionalidade pretendida, bem como providenciar uma interface do usuário comum e coerente. Estas aplicações terão de ser instaladas, local ou remotamente.

Esta alternativa traz a possibilidade de hospedar as aplicações remotamente, delegando as questões de disponibilidade e escalabilidade para o fornecedor da "nuvem", deixando ao cliente a escolha dos limites dos recursos.

Apenas empresas com grande dimensão e experiência nesta área têm a capacidade de implementar esses produtos para o público em geral e ainda manter os elevados padrões de qualidade. Por isso, é esperado que um número de sistemas de computação em nuvem publicamente disponíveis num futuro próximo seja reduzido.

Em relação aos custos desta solução, embora actualmente, apenas a EC2 da Amazon esteja disponível, é provável que o Google App Engine, e o Blue Cloud da IBM sigam aproximações semelhantes em relação à cobrança. No caso do Google App Engine, é possível utilizar uma versão grátis mas limitada a nível de computação, armazenamento e limites de tráfego, permitindo que os programadores testem as aplicações antes de adquirir recursos adicionais.

Com o cloud computing, os programadores podem criar facilmente aplicações a partir de serviços em todo o mundo, utilizando recursos livres e pagos, e as aplicações podem ser alojadas numa das computing clouds disponíveis. Devido aos diferentes ambientes em cada nuvem, haverá dificuldades em garantir a portabilidade de uma aplicação de uma nuvem para outra, mas alguns esforços recentes mostram que é possível fazê-lo para certos casos específicos [16].

A computação em nuvem é relevante, uma vez que transforma sistemas fiáveis, redundantes, escaláveis de qualidade empresarial e de alto custo, anteriormente apenas disponíveis a um conjunto restrito de empresas, e transforma-os em comodidades publicamente disponíveis a preços acessíveis.

A aplicação conjunta de computação em nuvem com aplicações orientadas em serviços permite a criação de um ambiente para um ecossistema novo de sistemas altamente conectados, alojados e geridos remotamente. O actual modelo de negócios de recursos estaticamente adquiridos mudou para um modelo dinâmico, onde o cliente tem acesso uma grande variedade de recursos disponíveis de alta qualidade a preços acessíveis a qualquer hora e em qualquer lugar.

2.5.5 Domain Driven Development

A ideia central do *Domain Driven Development* (DDD)[40] é facilitar a construção de aplicações, usando a modelação do domínio do problema como ponto central de configuração. Foi referida a utilização de modelos do domínio do problema para configuração das soluções de persistência

como JPA ou Hibernate e para a criação automática de toda uma aplicação no RoR, Grails e Seam.

Em contraste, temos o Trails[182], uma framework Java para criação de aplicações web, baseado em DDD, utilizando bibliotecas como o Spring, Hibernate e Tapestry. Tal como o RoR, Grails ou Seam, gera toda uma aplicação a partir dos modelos de domínio, mas ao contrário do RoR e do Grails, não cria os artefactos da aplicação, ou seja, não cria os ficheiros nem a estrutura de directórios que seriam necessários. Esta abordagem leva a vários problemas já identificados na descrição da framework RIFE, pois toda a configuração é mantida somente em memória, limitando a configuração que possa ser efectuada.

O Trails força o DDD e limita a interface a dois tipos de páginas, *list view* e edit view, ou seja, uma lista de edição de uma entidade ou uma vista de listagem de várias entidades. Embora seja possível alterar alguns aspectos da aplicação, a confusão da documentação e exemplos leva a crer que a framework só suporta a interface gerada automaticamente, o que é muito limitado.

Este exemplo leva a referir um outro caso extremo de aplicação do DDD, a framework Naked Objects[122]. É preciso esclarecer que o Naked Objects é uma framework para construção de aplicações em desktop, com recurso a o SWING como tecnologia de interface gráfica, pelo que não é abrangido na caracterização de frameworks Web. Mas é um exemplo conhecido do DDD que deu a inspiração a projectos como o Trails.

No caso do NakedObjects, a interface é gerida automaticamente, e oferece ao utilizador um ambiente "orientado aos objectos". Ou seja, o utilizador fica exposto a uma interface onde escolhe que classes de objectos quer manipular, depois decide se quer criar uma instância dessa classe ou editar uma já existente. Um exemplo das capacidades desta solução é a construção de relações entre objectos que são efectuadas por *drag'n'drop* de um objecto na interface de outro.

Embora seja importante a gestão automática de uma aplicação a partir do modelo do domínio, já que é essencial em termos de prototipagem, ganho de tempo e produção de código esqueleto inicial, submeter o utilizador a um ambiente de manipulação directa destes objectos está longe do ideal.

Não só as responsabilidades de definição e configuração de usabilidade devem estar fora do modelo de domínio, pelo princípio de separação de responsabilidades, a usabilidade da aplicação depende de muitos outros factores que não são abrangidos pela modelação de domínio.

A natureza da aplicação, o contexto de utilização, condicionantes ambientais e os diferentes perfis de utilização não podem ser simplificados para uma interface única independente de todos estes factores. A não consideração destes factores quanto mais, garante uma má usabilidade do produto final.

É preciso portanto esclarecer que o desenvolvimento orientado à modelação do domínio do problema é útil, mas também apresenta os seus limites.

2.5.6 Exemplo

Por forma a esclarecer a integração das tecnologias apresentadas anteriormente pode ser exposto um exemplo de uma aplicação modelo.

Uma pequena empresa necessita de uma aplicação para realizar facturação, programar eventos, gerir imagens e vídeos dos produtos, assim como gerir as informações relativas a todos os empregados e clientes.

A solução padrão para esta situação é a criação de uma aplicação MVC típica utilizando uma framework, implementando cada módulo ou sistema, e aplicando bibliotecas existentes de *open source* ou produtos comerciais. Todos estes sistemas seriam hospedados na mesma máquina, bem como a interface web da aplicação.

Em contraste, a Figura 5 ilustra uma possível aplicação e implementação de uma arquitectura utilizando as abordagens inovadoras descritas nesta secção.

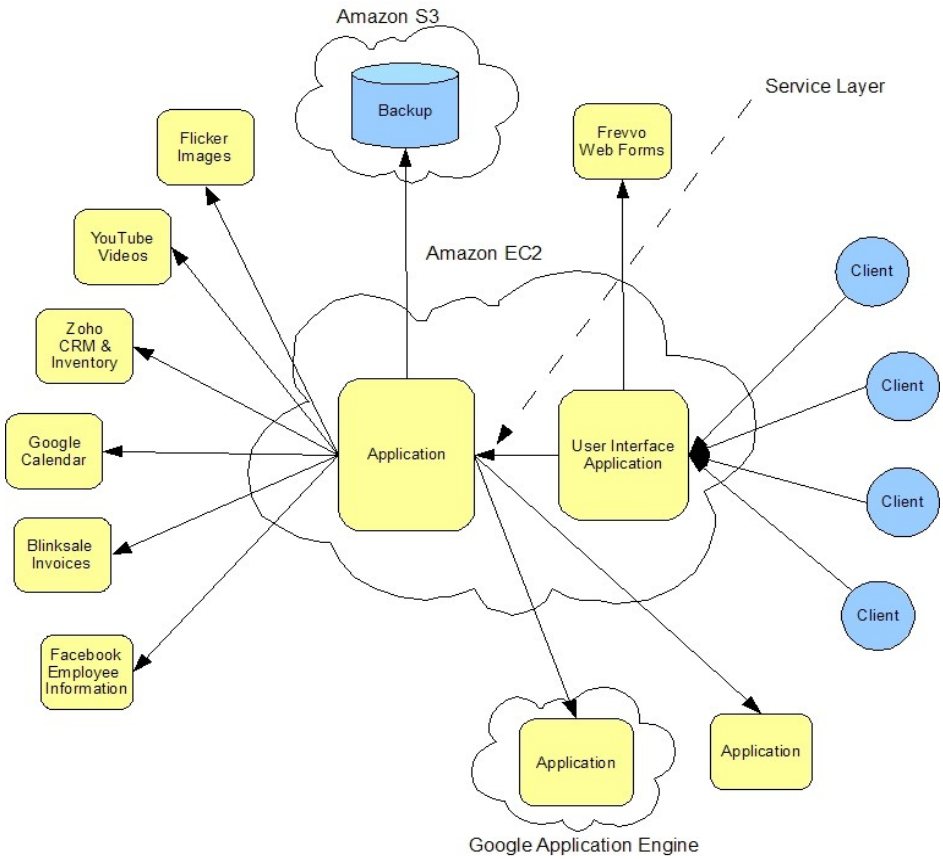

Figura 5: Exemplo de uma aplicação com SOFEA, SOA e Cloud Computing

Ao aplicar SOFEA / SOUI, a interface de utilizador seria removida para outra aplicação separada. O servidor de aplicação torna-se então mais simples de construir e manter, sendo responsável por regras de negócio, persistência e disponibilização de serviços.

Equipas distintas de peritos em interfaces de utilizador e peritos em aplicações de servidor podem

assim trabalhar em paralelo. Se surgir a necessidade de suportar outra tecnologia de interface no futuro, não são necessárias grandes mudanças para o servidor de aplicação.
Em vez da construção individual de cada subsistema, são utilizado serviços externos de diversos fornecedores , tais como Flickr [59] para imagens, YouTube [219] para vídeos, o Google Calendar [70] para agendamento de eventos, Facebook [55] ou alguma outra solução semelhante para informação de empregados, Frevvo [63] para os formulários web, Blinksale [22] para a facturação, Zoho CRM [220] para o apoio ao cliente, bem como gestão de inventário, e finalmente o S3 da Amazon para fins de backup.

A aplicação servidor que disponibiliza serviços a partir da integração das soluções referidas é desenvolvida no servidor Springsource[170], o que garante facilidade na manutenção e alteração de componentes. Esta aplicação é alojada numa *computing cloud*, com garantias de disponibilidade e escalabilidade quando necessário.

A aplicação cliente é implementada em JavaFX, e pode ser acedida a partir de outro servidor, de forma a não sobrecarregar a aplicação servidor com o tráfego de download do *front-end*.

A combinação destas soluções representam as abordagens mais recentes ao desenvolvimento, alojamento e gestão de aplicações web, abrindo o caminho para que equipas ágeis e flexíveis desenvolvam soluções inovadoras, de alta qualidade e baixo custo às necessidades do cliente.

2.6 Conclusão

Este estado da arte procurou estudar, dentro do possível, o estado actual e as tendências ao desenvolvimento de aplicações web.

As várias frameworks existentes apresentam abordagens inovadoras muito interessantes, que fornecem soluções adaptadas a vários problemas cruciais, tais como *Convention over configuration no ROR*, ou o suporte a AOP via interceptores no Struts 2, entre muitas outras soluções. Não obstante, sem dúvida que apresentam ainda varias dificuldades à construção de aplicações.

As ferramentas actuais, na sua generalidade, mostram ser evoluções às abordagens primitivas de geração e controle da aplicação via interfaces HTML. Cada framework apresenta um conjunto de soluções próprias para esse fim. Estas soluções são mais refinadas e acessíveis, suportando os casos de desenvolvimento mais comuns. O foco passou claramente de fornecer funcionalidade à sua usabilidade. Os objectivos actuais são a produtividade do programador, a usabilidade das APIs e optimização dos processos de desenvolvimento.

No entanto, as tendências actuais apontam numa outra direcção, com maior relevância nos serviços. É um cenário onde as interfaces HTML e as novas tecnologias RIA servem apenas de *front-end* a aplicações que efectuam a integração de vários serviços disponibilizados por uma variedade de fornecedores.

A arquitectura SOFEA / SOUI, por exemplo, separa a apresentação da aplicação principal. A responsabilidade de apresentação é colocado para numa outra aplicação que pode ser construída por especialistas de interface de utilizador, enquanto a aplicação servidor lida com com regras de negócio e serviços remotos. O servidor torna-se mais rápido de implementar, mais simples de entender e de ser construído.

Uma plataforma baseada em OSGi pode ser utilizada para modularizar a aplicação servidor, abstraindo cada componente e permitindo a sua alteração em tempo real, o que facilita ao seu desenvolvimento e manutenção.

Os Mashups, Web2.0 ou SOA torna, possível utilizar serviços já existentes e bem estabelecidos de diferentes fornecedores para fins específicos. O esforço para implementar cada subsistema está agora centrado na escolha dos fornecedores de serviços, assim como na aprendizagem e utilização das interfaces dos serviços.

Finalmente o cloud computing oferece aplicações massivamente escaláveis, disponíveis e redundantes a baixo custo, sem o esforço e conhecimentos técnicos necessários para implementar e manter essas características.

A partir destes dados, é levantada a questão de que as frameworks existentes não são adequadas ao desenvolvimento de aplicações web num ambiente que se espera ver dominado pelas tendências observadas, sobretudo o recursos a serviços.

Tem havido uma evolução gradual das frameworks, mas nenhuma foi desenvolvida desde o início com consideração destes factores, o que as torna em soluções pouco adaptadas aos cenários esperados num futuro próximo.

Destaca-se a importância crescente dos serviços revelada na apresentação das tendências actuais. Este aspecto requer um estudo mais aprofundado, que será desenvolvido no próximo capitulo.

3 Serviços

3.1 Introdução

O capítulo anterior identificou vários factores que apontam para um cenário de aplicações centradas em serviços remotos. Este capítulo é dedicado à exposição de várias tecnologias de serviços disponíveis. Na primeira parte será apresentado o surgimento dos serviços remotos com uma breve descrição das tecnologias mais relevantes na sua história. Os Web Services são expostos de seguida em maior detalhe, uma vez que constituem uma alternativa muito apelativa, onde desenvolvimentos recentes exigem também uma análise mais aprofundada.

3.2 História

A necessidade de comunicação entre processos levou à origem de várias soluções, da quais podem ser destacadas os *pipes*[136], e *COM*[32], entre várias outras. Com a popularização das redes e da Internet, essa necessidade estendeu-se à comunicação entre processos residentes em espaços de memórias distintos, ou seja, máquinas diferentes, comunicando por uma rede.

Assim surgiu o conceito de *Remote Procedure Call* (RPC)[149], descrito como uma tecnologia de comunicação entre processos permitindo que um programa numa máquina despolete a execução de uma função ou procedimento noutra máquina de forma transparente, ou seja, do ponto de vista do programador, o processo de executar essa função remota é indistinto de uma chamada de função local. O processo que inicia a execução é denominado de cliente, e envia um pedido com os respectivos parâmetros a um servidor remoto que efectuará a execução propriamente dita. Por sua vez o servidor retorna uma resposta ao cliente, que assim continua a execução do seu processo. Esta ideia foi definida em 1976 no RFC707, tendo a sua primeira implementação sido feita pela Xerox em 1981. O RPC também podem ser designado de *Remote Method Invocation* (RMI)[147], quando utilizado no contexto de uma linguagem orientada a objectos, como é o caso do Java.

Existem várias implementações de RPC para várias plataformas, distintas entre si por várias características, criando um cenário de incompatibilidade de protocolos.

É de destacar o Java RMI, que consiste numa API disponível na plataforma Java para a construção de aplicações em que objectos numa máquina virtual Java podem chamar métodos de outra máquina virtual, alojada na mesma ou em máquinas distintas. A utilização desta tecnologia requer a definição de interfaces, bem como a sua implementação numa classe Java. Finalmente um objecto dessa classe é instanciado e disponibilizado, podendo então ser acedido remotamente.

Outra implementação a referir consiste no XML-RPC[214], um protocolo muito simples, criado em 1998 como uma solução minimalista para os problema de RPC, utilizando HTTP como protocolo de transporte e XML como formato de conteúdo. Futuras alterações a este protocolo deram origem ao SOAP.

O *Common Object Requesting Broker Architecture* (CORBA)[34] foi lançado em 1991 e permite a comunicação entre processos mesmo que alojados em máquinas diferentes ou implementados em linguagens distintas. O CORBA utiliza uma *Interface Definition Language* (IDL)[87] para especificar as interfaces a disponibilizar. Essa especificação pode então ser mapeada a uma linguagem de programação específica.

Este mapeamento consiste, na parte do servidor, na geração automática de classes abstractas ou esqueletos, que escondem os detalhes de comunicação. São então criadas subclasses que implementam os métodos a executar durante um pedido. No cliente são geradas classes cujas instâncias servem de substitutos aos objectos remotos, advindo então a denominação de classes *stub*. Nestas classes reside todo o código necessário à comunicação com o objecto remoto, sendo que o utilizador destas classes não está exposto a esses detalhes.

O cliente e o servidor dispõem de um elemento em comum: O *Object Request Broquer*(ORB). No servidor o ORB disponibiliza os objectos, enquanto que no cliente, é responsável pelo seu acesso remoto. O protocolo utilizado nesta comunicação é definido de forma abstracta pelo *General Inter-ORB Protocol* (GIOP)[66]. Também existe uma implementação utilizando o protocolo TCP/IP denominada de *Internet Inter-Orb Protocol* (IIOP)[88].

É de referir que tanto o RPC como o CORBA são abordagens síncronas, ou seja, o cliente mantêm-se bloqueado enquanto o pedido é processado no servidor, resumindo a execução após a recepção da resposta. Este aspecto, em conjunto com a obrigação da definição de interfaces ou classes comuns, leva a caracterizar a comunicação entre cliente e servidor como fortemente acoplada.

No entanto, existem várias situações onde é desejável que um componente efectue um pedido mas continue a funcionar de imediato sem esperar por uma resposta, ou que a aplicação corra mesmo que alguns componentes estejam indisponíveis. É também desejável a possibilidade da fácil substituição dos componentes de uma aplicação, o que se torna dispendioso no caso do RPC e CORBA pois dependem de informação partilhada – interfaces ou classes – cuja implementação é morosa.

Estas necessidades requerem uma outra abordagem, com a adição de uma camada intermédia, responsável pela comunicação como a recepção e entrega de mensagens entre componentes. Um exemplo de implementação desta abordagem é o *Java Message Service* (JMS)[105], uma API Java para o envio de mensagens entre dois ou mais componentes.

A comunicação entre um cliente e um servidor deixa de ser directa passando o cliente então a criar e a enviar uma mensagem a um elemento intermédio, o servidor de mensagens, que coloca o pedido numa fila. Esta fila vai sendo processada até que a mensagem é finalmente enviada ao componente destino, o servidor que pode executar o pedido. Esta comunicação indirecta tem o benefício dos remetentes não necessitarem de conhecimento preciso sobre os destinatários, abstraindo dos detalhes de protocolos e plataformas específicos a cada parte. Permite ainda a comunicação

assíncrona, não sendo o cliente forçado a esperar por uma resposta. O JMS é portanto, um exemplo de *Message Oriented Middleware* (MOM) [116], uma alternativa de fraco acoplamento ao uso de RPC e CORBA.

Existe já uma solução comercial disponível, a *Amazon Simple Queue Service* (Amazon SQS)[11] que consiste num serviço da Amazon que suporta o envio de mensagens como meio de comunicação entre programas, em alternativa à manutenção própria de uma máquina com um servidor de mensagens via JMS. A cobrança é efectuada pela quantidade de mensagens processada.

O RPC, CORBA, e JMS são tecnologias comprovadas, escolhidas pela sua arquitectura escalável, com grande uso em aplicações de grande dimensão em ambiente empresarial. Em contrapartida, os conceitos associados são complexos com uma curva de aprendizagem íngreme, o que se reflecte no grande esforço requerido na configuração, alojamento e utilização de aplicações com recurso a estas tecnologias.

Finalmente, resta referir a abordagem de Web Services, que se destacam pelo uso de standards como HTTP e XML, conhecidos por todos os programadores de aplicações Web. A grande popularidade destas tecnologias em detrimento de RPC, CORBA ou JMS para pequenas e médias aplicações requer um estudo mais aprofundado, apresentado na secção seguinte.

3.3 Web Services

O *World Wide Web Consortium* (W3C) [191] define *Web Services* como "métodos standard de interoperabilidade entre diferentes aplicações de software, executando numa variedade de plataformas e / ou frameworks."[41]. Esta definição é de carácter geral, podendo abranger uma diversidade de soluções, embora na prática se aplique às abordagens onde é utilizado o protocolo HTTP para a comunicação entre clientes e servidores.

Os *Web Services* são um elemento essencial para a integração de aplicações, tendo-se tornado ainda mais relevantes com o advento e popularização de mashups e aplicações Web2.0. É imperativo, portanto, comparar as metodologias disponíveis para a prestação de serviços na web.

3.3.1 Web Services/SOAP

Os Web Services/SOAP dependem de um protocolo de mensagens denominado SOAP para transferir informações entre aplicações. SOAP originalmente significava *Simple Object Access Protocol*, mas foi mais tarde renomeado *Services Oriented Access Protocol*. SOAP consiste em comunicar informações estruturadas com mensagens XML através de um protocolo Internet como HTTP ou SMTP[162].

É de notar que os Web Services/SOAP são mais comummente referidos simplesmente por *Web Services*, criando confusão entre esta metodologia em particular e a definição geral de *Web Services*

tal como entendido pelo W3C. Geralmente o uso do termo *Web Service* refere-se a esta abordagem específica com recurso ao SOAP em vez da definição geral. No entanto, no contexto deste trabalho sempre que for utilizado o termo simples *Web service* entenda-se a definição geral pelo W3C. A referência *Web Services/SOAP* denomina então a metodologia apresentada nesta secção, que também pode ser conhecida por *Big Web Services* (BWS).

Um BWS também deve ser descrito em *Web Service Description Language* (WSDL)[208]. O WSDL é uma linguagem baseada em XML que fornece um modelo para descrever BGW, actualmente na sua segunda versão. O ficheiro WSDL especifica a localização do serviço, as operações ou métodos expostos, assim como os tipos de dados e protocolos de comunicação em utilização.

O objectivo é que um programa cliente que deseja conectar-se a um serviço web acede, em primeiro lugar, a um ficheiro WSDL, descobrindo que métodos estão disponíveis no servidor. Em seguida, o cliente solicita um método utilizando SOAP, enviando os dados necessários e recebendo uma resposta.

Os BWS podem ser listados num registo, como o sistema *Universal Description, Discovery and Integration* (UDDI) [185]. O UDDI consiste numa iniciativa aberta da indústria e é definido como um registo baseado em XML, independente de plataformas específicas, que permite a empresas a nível mundial publicar listas de serviços na Internet, descobrir serviços e definir a forma como os serviços ou aplicações de software interagem através da Internet.

Outra solução de registo consiste no *Electronic Business using eXtensible Markup Language* (ebXML)[47], definido pela *United Nations Centre for Trade Facilitation and Electronic Business* (UN/CEFACT)[187] e a *Organization for the Advancement of Structured Information Standards* (OASIS)[126]. O ebXML resulta da colaboração de várias entidades empresariais, governamentais e e indivíduos por todo o mundo, com o objectivo de expôr, descobrir e utilizar informação sobre BWS.

Ao contrário do UDDI, que consiste apenas num registo de metadados, o ebXML permite também armazenar conteúdo arbitrário. Ou seja, enquanto que no UDDI estaria apenas disponível metadados sobre os serviços web, cabe aos fornecedores desses serviços disponibilizarem os ficheiros de descrição WSDL no próprio site. No ebXML esses ficheiros podem ficar alojados no próprio registo, bem como qualquer documentação adicional necessário à sua utilização.

O UDDI está integrado na *Web Services Interoperability Organization* (WS-I)[203], um consórcio constituído por gigantes da indústria como a IBM, Microsoft, BEA Systems, SAP, Oracle, Fujitsu, Hewlett-Packard, Intel, Sun Microsystems e webMethods. O WS-I foi organizado para promover a interoperabilidade de serviços web, através da definição de perfis, aplicações modelo e ferramentas de teste à conformidade de perfis.

Um perfil WS é definido pela WS-I como um conjunto de especificações, juntamente com um conjunto de directrizes de implementação e interoperabilidade que recomendam a forma como as especificações podem ser utilizadas para desenvolver serviços web compatíveis.[referência].

O *WS-I Basic Profile* (WS-BP)[201] fornece linhas de orientação para a interoperabilidade de especificações essenciais aos BWS, tais como SOAP, WSDL e UDDI. A versão actual é a 1.1, com uma proposta de versão 1.2 e versão 2.0 sendo escrita.

Além disso, existem ainda especificações BWS adicionais referidas colectivamente como "WS-*", com o objectivo de alargar capacidades disponíveis, tais como suporte a mensagens, segurança, gestão, processo de negócios, entre outras.

3.3.2 REST

Representational state transfer (REST) é um estilo arquitectural descrito por Roy T. Fielding, em 2002[140]. REST não é um padrão, mas sim um conjunto de restrições a serem utilizadas no processo de construção de aplicações web distribuídas. Como um conceito de alto nível, REST não está associado a uma tecnologia específica ou implementação.

Na prática geral, porém, as restrições REST caracterizam uma arquitectura centrada em recursos, especificando que cada recurso é identificado por um *Universal Resource Indicator* (URI)[188], mediante o qual um conjunto de operações pode ser aplicado através de uma interface uniforme. Esta interface padrão uniforme para a comunicação entre servidores e clientes é o *HyperText Transfer Protocol* (HTTP) e, em vez de declarar métodos, são aplicadas acções HTTP, tais como POST, GET, PUT e DELETE. Estas quatro acções podem ser mapeadas para as acções típicas de dados CRUD: *Create*, *Read*, *Update* e *Delete*.

Os dados comunicados entre os componentes da rede - clientes e servidores - é a representação do recurso. É importante diferenciar um recurso da sua representação. Como um recurso é um objecto conceptual, quando uma aplicação efectua um pedido de GET sobre um URI específico, ela está a pedir uma representação do recurso identificado pelo URI, que pode ser representado como XML, HTML, GIF, JPEG ou AVI, entre muitas outras possibilidades. A representação desejada no pedido é especificada pelo cliente através de *Multipurpose Internet Mail Extensions* (MIME)[115] - text/xml, text/html, image/gif, image/jpeg.

Embora REST seja um conceito de alto nível, a referência ao termo REST geralmente implica a instância mais concreta de REST que recorre ao protocolo HTTP como uma interface uniforme, aos URIs para a identificação dos recursos e ao XML como forma de representação comum.

Um exemplo de um design que segue as restrições REST - também denominado por RESTful - é a *World Wide Web* (WWW)[210]. Um *web browser* normalmente executa operações HTTP GET para solicitar dados, a fim de renderizar uma página web. Quando a página contém uma imagem, o

browser simplesmente executa um HTTP GET adicional para o URI da imagem, a fim de obter os dados num formato específico (JPEG, GIF, etc). O mesmo processo é efectuado para todas as referências externas até que todos os recursos requeridos sejam obtidos e a página possa ser exibida.

A fim de ilustrar a utilização de REST, considere um exemplo prático, como uma aplicação para gestão de documentação. Nesta aplicação existe o recurso *Autor,* identificado por: */autores/{id}*, onde */autores* corresponde à secção estática do URI, seguido do parâmetro *{id}*. Este último segmento é substituído por um valor que identifique o Autor especifico pedido efectuado pela aplicação cliente. Se o cliente deseja aceder ao Autor identificado com o número 5, o URI seria */autores/5*.

Este exemplo utiliza um URI relativo, pois por si só não define a localização completa do recurso. Para esse fim é requerido um URI absoluto, contendo o protocolo utilizado, em regra geral HTTP, e o URI do servidor que aloja o recurso. A passagem a URIs absolutos neste exemplo pode ser efectuada considerando o alojamento da aplicação em *http://teste.com*. Neste caso o recurso Autores teria o seguinte URI absoluto:

```
http://teste.com/autores/{id}
```

Logo, um pedido GET efectuado ao URI absoluto de *http://teste.com/autores/245* descreve uma ordem para obter dados relativos ao Autor identificado pelo *id* de 245.

O código que implementa o processamento deste pedido deve receber a lista de parâmetros, associando a cada parâmetro um valor. No exemplo acima o código teria acesso a um parâmetro *id* de valor 245. Com esta informação, é possível invocar APIs de persistência por forma a obter os dados do Autor a partir do seu *id*, e finalmente retornar uma representação desses dados.

O tipo de representação devolvida depende dos tipos MIME indicados no pedido e suportados pelo servidor. Se o pedido indicar que aceita apenas o tipo MIME *application/xml* e o servidor suportar esse formato*,* então a resposta terá o código HTTP 200 *OK*, que indica sucesso e o conteúdo será em XML, podendo consistir, por exemplo, nos seguintes dados:

```
<autor>
   <nome>Cátia Nunes</nome>
   <morada>Rua do Sabão</morada>
</autor>
```

Se por outro lado o pedido especificar o tipo MIME *application/pdf* o processo resulta na criação da representação do autor em formato PDF. A especificação dos tipos é efectuada no cabeçalho do pedido HTTP como uma par chave e valor. A chave é "*Accept*" e o valor correspondente consiste na lista de tipos MIME aceites, separados por vírgula.

No caso em que mais de um tipo é especificado no pedido, existe um processo de negociação de conteúdos, entre o cliente e o servidor, por forma a decidir entre os tipos especificados e o tipos disponíveis. No caso de não haver compatibilidade de tipos, é devolvido uma resposta com o código

de erro HTTP 415 *Unsupported Media Type*, pois não foi possível obter um tipo comum para representar o recurso.

O processo de criação de um autor consiste num pedido POST com conteúdo que represente os dados desse Autor, efectuado para o seguinte URI: *http://teste.com/autores*. O conteúdo para este exemplo será em XML, podendo ser semelhante ao seguinte excerto:

```
<autor>
   <nome>Guilherme Gomes</nome>
   <morada>Bairro do Livramento</morada>
   <contactos>
      <contacto>291291291</contacto>
      <contacto>9696969696</contacto>
   </contactos>
</autor>
```

O servidor criará então esse Autor e confirma o processo retornando uma resposta com o código HTTP 201 *Created*, que significa que um recurso foi criado. Adicionalmente, terá de indicar no conteúdo da resposta uma representação do autor. Se for utilizado novamente o XML esse conteúdo seria:

```
<autor id="79" uri="http://teste.com/autores/79">
   <nome>Guilherme Gomes</nome>
   <morada>Bairro do Balão</morada>
   <contactos>
      <contacto>291291291</contacto>
      <contacto>9696969696</contacto>
   </contactos>
</autor>
```

A adição do atributo URI permite localizar o recurso acabado de criar. Similarmente, o processo de edição consiste em enviar uma representação dos dados a editar. Por exemplo, se for o caso de uma alteração apenas do nome, pode ser efectuado um pedido PUT ao URI : *http://teste.com/autores/79* com o seguinte conteúdo:

```
<autor>
   <nome>Guilherme Gustavo Ramos Gomes</nome>
</autor>
```

O servidor efectuará a alteração dos campos indicados, mantendo-se inalterados todos os outros atributos. A resposta retornada terá o código HTTP 200 *OK*, que indica sucesso na operação, e retorna uma representação actualizada do recurso:

```
<autor id="79" uri="http://teste.com/autores/79">
   <nome>Guilherme Gustavo Ramos Gomes</nome>
   <morada>Bairro do Balão</morada>
   <contactos>
      <contacto>291291291</contacto>
      <contacto>9696969696</contacto>
   </contactos>
</autor>
```

Uma outra forma de efectuar a alteração é aceder directamente ao nome como recurso. A aplicação pode expôr o nome no seguinte URI relativo: */autores/{aut-id}/nome*. Um pedido GET efectuado ao URI *http://teste.com/autores/79/nome* retornaria na resposta um conteúdo do tipo texto de "*Guilherme Gomes*".

Para alterar esse campo, basta efectuar um pedido PUT com o conteúdo textual de "*Guilherme Gustavo Ramos Gomes*" para o URI: *http://teste.com/autores/79/nome*. O servidor altera então o nome do Autor e retorna o recurso actualizado, que neste caso é apenas o nome. Logo o conteúdo da resposta será "*Guilherme Gustavo Ramos Gomes*".

Para remover um Autor, é enviado um pedido DELETE ao URI *http://teste.com/autores/79*, que retornará uma resposta sem conteúdo e com o código HTTP 200 *OK*. Qualquer pedido GET ao URI *http://teste.com/autores/79* retorna agora o código de erro HTTP 410 *Gone*, confirmando que o recurso foi apagado.

Este processo pode ser agora aplicado à documentação, relacionando cada recurso *Documento* a um URI relativo : */documentos/{doc-id}*. Neste caso o parâmetro *doc-id* identifica de forma única um documento específico disponível no servidor. No entanto os documentos possuem uma estrutura interna típica, pois em geral são constituídos por capítulos. Se um caso de utilização comum for o acesso a capítulos de documentos em vez do acesso ao documento completo, então estes podem ser expostos como recursos, acessíveis por um URI tal como:

```
/documentos/{doc-id}/capitulos/{cap-id}
```

Neste caso, o *doc-id* identifica o documento e *cap-id* identifica o capítulo desse documento. Temos neste caso dois parâmetros no URI, que por razões óbvias, não podem possuir o mesmo nome.

Um exemplo da utilização deste caso consiste em obter o capítulo 3 de um documento 45 no servidor, o que é traduzido por um pedido GET ao seguinte URI absoluto:

```
http://teste.com/documentos/45/capitulos/3
```

Se o pedido especificou que deseja uma representação textual, a resposta terá no seu conteúdo o texto do capítulo. É necessário referir que os capítulos são expostos como recursos componentes do documento, que é outro recurso, não havendo lógica em disponibilizar um acesso directo aos capítulos. Não faz sentido expôr os capítulos via um URI absoluto de *http://teste.com/capitulos/{cap-id}*, uma vez que estes só existem como partes constituintes de um documento.

Mas é possível relacionar diferentes recursos. Por exemplo, associar a cada documento uma lista de autores. Neste caso os autores do documento estariam disponíveis no seguinte URI relativo:

```
/documentos/{doc-id}/autores/{aut-id}
```

Onde o parâmetro *doc-id* identifica o documento e *aut-id* o índice do autor na lista de autores do documento. No entanto os autores são recursos independentes, e residem no seu próprio URI, como

visto anteriormente. Como tal, o autor de um documento consiste na verdade numa referência a um Autor já existente. Esta relação é tornada evidente pelas respostas do servidor aos pedidos.

Um pedido de GET ao URI http://teste.com/documentos/45 em formato XML pode retornar o seguinte conteúdo:

```
<documento id="45" uri="http://teste.com/documentos/45 ">
   <autores>
      <autor id="12" uri="http://test.com/autores/12" />
      <autor id="754" uri="http://test.com/autores/754" />
      <autor id="68" uri="http://test.com/autores/68" />
   </autores>
   <capítulos>
      <capitulo  uri="http://teste.com/documentos/45/capitulos/1" />
      <capitulo  uri="http://teste.com/documentos/45/capitulos/2" />
      <capitulo  uri="http://teste.com/documentos/45/capitulos/3" />
   </capítulos>
</documento>
```

Esta representação contém hiperligações para outros recursos, a fim de obter representações mais detalhadas de um recurso. Neste caso, existem *hyperlinks* para obter Autores e capítulos. Esta abordagem é poderosa no sentido de que ela especifica a localização de recursos opcionais, e proporciona ao cliente uma abordagem mais granular para obter informações.

Por exemplo, o acesso ao autor principal do documento 45 resulta num pedido GET a :

```
http://teste.com/documentos/45/autores/1
```

Se o pedido especificar que deseja uma representação em texto, ao invés de retornar uma descrição textual do Autor, é retornado um link:

```
http://test.com/autores/12
```

Este link representa a referência ao recurso Autor indicando a sua localização. O primeiro autor do documento 45, referido por *aut-id*=1 no URI dos Documentos, é então o Autor identificado por *aut-id*=12 no URI dos Autores. Para obter informação sobre este autor, é efectuado um pedido GET adicional ao URI http://test.com/autores/12.

Na condição que o pedido a http://teste.com/documentos/45/autores/1 deseje uma representação em XML do autor, esta representação deverá na mesma tornar evidente a referência à localização do recurso Autor. Por exemplo o XML retornado pode ser:

```
<autor id="12" uri="http://test.com/autores/12">
    <nome>Guilherme Gomes</nome>
...
</autor>
```

A resposta devolve não só os dados do autor, mas também torna evidente a sua localização.

Estes exemplos demonstram a abordagem mais comum à criação de aplicações REST, mas é

necessário ter em conta que os recursos, o detalhe com que os seus atributos são expostos, os métodos HTTP e formatos disponibilizados dependem do desenho da aplicação, não estando obviamente limitados ao que foi apresentado.

Além do estilo arquitectural cliente-servidor e uma interface uniforme de comunicação, as restrições REST incluem, entre outras características:

- **Comunicação *stateless***: as aplicações não guardam o estado da comunicação, pois cada interacção entre o cliente e o servidor contém todos os dados necessários para a realização da mesma;
- ***Cacheability***: a resposta pode indicar se os dados podem ser armazenadas em cache, reduzindo a necessidade de chamadas adicionais para o mesmo recurso e aumentando a performance, e;
- **Aplicações em camadas**: utilização de um sistema de componentes em camadas para a construção da aplicação que promove a flexibilidade, escalabilidade e encapsulamento.

Em resumo, o REST constrói a aplicação como um gestor de recursos em vez de um conjunto de APIs a serem chamadas remotamente. Baseia-se no HTTP como interface uniforme de acesso, e acções HTTP como os métodos disponíveis a serem realizados nos recursos e no XML como forma de representação comum.

3.3.3 REST vs. Web Services

A escolha entre implementar a camada de serviços de uma aplicação web como BWS ou baseadas em REST depende de vários factores e, como tal, existem situações onde uma abordagem é mais adequada do que a outra. Portanto, é importante uma análise comparativa entre as duas abordagens.

Esse estudo, disponível no anexo 2, foi desenvolvido tendo em contas aspectos tais como a existência de normas e protocolos, suporte a transacções e facilidade de implementação, entre outros aspectos.

Esta exposição permite verificar que os Web Services e o seu suporte ainda são vistos como um aspecto complementar ao desenvolvimento de aplicações web, não usufruindo sequer de nenhum esforço de integração por parte de várias soluções.

É constatado que mesmo nas frameworks que efectuam alguma integração das soluções existentes, quer nas que implementam a sua própria solução, as capacidades de dinamismo na definição e alteração de serviços são reduzidas. O RoR e o Grails surgem como excepções devido à sua natureza baseada em linguagens de scripting.

No geral, a adopção de BWS é superior à adopção de REST como abordagem aos Web Services,

pois apenas 4 frameworks - Grails, RoR, Struts 2 e Java5EE - possuem ou integram ferramentas para desenvolvimento de serviços REST.

No entanto, a arquitectura REST tem gerado um grande interesse devido à sua abordagem simples e uso directo de padrões como HTTP e XML. Este padrões são bem conhecidos pelos programadores de aplicações web, facilitando a aprendizagem da abordagem REST. Como resultado, têm surgido iniciativas à adição de ferramentas para desenvolvimento de aplicações.

Não obstante estes esforços, subsistem ainda várias questões na criação de aplicações REST:

- Não existe nenhuma especificação standard globalmente aceite pela comunidade para definir a camada de serviços de uma aplicação;
- Os princípios REST são independentes da plataforma, mas a ferramentas actuais são bastante específicas. Os programadores têm de aprender um novo conjunto de ferramentas para cada plataforma onde queiram implementar os serviços;
- Implementar uma camada de serviços requer testes exaustivos que levam a alterações, mas as ferramentas actuais não suportam alterações dinâmicas, pelo que pequenas alterações levam a reiniciar a aplicação. Estas situações atrasam o desenvolvimento por serem muito frequentes, e;
- Não existe descrição de uma interface ou serviço de administração comum, independente de plataforma, capaz de aplicar mudanças dinamicamente à camada de serviços.

Com base nos dados expostos neste capítulo, é possível retirar várias conclusões, apresentadas de seguida.

3.3.4 Conclusão

As tecnologias de RPC, CORBA e JMS embora comprovadas e maduras necessitam não só de um grande esforço na sua configuração e utilização bem como são caracterizadas pela dificuldade de aprendizagem.

As metodologias de Web Services são actualmente mais apelativas à implementação da camada de serviços de uma aplicação, pelo uso de standards bem conhecidos, relativamente simples e de aprendizagem mais acessível.

As frameworks actuais adicionam suporte aos WS como algo opcional, geralmente não beneficiando de abordagens que facilitem o desenvolvimento da camada de serviços, como acontece nas camadas de MVC.

Das metodologias de WS, o REST possui uma dificuldade inicial de aprendizagem menor que os BWS, contando com a familiaridade dos programadores de aplicações web com as tecnologias

utilizadas, para tornar os seus conceitos mais simples de entender. À medida que as aplicações orientadas a serviços ganham popularidade e mais programadores têm interesse em implementá-las, há a necessidade de uma abordagem mais simples.

Na prática, ambas as soluções podem ser utilizadas lado a lado, já que há situações onde uma abordagem orientada a actividades é mais apropriada, ou quando é necessário um suporte extenso a aspectos de segurança e transacções.

O REST destaca-se pela simplicidade, agilidade bem como ausência de especificações extensas e seu impacto respectivo, o que se reflecte na popularidade e velocidade de adopção desta abordagem sobretudo em aplicações de pequena e média dimensão. Conclui-se portanto que uma framework centrada em serviços necessita de suportar extensivamente o estilo arquitectural REST devido à sua crescente relevância.

O ponto mais relevante deste capítulo é a desadequação das frameworks actuais para o desenvolvimento de aplicações centradas em serviços. É sem dúvida possível construir essas aplicações, mas as metodologias actuais apresentam dificuldades no desenvolvimento de serviços que se assemelham às dificuldades existentes no período inicial de construção de aplicações com interface HTML.

O suporte a serviços tem vindo a ser gradual e a pequenos passos, não beneficiando do suporte a mecanismos de alteração dinâmicos. A sua implementação fica assim dificultada e ainda é relegada para segundo plano.

Com este capítulo, terminou a secção de avaliação das tecnologias existentes, iniciada no estado da arte. Estes dois capítulos apresentaram um conjunto de informações essenciais a esta dissertação, pois é possível afirmar finalmente que esta área requer um trabalho que aborde os problemas identificados. A proposta desse trabalho é apresentada no próximo capítulo.

4 Proposta

4.1 Introdução

Neste capítulo é feita a descrição de uma proposta para uma nova framework web. A construção de uma nova ferramenta desta natureza não pode ser tomada sem considerar que consiste num projecto de grande esforço na delineação de uma arquitectura e sua implementação, numa área onde actualmente existem várias soluções estáveis e funcionais, que variam nas suas características, como evidenciado no estado da arte.

No entanto, torna-se claro que existe ainda espaço para inovação, como verificado pelo impacto do recente *Ruby on Rails*. Esta framework apresentou novas metodologias de configuração e desenvolvimento de aplicações que foram elogiadas e cujos conceitos foram assimilados por outras frameworks. Se forem consideradas todas as frameworks existentes na altura, seria de presumir que um aspecto tão essencial como a configuração de uma aplicação fosse um processo optimizado e bastante usável. No entanto essa expectativa está incorrecta, como demonstrado. Deve ser considerada então a possibilidade de que algum aspecto essencial ao desenvolvimento de aplicações web não esteja à altura do esperado nem do desejado face às necessidades actuais.

Este capítulo é estruturado da seguinte forma: em primeiro lugar é justificada a necessidade de uma nova framework face aos dados obtidos no estado da arte, e em seguida é efectuada a descrição geral das características dessa mesma framework. Essa características consistem em requerimentos relevantes face às conclusões do estado da arte bem como considerações de usabilidade dos programadores. Como tal a descrição é feita a um alto nível, não indicando tecnologias específicas.

4.2 Fundamentação

Como evidenciado no estado da arte, pode ser levantada a questão da adaptabilidade das soluções existentes, face às forças que afectam o processo de desenvolvimento de aplicações web, enunciadas de seguida:

- As novas tecnologias RIA não necessitam das estratégias desenvolvidas pela grande parte das frameworks existentes em relação à criação de páginas HTML e gestão do fluxo da aplicação;
- A arquitectura SOFEA/SOUI apresenta-se como uma alternativa mais adequada à implementação do *backend* de uma aplicação onde o *frontend* consiste numa solução RIA, apresentando ainda vários pontos positivos a nível de redução de complexidade e custos de manutenção;
- A existência e disponibilização de vários tipos de serviços remotos permite utilizar uma solução já implementada, comprovada e escalável ao invés do esforço de produção de um subsistema equivalente;

- O *cloud computing* permite o alojamento de uma aplicação numa nuvem de recursos disponíveis à distância de uma chamada a um serviço, com garantias de disponibilidade, escalabilidade e segurança, e;
- Existe uma grande adesão à utilização dos *Web Services* como solução à disponibilização de serviços remotos, sobretudo ao estilo arquitectural REST que devido à sua simplicidade, tem ganho popularidade e relevância.

Estes factores apontam para o cenário apresentado na aplicação exemplo do estado da arte, onde o aspecto mais saliente é a relevância dos serviços, enquanto que todo o processo de construção de interfaces passa a ser abrangido pelas novas tecnologias RIA. A partir das secções anteriores sobre estado da arte e serviços é possível retirar justificações para uma nova framework:

1) Foi verificada a extensão e qualidade do suporte a serviços nas várias frameworks estudadas, onde foi concluído que nenhuma framework em particular se destaca. As frameworks actuais adicionaram suporte a BWS e em menor escala a REST como um aspecto opcional, não existindo nenhuma framework orientada a serviços;

2) As frameworks desenvolveram várias abordagens para a geração dinâmica e controle da interface HTML, a partir de JSP, JSF, rHTML[143], GSP, entre várias outras alternativas. A maioria dessas abordagens permite alterar a interface *on-the-fly* de forma a facilitar o desenvolvimento e diminuir o tempo necessário à sua implementação, mas o mesmo não acontece com o suporte aos serviços. Verifica-se uma adopção de abordagens iniciais ao desenvolvimento de serviços, havendo uma falta de abordagens mais refinadas;

3) A separação lógica e física das camadas de uma aplicação é uma abordagem bem estabelecida, tal como introduzido na apresentação da arquitectura MVC. Esta solução é utilizada sobretudo nas grandes aplicações industriais, onde existe uma infraestrutura própria para garantias de escalabilidade e disponibilidade, serviços específicos internos, e equipas qualificadas para gestão de todos estes sistemas. No entanto estas soluções requerem metodologias genéricas, flexíveis e robustas associadas a um grande custo e complexidade de implementação e manutenção, como por exemplo o *Enterprise Service Bus* (ESB)[54].

Pode ser questionada a eficiência da aplicação destas metodologias no desenvolvimento de aplicações de pequena e média dimensão num cenário marcado pelos factores anteriormente referidos. São então necessárias soluções mais simples e ágeis adaptadas a esta nova realidade, e;

4) Existe ainda uma lacuna em ferramentas e standards multi-plataforma para o desenvolvimento de aplicações REST.

Estes quatros pontos são a essência da fundamentação para este trabalho de investigação. Na

próxima secção é discutida o desenvolvimento de uma framework como abordagem escolhida para responder à questões acima identificadas.

4.3 Abordagem

Como verificado nas secções anteriores, existem várias lacunas na área de serviços REST, sobretudo no suporte em frameworks para aplicações web. Estas lacunas apresentam várias hipóteses de temas para trabalhos de investigação, desde o desenvolvimento de standards e especificações, à elaboração de metodologias para desenvolvimento, mecanismos de configuração dinâmica, ou ainda ferramentas auxiliares.

O desenvolvimento de uma nova framework adveio da vontade de elaborar esses temas num contexto de implementação de aplicações web centradas em serviços. Neste ambiente é possível não só criar especificações, metodologias, mecanismos e ferramentas mas também a sua implementação, testando desde já a sua validade e usabilidade.

A criação de uma framework completa conduz ao desenvolvimento de um conjunto de especificações, processos de desenvolvimento, algoritmos e ferramentas como servidores e editores que se tornam disponíveis para utilização, desenvolvimento e estudos posteriores. A framework constitui assim uma base sólida para fundamentação das estratégias elaboradas bem como um ponto de partida para novas iniciativas.

Esta abordagem é escolhida por ser considerada a mais adaptada a propor respostas válidas às questões identificadas na secção anterior. Na próxima secção será então feita a caracterização de uma framework para desenvolvimento de aplicações web centradas em serviços.

4.4 Características

Nesta secção define-se um conjunto de requisitos considerados essenciais a uma framework para desenvolvimento de aplicações web centradas em serviços REST.

Uma framework no contexto deste trabalho é definida como um ambiente de suporte ao desenvolvimento de aplicações web, constituído por diversos componentes. Neste trabalho é fornecido um servidor para desenvolvimento, alojamento e teste das aplicações criadas, definem-se metodologias à implementação das aplicações e disponibilizam-se ferramentas auxiliares como editores gráficos.

Os capítulos anteriores forneceram um conjunto de dados essenciais para a elaboração dos requisitos para esta framework. Destaca-se sobretudo a necessidade de uma framework orientada a serviços não como uma opção complementar, mas como ponto central do seu funcionamento.

Em primeiro lugar tem de ser explicada a arquitectura das aplicações a desenvolver nesta

framework. A arquitectura em camadas MVC foi descrita bem como os seus benefícios, e a sua utilização extensiva nas várias frameworks. Uma boa arquitectura em camadas exige um fraco acoplamento entre estas, de forma a flexibilizar a escolha de diferentes tecnologias de implementação das camadas. Assim sendo cada camada desenvolvida não só deve responder a uma série de requisitos, como deve abstrair as restantes camadas.

Devido ao foco nos serviços ao invés da interface, as aplicações a desenvolver seguirão a arquitectura SOFEA/SOUI, que define uma clara separação entre servidor e interface gráfica. A interface ou *frontend* é implementada em aplicações distintas do servidor, utilizando de preferência as novas tecnologias RIA. É sempre possível implementar uma interface típica em HTML ou um *fat client*, conforme as necessidades de usabilidade ou compatibilidade com as capacidades limitadas de certos dispositivos.

A utilização de SOFEA/SOUI traduz-se numa aplicação em camadas *Model Service Controller*, mas esta abordagem não pode prender os utilizadores desta framework a nenhuma implementação em particular de cada camada. É exigido que seja possível utilizar apenas a camada de serviços com qualquer tecnologia de controle ou de acesso a dados escolhida pelo programador. De forma similar, pode ser utilizado a camada de controle com outras abordagens para camada de serviços e persistência. Pode até ser usada a camada de controle sob uma interface gráfica em vez de serviços.

Por forma a estruturar a declaração dos requisitos, estes podem ser agrupados por camada funcional na aplicação. Assim sendo, de seguida são definidas características relativas à camada de serviços:

- A utilização da arquitectura SOFEA/SOUI enfatiza os serviços como *backend* às várias tecnologias de *frontend*. Foram vistas várias tecnologias de serviços, onde se destacam os Web Services e sobretudo a abordagem REST como a melhor escolha para pequenas e médias aplicações devido às razões enunciadas na secção de serviços. O estilo arquitectural REST é então escolhido como abordagem a utilizar na implementação de serviços para as aplicações a desenvolver na framework;
- A falta de ferramentas e standards de definição de serviços em REST leva a exigir que no âmbito desta framework seja feitos progressos significativos nessas áreas por forma a facilitar e uniformizar a definição de serviços em REST, e;
- As frameworks estudadas desenvolveram diferentes formas de dinamizar a implementação da interface gráfica em HTML, de modo a não ser necessário reiniciar a aplicação para aplicar alterações, tornando o ciclo de implementação e teste mais rápido. Este é um aspecto de usabilidade do programador essencial para a produtividade. Mas nas aplicações criadas com esta framework, a interface externa consiste em serviços, pelo que terão de ser desenvolvidas formas de permitir a criação e configuração dinâmica da camada de serviços.

No que se refere à camada de controle:

- Deve ser possível associar a cada serviço a execução de um elemento de controle, passando a este os dados necessários recebidos no pedido. Este elemento processa os dados e retornará uma resposta a ser enviada pela camada de serviços;
- Adicionalmente o processo de associação de um serviço a um elemento de controle deve ser simples e rápido;
- Os elementos de controle devem suportar alterações dinâmicas à sua implementação, por forma a facilitar o processo de desenvolvimento;
- Os elementos de controle deverão suportar algum tipo de encadeamento de acções, seguindo o padrão *Chain of Responsability*, por forma a potenciar a reutilização de elementos, e;
- Deve ser suportado alguma forma de *Aspect Oriented Programming*, por forma a adicionar funcionalidade de forma transparente à execução dos elementos de controle, tais como logging e autenticação.

Em relação à camada de persistência:

- A API deve ser orientada a objectos, reflectindo a tendência da adopção desta abordagem por permitir uma aprendizagem mais fácil e utilização simples;
- A API de persistência deve abstrair a implementação específica utilizada, seja SGBD ou simples ficheiros XML;
- O sistema de persistência deve suportar a redefinição de classes *on-the-fly*. Esta capacidade permite ao programador alterar as classes do domínio da aplicação sem forçar ao reinício do servidor, e;
- O sistema de persistência deve suportar *refactoring* sem perda de dados.

Características gerais da framework:

- O objectivo comum a muitas aplicações consiste em disponibilizar capacidades de acções CRUD a estruturas de dados, definidas como classes quando a plataforma é orientada a objectos. Frameworks como Ruby on Rails, Grails, Seam e outras têm a capacidade de gerar aplicações CRUD a partir de um conjunto de classes que definem o domínio do problema. Esta framework tem de fornecer um mecanismo semelhante de geração de aplicação a partir de classes que permita criar, editar e apagar objectos através dos serviços;
- As aplicações assim geradas têm de fornecer um conjunto básico de funcionalidades. Por exemplo, a aplicação terá de suportar acções CRUD via serviços REST com representação de dados por XML. Estas capacidades são muito comuns na comunicação com o frontend;
- O processo de geração deve ser genérico por forma a que o programador possa configurar todos os seus aspectos. Por exemplo, modificar a representação suportada de XML para JSON, ou modificar os tipos de serviços ou a solução de persistência gerados;

- A framework deve suportar conceitos avançados tais como injecção de dependências, *Aspect Oriented Programming* entre outros, disponibilizando as suas funcionalidades aos programadores avançados. Aos programadores iniciantes tais pormenores devem ser ocultados;
- Sempre que possível aplicar o princípio de *Convention over Configuration* por forma a reduzir o esforço e complexidade de configuração das aplicações;
- As acções mais comuns durante a criação, a configuração e a implementação de aplicações devem ser fáceis e simples, tendo em conta sempre a usabilidade do ponto de vista do programador e a sua necessidade de soluções ágeis;
- Reduzir a utilização de XML nos processos de configuração. Nos casos excepcionais, fornecer ferramentas gráficas que removam o esforço de criação manual do XML;
- A aplicação deve ser o mais dinâmica possível, suportando vários tipos de alterações sem ser necessário reiniciar;
- As aplicações devem poder ser desenvolvidas de forma modular, e;
- As camadas da aplicação devem ser fracamente acopladas de forma a possibilitar a utilização de outras tecnologias, sem forçar alterações extensas.

Esta caracterização revela os atributos gerais que a framework proposta deverá possuir.

4.5 Conclusão

A partir dos capítulos de estado da arte e de serviços foi possível obter caracterizar os cenários actual e futuro da implementação de aplicações web. Estes resultados mostraram uma expansão da utilização de serviços, sobretudo do tipo REST, que as frameworks actuais não suportam extensivamente. As lacunas nesta área deram origem a um trabalho cuja proposta foi apresentada neste capítulo.

A proposta, em primeiro lugar, fundamentou a sua razão de ser a partir dos dados obtidos nos capítulos anteriores. De seguida delineou a abordagem a adoptar para o desenvolvimento desse projecto, nomeadamente, a construção de uma framework para desenvolvimento de aplicações web centradas em serviços REST. Finalmente foram descritos vários requisitos para essa framework, bem como para as aplicações nela desenvolvidas.

É com base nesta proposta que no próximo capítulo se introduzem especificações por forma a solucionar os problemas identificados.

5 Especificação

5.1 Introdução

A partir da proposta do capítulo anterior, foram desenvolvidas abordagens ao desenvolvimento de serviços REST que se espera que resolvam os problemas identificados nesta área, e satisfaçam os requisitos identificados para este trabalho.

Um desses problemas é a falta de ferramentas e especificações para serviços REST e sobretudo a falta de soluções multi plataforma. Os requisitos delineados requerem um trabalho que solucione efectivamente estes aspectos. Para esse fim, foram criadas abordagens inovadoras que são descritas neste capítulo.

No entanto, a necessidade de fornecer soluções portáveis requer que estas sejam descritas de forma abstracta à sua implementação. Levanta-se então a questão de como introduzir essas abordagens sem forçar restrições de plataformas. Uma forma possível seria a apresentação de arquitecturas, mas uma arquitectura já apresenta uma forma de solução, um guia para a estrutura da implementação.

Ao contrário, pretende-se apenas descrever essas soluções por forma a que possam ser implementadas de diversas formas em várias plataformas mantendo sempre o mesmo funcionamento. Esse objectivo é realizado com a introdução de especificações.

As especificações descrevem a funcionalidade pretendida a alto nível das camadas MSC adoptadas pelo servidor de aplicações, sem induzir a uma única forma de implementação. No entanto, para ilustrar uma possível arquitectura, a solução desenvolvida para este projecto é detalhada no capítulo de implementação.

A framework a desenvolver não adopta as abordagens existentes noutras frameworks em relação à geração de interfaces HTML, escolhendo a opção distinta de suportar apenas serviços REST. Devido a esta quebra com o modelo standard de desenvolvimento de aplicações web e a sua criação de raiz, a framework denomina-se Tábula, a partir da expressão em latim Tabula Rasa, que significa 'quadro em branco'.

Este capítulo é está organizado da seguinte forma: em primeiro lugar é introduzida a especificação relacionada com a configuração e administração dos serviços REST. De seguida é apresentada uma solução para a camada de controle com suporte a AOP e finalmente é descrita a API do sistema de persistência.

5.2 Serviços

A camada de serviços é baseada no estilo arquitectural REST por razões apresentadas nos capítulos anteriores, mas que podem ser resumidas à sua facilidade de aprendizagem e utilização. No entanto, existem ainda várias dificuldades nesta área, não só relacionadas com a generalidade dos Web

Services, mas também especificamente com REST. Essas dificuldades foram expostas no capítulo sobre serviços, e podem ser resumidas na seguinte lista:

- Não existe um standard comum aceite pela comunidade para descrever serviços REST;
- Os princípios REST são independentes da plataforma, mas as ferramentas actuais são específicas a certas plataformas;
- O desenvolvimento dos serviços requer muitos testes que levam a várias alterações, quase sempre requerendo o reinício do servidor. Não existe pois um mecanismo que suporte a alteração dinâmica dos serviços, e;
- Não existe descrição de funções de administração remota, independentes de plataforma que permitam aplicar alterações aos serviços dinamicamente.

Estas questões exigem um trabalho que crie uma solução genérica e reutilizável de forma a beneficiar a área de suporte ao desenvolvimento de serviços REST.

O projecto de uma aplicação REST exige a organização do serviços, o que implica em várias actividades, tais como escolher que recursos expôr, que relações existem entre recursos e como identificá-los por URIs, que métodos HTTP, que formas de representação suportar, etc.

As próximas secções apresentam uma especificação para declarar e configurar a camada de serviços REST de um servidor. É importante neste momento distinguir uma especificação REST de um servidor REST. Um servidor REST no contexto deste trabalho é definido como uma aplicação que providencia uma implementação da especificação descrita.

5.2.1 Organização de Serviços REST

No capítulo de serviços pudemos verificar que os URIs são parte integral dos serviços REST, e que é composto de secções estáticas e parâmetros, e podem ser utilizados em casos comuns.

É evidente que numa aplicação com vários recursos expostos bem como os seus atributos, existe um grande número de URIs correspondentes. Para que o desenvolvimento e manutenção dessa aplicação se mantenham eficientes, é necessária alguma forma de organização desses URIs.

Tal como um URI identifica um único recurso, um conjunto de recursos logicamente relacionados pode ser identificado como um módulo. Por exemplo, na aplicação de documentação no capítulo de serviços, existiria um Módulo de Autores onde os recursos seriam os autores, os seus contactos telefónicos, a morada, e lista de documentos criados, constituindo o seguinte conjunto de URIs:

/autores/{id}
/autores/{id}/contacto/{ctc-id}
/autores/{id}/morada
/autores/{id}/documentos/{doc-id}

No módulo de Documentos, os recursos seriam os documentos, os seus capítulos e lista de autores:

/documentos/{doc-id}
/documentos/{doc-id}/capitulos/{cap-id}
/documentos/{doc-id}/autores/{aut-id}

Como ilustrado, cada módulo agrupa vários recursos fortemente relacionados entre si. Há sem dúvida uma associação entre Autores e Documentos, mas essa relação é intermodular, reflectida no uso de referências como visto anteriormente. De forma similar, numa outra aplicação de Vendas pode existir um Módulo de Clientes, onde os recursos são os clientes, os seus contactos, os endereços e as encomendas efectuadas. O módulo de facturação, por sua vez, consistiria no conjunto de recursos relativos a facturas, a recibos, e respectivas encomendas.

Conclui-se então que uma aplicação é composta por um ou mais módulos. No exemplo da aplicação de documentação, esta é constituída pelo módulo de Autores e de Documentos. Na aplicação de Vendas, passa a existir um módulo de Cliente e outro de Facturação.

A organização de serviços pode ser então visualizada de modo geral e a alto nível na Figura 6:

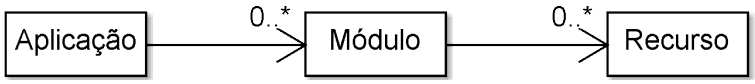

Figura 6: Diagrama de classes para a organização de alto nível de uma aplicação REST

No entanto, a funcionalidade requerida para uma aplicação REST no contexto deste trabalho depende de outros aspectos, tais como os métodos específicos ao protocolo HTTP e as formas de representação identificadas por tipos MIME. Para cada recurso da aplicação, para além do seu URI, é necessário definir o conjunto de métodos HTTP suportados, e a cada método, que formas de representação serão aceites (Figura 7).

Figura 7: Diagrama de classes para a organização a baixo nível de uma aplicação REST

Como as acções GET, PUT e POST lidam directamente com representações de recursos, é necessário listar as formas de representação suportadas, tais como XML, JPEG, PDF, entre outras,identificadas por um tipo MIME. A acção DELETE, por sua vez, consiste em remover o recursos, pelo que não requer declaração de tipos de representação suportados.

O conjunto de elementos necessários à organização dos serviços de uma aplicação REST pode então ser representado na Figura 8:

Figura 8: Diagrama de classes para a organização de uma aplicação REST

Na próxima secção será discutido o processamento dos pedidos.

5.2.2 Processamento de pedidos

Em relação à implementação, é necessário identificar e associar o código que realmente processa o pedido HTTP a um determinado recurso. A esse código é fornecido um conjunto de parâmetros necessários à sua execução e finalmente o pedido pode ser processado.

Esta abordagem, no entanto, é simplista, uma vez que o processamento a efectuar não só depende do recurso, como também do tipo de pedido HTTP e formato MIME especificado. Um pedido GET a um recurso para obter uma representação *plain/text* resultará num processamento diferente de um pedido PUT com conteúdo *application/xml* a esse mesmo recurso.

O processamento em regra geral não é linear, mas sim depende dos factores acima, decidindo-se então por um ou outro caminho de processamento conforme o pedido for GET, POST, PUT e DELETE. De forma similar, existirão diferentes caminhos de processamento conforme o formato a utilizar na representação do recurso.

Se a solução de serviços REST a implementar neste trabalho for limitada a associar código ao URI do recurso, então cabe ao programador criar toda a infraestrutura restante necessária para alterar o processamento conforme o tipo de pedido e formato. Basicamente o que é pedido é uma forma de declarar uma árvore de processamento simples, onde o caminho escolhido depende das características do pedido a processar.

Como esta é uma necessidade básica para aplicações com serviços REST, então esta framework tem de apresentar uma solução a este problema. Afinal o objectivo de uma framework é responder às necessidades dos casos comuns de utilização, ou seja, às tarefas e necessidades comuns dos programadores.

Outro aspecto a considerar é a reutilização de código. Considere-se o caso comum em que a um pedido GET está associado código que obtém dados da camada de persistência e é criada uma representação em XML do recursos. Este código exacto, se genérico o suficiente, pode ser reutilizado para todos os pedidos GET em formato XML, independentemente de que recursos recebam os pedidos, requerendo apenas a alteração de apenas alguns parâmetros conforme o recurso.

A solução proposta resolve estes problemas definindo uma sequência de processamento em três etapas, como ilustrado na Figura 9.

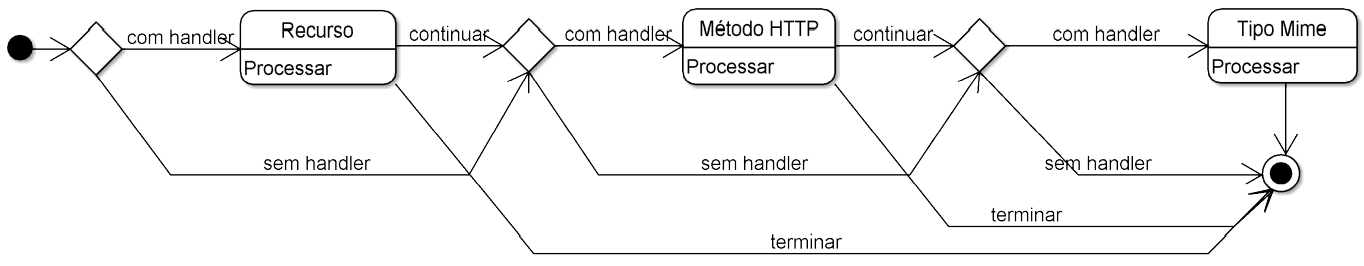

Figura 9: Sequência de processamento de pedido

Em cada etapa é possível associar um *handler*, definido como qualquer código que processe parte ou todo o pedido. Este código pode retornar de imediato uma resposta e terminar a sequência, ou apenas efectuar algum processamento e deixar que a sequência continue.

O comportamento pré-definido da sequência é executar o *handler* do recurso caso exista, então passar ao *handler* do método HTTP do pedido efectuado, e finalmente determinar o tipo MIME especificado e o respectivo *handler*. É possível então associar *handlers* a todas ou apenas em algumas etapas da sequência. Se o pedido for efectuado a um recurso, método ou tipo MIME não suportado, é retornado um erro correspondente na resposta.

De forma a esta abordagem funcionar, é requerido que todo o código nos *handlers* tenha acesso a um conjunto específico de informações cruciais à sua execução, tais como o pedido e resposta HTTP. É também requerido alguma forma de passar dados entre os *handlers* em diferentes etapas da sequência de processamento.

Esta especificação não força uma linguagem de programação específica ou plataforma, permitindo o uso de linguagens de *scripting* ou compiladas. Portanto é possível implementar servidores REST compatíveis com qualquer plataforma ou linguagem de escolha.

5.2.3 Configuração

A configuração é baseada em XML uma vez que é um padrão que proporciona a interoperabilidade sobre diferente plataformas, é legível tanto por pessoas como por máquinas, e existem várias ferramentas disponíveis.

Como uma aplicação é composta por vários módulos, pode então ser definida como :

```
<application name="CRM" active="true">
   <relative-uri>app</relative-uri>
   <modules>
      <module name="clients" />
      <module name="billing" />
   </modules>
</application>
```

O atributo *active* indica se a aplicação está activa ou não. Caso não esteja, todos os módulos e recursos associados não estarão acessíveis a pedidos.

Adicionalmente, a aplicação define um atributo opcional *relative-uri*, que pode ser utilizado de forma a definir um *namespace* específico à aplicação. Se a aplicação estiver alojada em *http://xpto.org*, então neste exemplo, todos os recursos possuirão no seu URI absoluto, o prefixo */app* : *http://xpto.org/app/costumer/1*

O módulo consiste num conjunto de vários recursos relacionados, como definido de seguida

```
<module name="clients" active="true">
   <relative-uri>test</relative-uri>
   <resources>
      <resource>…</resource>
      <resource>…</resource>
   </resources>
</module>
```

O atributo *active* determina se o módulo está actualmente activo no servidor, enquanto que o atributo *relative-uri* é utilizado para definir um *namespace* específico a todos os recursos deste módulo. É de referir que o URI absoluto de um recurso é construído tendo em conta o *relative-uri* da aplicação e do módulo. Neste exemplo, todos os recursos do módulo *client* possuirão um prefixo de */app/test* nos seus URIs: *http://teste.com/app/test/costumer/1*.

Os recursos são declarados da seguinte forma:

```
<resource active="true">
   <target-uri>/customers/{id}</target-uri>
</resource>
```

O URI do recurso é indicado no atributo *target-uri*, enquanto que o atributo *active* indica se o recurso está disponível a pedidos ou não. O URI neste caso é constituído pela secção estática */customers/* e pelo parâmetro *{id}*, sendo o URI absoluto de *http://xpto.org/app/test/customers/{id}*. Qualquer pedido efectuado a esse URI, como por exemplo *http://xpto.org/app/test/customers/15*, seria direccionado a este recurso. No entanto, nada é executado sem a este recurso estar associado um *handler* por forma a lidar com o pedido, como no exemplo seguinte:

```
<resource active="true">
   <uri>/customers/{id}</uri>
   <handler active="false">
      <path>/path/resourceHandler.groovy</path>
      <parameters>
         <parameter name="X" value="5" />
      </parameters>
   </handler>
</resource>
```

O *handler* define o caminho para o ficheiro com o código a ser executado em caso de pedidos para este recurso. Neste exemplo é indicado um ficheiro de *script* em *Groovy*. É permitido ainda definir uma lista de pares *nome-valor* a serem utilizados como parâmetros pelo *handler*. Esta abordagem permite reutilizar o mesmo *handler* em diferentes recursos mudando apenas alguns parâmetros.

Por exemplo, se o *handler* acede a uma tabela de uma base de dados relacional, obtém uma linha dessa tabela e retorna uma representação da mesma, essa funcionalidade pode ser utilizada tanto para um recurso de clientes como para um recurso de empregados de outro módulo.

Basta generalizar o código de forma a depender de um parâmetro "*tabela*" que indica o nome da tabela relacional de onde obter a linha identificada pelo parâmetro *{id}* do pedido. No caso do recurso *customers*, pode então estar definido o seguinte:

```
<resource active=”true”>
   <uri>/customers/{id}</uri>
   <handler active=”false”>
      <path>/path/resourceHandler.groovy</path>
      <parameters>
         <parameter name=”tabela” value=”clients_table” />
      </parameters>
   </handler>
</resource>
```

Enquanto que para o recurso empregados seria:

```
<resource active=”true”>
   <uri>/employees/{id}</uri>
   <handler active=”false”>
      <path>/path/resourceHandler.groovy</path>
      <parameters>
         <parameter name=”tabela” value=”employees_table” />
      </parameters>
   </handler>
</resource>
```

Esta abordagem permite definir parâmetros diferentes utilizados como variáveis pelo código por forma a ser reutilizado.

Existe, no entanto, alguns problemas associados à forma como o caminho foi indicado nos exemplos anteriores. A utilização de barras '/' depende da plataforma utilizada, impede a indicação de ficheiros existentes em bibliotecas JAR, e explicita a forma de implementação deste *handler.*

Considere o cenário onde este mesmo ficheiro de *script* é reutilizado extensivamente numa aplicação, referido em vários *handlers*, e surge a necessidade de alterar a implementação deste *script* para uma classe Java por motivos de performance. Esta alteração levaria a ter de modificar todas as referências ao *script* por uma referência a uma classe Java.

Outro caso possível seria querer utilizar a mesma configuração, mas noutra plataforma, onde os *handlers* são implementados em PHP. Se a configuração especifica a extensão do ficheiro, então requer uma alteração extensa em todas as referências nos *handlers*.

Estes problemas podem ser resolvidos adoptando uma notação especifica, denominada *Fully Qualified Class Name* (FQCN)[60]. Esta notação foi criada por forma a referir de forma absoluta às classes de uma aplicação que residem em pacotes estruturados. Por exemplo, uma classe Teste, definida num ficheiro *Teste.java* de um pacote *exemplo*, que por sua vez reside no pacote *xpto*, terá um FQCN de: *xpto.exemplo.Teste*.

Com esta notação, o uso de pontos para denotar uma hierarquia de pacotes pode ser utilizado para denotar uma hierarquia de directórios em qualquer plataforma. Por outro lado, não é indicada a extensão do ficheiro, o que permite abstrair a implementação específica. Seguindo o exemplo anterior, o caminho */path/resourceHandler.groovy* pode ser representado como

path.resourceHandler. Cabe ao servidor procurar por ficheiros e classes identificadas pelo FQCN para processar o pedido.

No entanto, se existir uma classe Java com este FQCN, bem como um ficheiro numa estrutura de directórios também representada por este FQCN, reside o problema de determinar qual das soluções utilizar no processamento do pedido.

É da responsabilidade do servidor convencionar uma ordem de precedências. Numa plataforma Java, é lógico que um *handler* em Java tenha precedência sobre um *handler* em Groovy, por exemplo.

Então a forma correcta de indicar o handler será:

```
<resource active="true">
   <uri>/employees/{id}</uri>
   <handler active="false">
      <path>path.resourceHandler</path>
      <parameters>
         <parameter name="tabela" value="employees_table" />
      </parameters>
   </handler>
</resource>
```

Voltando ao caso em que é necessário modificar a implementação do *handler* de *script* Groovy para Java, se os caminhos forem indicados via FQCN e o servidor priorizar uma implementação em Java a uma implementação em Groovy, então basta criar a classe Java e disponibilizar à aplicação. Nenhuma outra alteração é necessário, mantendo-se a configuração dos serviços.

Quando um pedido for recebido, o servidor determina as implementações existentes e escolhe aquela que possui prioridade. Neste exemplo, o *handler* passa a executar uma classe Java em vez de um *script* Groovy. Esta solução permite portanto alterar a implementação, sem modificar a configuração, pelo que é mais simples, flexível e requer menos esforço de manutenção.

A sequência de processamento suporta a associação de *handlers* a métodos HTTP e tipos MIME, como demonstrado no exemplo seguinte.

```
<resource active="true">
   <target-uri>/customers/{id}</target-uri>
   <handler active="true">
      <path>path.resourceHandler</path>
   </handler>
   <http-methods>
       <http-method name="GET" active="true">
         <handler active="true">
            <path>path.getHandler</path>
         </handler>
         <mime-type-handlers>
            <mime-type-handler>
              <mime-type>application/xml<mime-type>
              <handler active="true">
```

```
                <path>path.GetXML</path>
              </handler>
            </mime-type-handler>
            <mime-type-handler>
               <mime-type>application/pdf</mime-type>
               <handler active="true">
                  <path>path.GetPDF</path>
               </handler>
            </mime-type-handler>
         </mime-type-handlers>
      </http-method>
      <http-method name="DELETE" active="true">
         <handler active="true">
            <path>path.DeleteHandler</path>
         </handler>
      </http-method>
   </http-methods>
</resource>
```

Neste caso um *handler* é especificado para o recurso, mas para os métodos GET e DELETE são associados outros *handlers*. Esta abordagem permite colocar código comum a ambos os métodos no *handler* do recurso. Este será executado independentemente do método do pedido.

O recurso possui uma lista de métodos HTTP suportados, definidos com o atributo *active*, de forma a indicar se estão disponíveis ou não. Os métodos HTTP possuem ainda uma lista de *mime-type-handlers,* que associam um tipo MIME a um *handler* correspondente. Esta abordagem granular de associar *handlers* com tipos MIME específicos, permite dividir o problema, permitindo que se isole e desenvolva de forma independente o código para criação de um tipo de representação específico.

O exemplo anterior define a localização da implementação para os vários *handlers* associados ao recursos, aos métodos GET e DELETE, e aos tipos MIME *application/xml* e *application/pdf*. O servidor com base nesta especificação executa os *handlers* em sequência, conforme o pedido. Os caminhos de processamento possíveis para este recursos pode ser visualizados na Figura 10.

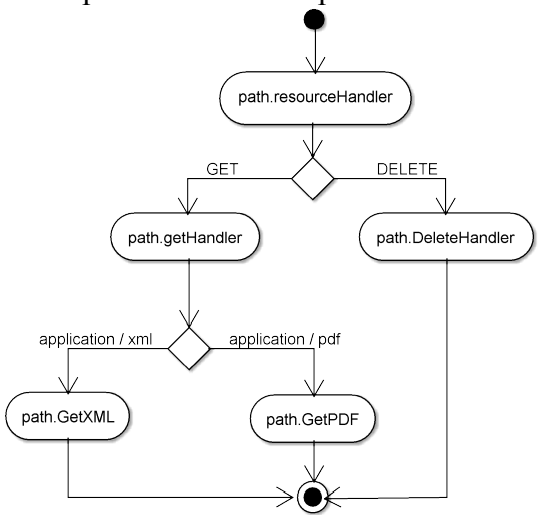

Figura 10: Exemplo de árvore de decisão de processamento

Desta maneira a especificação disponibiliza ao programador uma infraestrutura de suporta à definição de *handlers* conforme as características do pedido.

Como o protocolo HTTP define outros métodos para além de GET, POST, PUT e DELETE e é um protocolo extensível, esta especificação suporta ainda a extensão de outros métodos HTTP através do atributo *name* do *http-method*.

A Figura 11 mostra um exemplo de configuração de aplicações REST com esta especificação:

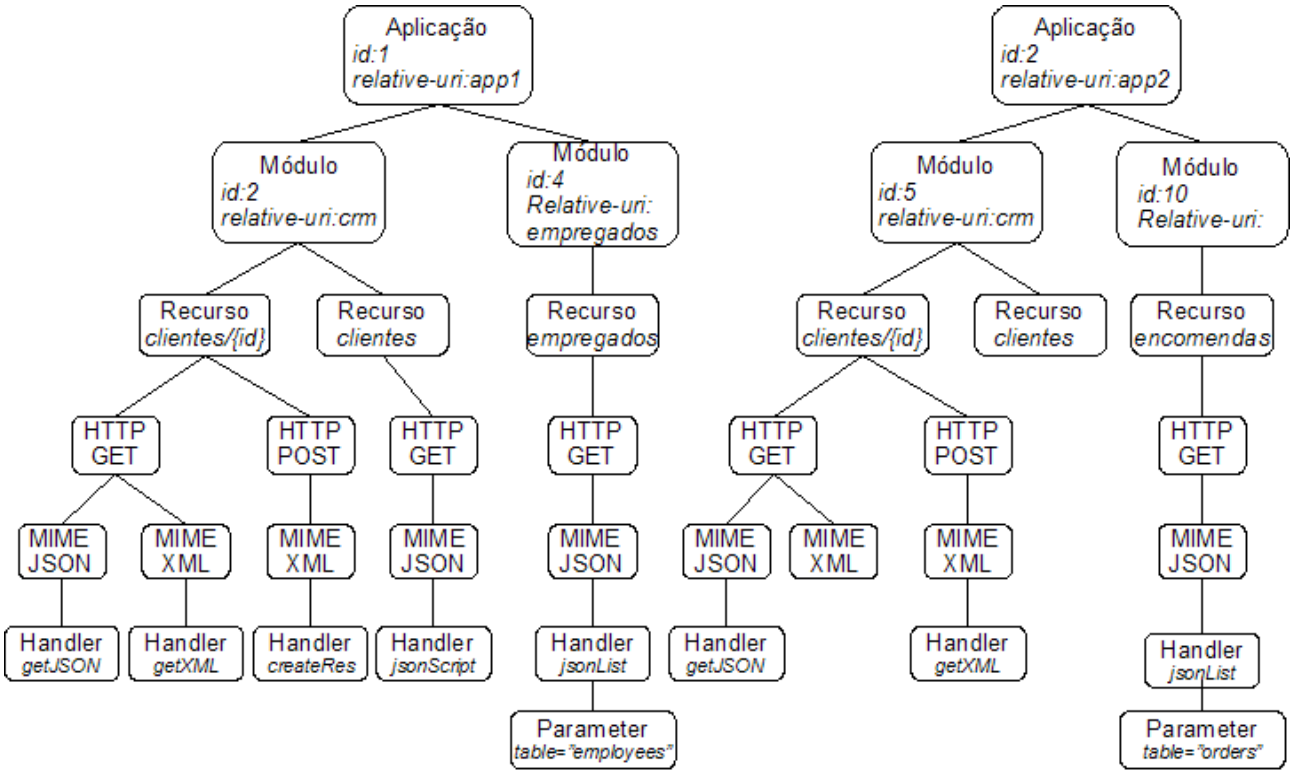

Figura 11: Exemplo de configuração de aplicações REST

A elaboração desta especificação permite definir a camada de serviços REST para aplicações. Na próxima subsecção será apresentado um serviço de administração remota desta configuração.

5.2.4 Administração REST

Um dos obstáculos ao desenvolvimento de aplicações centradas em serviços é a necessidade de reiniciar a aplicação sempre que alguma alteração é efectuada à definição da camada de serviços. De forma a resolver esta questão, e permitir que os serviços sejam modificados *on-the-fly*, é definido um serviço de administração remota da configuração REST.

O conceito base consiste em expôr a própria configuração REST como um recurso, permitindo que seja alterada através de pedidos HTTP com conteúdo em XML, forçando a actualização dos recursos disponíveis. O serviço de administração é baseado na especificação da configuração REST, pelo que a sua utilização torna-se relativamente simples.

Considere o caso de um servidor REST em *http://xpto.com/*. A aplicação de administração pode, por exemplo, utilizar o URI base de *http://xpto.com/admin/rest*. Como visto na especificação da

configuração, o elemento de topo é a aplicação. Os recursos do tipo Aplicação podem estar acessíveis no URI *http://xpto.com/admin/rest/applications*. Para obter uma lista das aplicações existentes, é enviado um pedido GET ao URI das aplicações, o que retorna o seguinte conteúdo:

```
<paged-list page="1" item-count="2" items-per-page="20">
     <application id=”1” name=”crm” active=”true”
     uri=”http://xpto.com/admin/rest/applications/1” />
     <application id=”2” name=”test” active=”false”
     uri=”http://xpto.com/admin/rest/applications/2” />
</paged-list>
```

É obtida uma lista das aplicações disponíveis, com alguns atributos base e os *hyperlinks* que localizam cada aplicação. Devido ao comprimento dos URIs absolutos, nos exemplos seguintes todos os URIs indicados passam a ser relativos ao URI base *http://xpto.com/admin/rest*.

Pode ser obtida informação adicional sobre a primeira aplicação através de um GET ao URI *http://xpto.com/admin/rest/applications/1*, o que retorna:

```
<application id=”1” name=”crm” active=”true” uri=”…/applications/1”>
   <modules>
      <module name=”clients” active=”true” uri=”…/modules/1” />
      <module name=”billing” active=”true” uri=”…/modules/2” />
   </modules>
</application>
```

O serviço de administração utiliza conteúdo semelhante ao da especificação da configuração REST, com duas excepções essenciais: todos os recursos possuem um atributo *uri* e a referência a outros recursos é feita por meio de uma representação reduzida, sempre indicando o URI desse recurso. Assim é permitido que se obtenha informação de forma incremental em vez de receber toda a configuração. De forma similar, um GET a *http://xpto.com/admin/rest/modules/1* obtém:

```
<module id=”1” name=”xp” active=”true” uri=”…/modules/1”>
   <relative-uri>expo</relative-uri>
   <resources>
      <resource id=”1” active=”true” uri=”…/resources/1”/>
         ...
      <resource id=”n” active=”true” uri=”…/resources/n”/>
   </resources
</module>
```

É indicada a lista de recursos abrangidos pelo módulo, novamente numa forma reduzida e indicando a sua localização pelo atributo *uri*. A informação dos recursos, por sua vez segue o seguinte exemplo:

```
<resource id=”1” active=”true” uri=”…/resources/1”>
   <target-uri>/test/clients</target-uri>
   <handler uri=“…/handlers/34” />
   <http-methods>
     <http-method name=”GET” active=”true” uri=”…/http-methods/4”/>
   </http-methods>
</resource>
```

E em relação ao *handler* localizado em http://xpto.com/admin/handlers/34, seria obtido:

```
<handler active="true" uri="…/handlers/34" >
   <path>filepath.to.getClientList</path>
   <parameters>
      <parameter name="X" value="245" uri="…/parameters/56"/>
   </parameters>
</handler>
```

Pedidos adicionais retornariam informação sobre o *http-method*, os seus *mime-type-handlers* e respectivos *handlers*, seguindo a mesma orientação de usar representações simplificadas com URIs quando se refere a outros recursos.

Devido ao grande número de recursos que podem ser referidos numa lista, é aceite que se faça uma representação reduzida da própria lista. Por exemplo, um *handler* com dezenas de parâmetros pode ser representado da seguinte forma:

```
<handler active="true" uri="…/handlers/34" >
   <path>filepath.to.getClientList</path>
   <parameters>
      <parameter name="X" value="245" uri="…/parameters/56"/>
       <parameter name="y" value="1" uri="…/parameters/67"/>
       ...
       <parameter name="d2" value="123" uri="…/parameters/33"/>
       <parameter name="ex" value="56" uri="…/parameters/235"/>
   </parameters>
</handler>
```

Devido ao grande número de parâmetros, o conteúdo da resposta é extenso. Aceita-se também a seguinte representação mais concisa:

```
<handler active="true" uri="…/handlers/34" >
   <path>filepath.to.getClientList</path>
   <parameters item-count="53" />
</handler>
```

Assim refere-se apenas à dimensão da lista, reduzindo o conteúdo da resposta. De forma a obter a lista em si, envia-se um GET a *http://xpto.com/admin/rest/handlers/34/parameters*, o que retorna:

```
<paged-list page="1" item-count="53" items-per-page="20"
next="http://xpto.com/admin/rest/handlers/34/parameters?size=20&amp;page=2"
last="http://xpto.com/admin/rest/handlers/34/parameters?size=20&amp;page=3" >
      <parameter name="X" value="245" uri="…/parameters/56"/>
       <parameter name="y" value="1" uri="…/parameters/67"/>
       ...
       <parameter name="a17" value="123" uri="…/parameters/33"/>
       <parameter name="a18" value="56" uri="…/parameters/235"/>
</paged-list>
```

Esta lista paginada indica o número da página visualizada, o número total de itens na lista, o número de itens por página e de seguida, os *hyperlinks* para a próxima e a última páginas. Neste *hyperlinks* é especificado o número de itens e a página da lista a visualizar. Se o pedido inicial for *http://xpto.com/admin/rest/handlers/34/parameters?size=10* então o resultado será:

```
<paged-list page="1" item-count="53" items-per-page="10"
next="http://xpto.com/admin/rest/handlers/34/parameters?size=10&amp;page=2"
last="http://xpto.com/admin/rest/handlers/34/parameters?size=10&amp;page=6" >
      <parameter name="X" value="245" uri="…/parameters/56"/>
       <parameter name="y" value="1" uri="…/parameters/67"/>
```

```
        ...
        <parameter name="a7" value="123" uri="…/parameters/21"/>
        <parameter name="a8" value="56" uri="…/parameters/644"/>
    </paged-list>
```

O cliente pode especificar o número de itens a mostrar em cada página, o que modifica o número total de páginas necessárias para representar toda a lista. Esta abordagem aplica-se a todas os atributos do tipo lista, como por exemplo, a lista de métodos de um recurso, ou a lista de recursos de um módulo.

Este mecanismo de paginação é similar ao utilizado nas pesquisas em bases de dados relacionais. Por exemplo, a resposta a uma *query* em SQL "*SELECT ... FROM ... LIMIT 0, 10"* retornaria 10 registos a partir do primeiro, ou então *"SELECT ... FROM ... LIMIT 20, 10"* para retornar 10 registos a partir do vigésimo registo.

Constitui assim uma solução genérica ao problema das listas de grande dimensão, apresentando uma interface uniforme para a sua paginação.

É importante referir que todos os atributos de cada topo de recursos também são expostos. Ou seja, se é pretendido obter o *relative-uri* de um módulo, basta enviar um GET para, por exemplo : *http:// xpto.com/admin/rest/modules/1/relative-uri*. A resposta terá um conteúdo do tipo texto com o valor do atributo *relative-uri*, que neste caso é "*expo*". Da mesma forma, pode-se saber se uma aplicação está activa ou não, através de um GET a *http://xpto.com/admin/rest/applications/2/active*, que retorna "*false*". Ou ainda saber qual o valor de um parâmetro com um GET a *http://xpto.com/admin/rest//parameters/56/value*, que retorna "*245*".

O serviço de administração tem de permitir não só ler a configuração, mas também editá-la. O processo de edição de nova informação é simples. Por exemplo, criar um parâmetro consiste apenas em efectuar um POST a *http://xpto.com/admin/rest/parameters* com um conteúdo do tipo application/xml de:

```
<parameter name="Y" value="31"/>
```

A resposta teria o código HTTP 201 *Created*, indicando o sucesso da criação e conteria a representação de um parâmetro recém-criado, com o atributo *uri* definindo a sua localização:

```
<parameter name="Y" value="31" uri=".../parameters/635"/>
```

A adição de um parâmetro a uma lista de parâmetros de um handler pode ser feito de forma semelhante. O POST seria efectuado ao URI http://xpto.com/admin/rest/handlers/34/parameters com um conteúdo do tipo XML: <parameter name="tabela" value="clientes"/>

A resposta novamente teria o código HTTP 201 *Created*, indicando o sucesso da criação e conteria a representação actualizada do handler, já listando o novo parâmetro:

```
<handler active="true" uri="…/handlers/34" >
   <path>filepath.to.getClientList</path>
   <parameters>
```

```
        <parameter name="X" value="245" uri="…/parameters/56"/>
         <parameter name="tabela" value="clientes" uri="…/parameters/636"/>
      </parameters>
   </handler>
```

Pode ser também adicionado um parâmetro já existente. Neste caso o conteúdo do POST é do tipo texto e indica a localização do parâmetro :

```
http://xpto.com/admin/rest/parameters/635
```

O servidor efectua então a associação do parâmetro já existente com a lista de parâmetros do handler, retornado a nova localização:

```
<handler active="true" uri="…/handlers/34" >
   <path>filepath.to.getClientList</path>
   <parameters>
      <parameter name="X" value="245" uri="…/parameters/56"/>
       <parameter name="tabela" value="clientes" uri="…/parameters/636"/>
        <parameter name="Y" value="31" uri="…/parameters/635"/>
   </parameters>
</handler>
```

A modificação de recursos existentes também é simples. Por exemplo, desactivar um módulo consiste em efectuar um PUT a *http://xpto.com/admin/rest/modules/1/active* com um conteúdo textual de *false*. O servidor processa este pedido desactivando todos os recursos associados àquele módulo.

O remover de um parâmetro leva, por exemplo a enviar um pedido de DELETE ao URI *http://xpto.com/admin/rest/parameters/635*. Este parâmetro também seria retirado de todos os *handlers* que o referenciavam nas suas listas de parâmetros.

O serviço de administração expõe toda a informação de configuração, bem como os seus atributos, permitindo uma abordagem granular à edição da camada de serviços. Qualquer alteração efectuada força a actualização dos recursos expostos *on-the-fly*, sem ser necessário reiniciar o servidor, tornando o processo de desenvolvimento e manutenção mais rápido e simples.

5.2.5 REST EDITOR

A utilização de XML e pedidos HTTP para definir a configuração da camada de Serviços REST pode se tornar uma tarefa árdua e pouco eficiente se efectuada manualmente. De forma a retirar esse esforço aos programadores, foi criada uma aplicação em SWING capaz de utilizar o serviço de administração REST, tal como apresentado na subsecção anterior.

Esta aplicação é denominada de Editor REST/ REST Editor [176] e funciona como um *frontend* ao sistema de administração, permitindo a configuração dos serviços através da interface gráfica. Esta abordagem permite automatizar a criação de XML e o envio de pedidos.

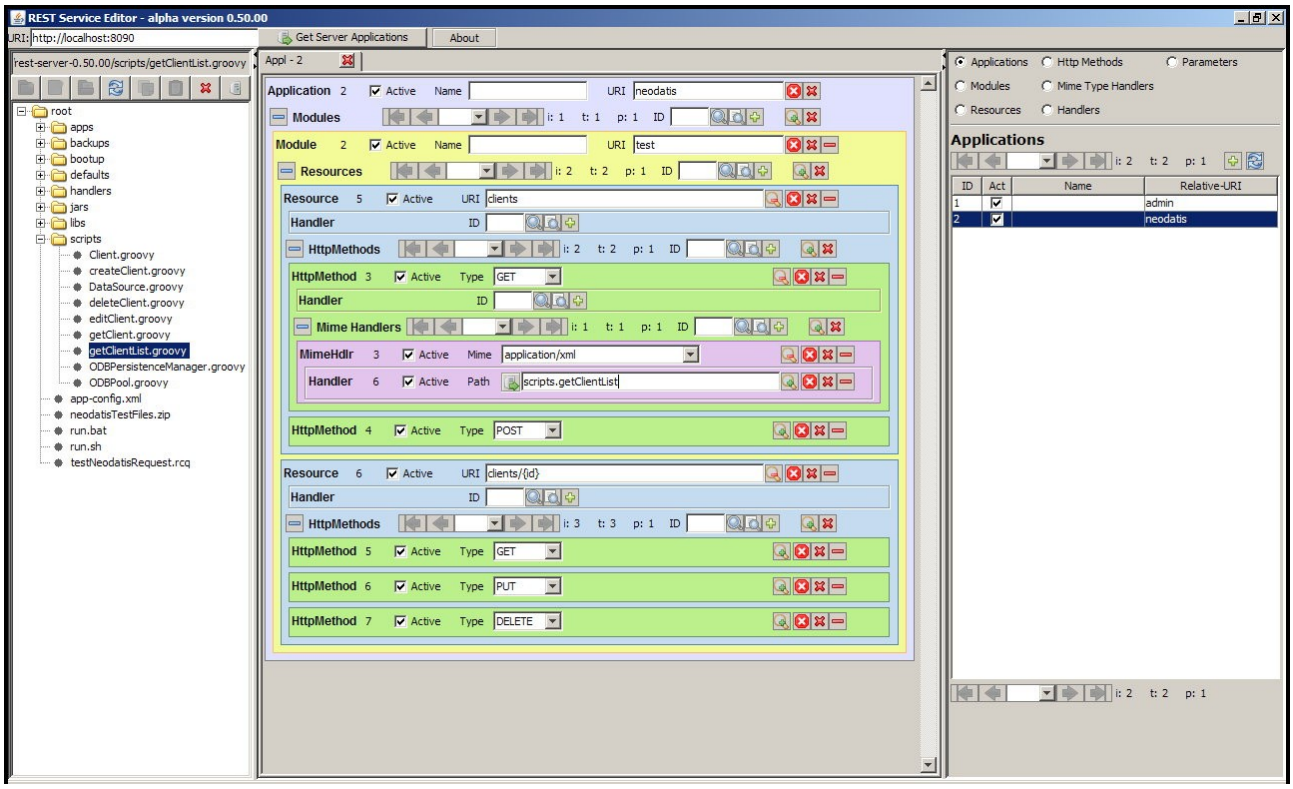

Figura 12: Interface Gráfica do Editor REST

Qualquer modificação à configuração efectuada no editor é imediatamente enviada ao serviço de administração no servidor, forçando a actualização dos serviços. Desactivar um recurso ou um módulo completo pode ser efectuado com um simples clique sobre uma *checkbox*.

Como visto na Figura 12, o editor é composto por 4 partes:

- No topo está acessível o endereço do serviço de administração no servidor. É a partir deste URI que os pedidos são efectuados aos vários tipos de recursos necessários à configuração;
- À esquerda existe um explorador de ficheiros, que permite a funcionalidade típica tal como criar, mover e remover directórios e ficheiros;
- Na direita está disponível um explorador da configuração REST. Este componente permite visualizar os vários tipos de recursos da configuração disponíveis no servidor indicado, e;
- Ao centro está a área principal de edição. Suporta várias abas – *tabs* – em que cada uma apresenta a interface gráfica de um recurso da configuração REST.

A descrição pormenorizada da funcionalidade do editor seria extensa, pelo que no âmbito deste trabalho é feita uma descrição geral da sua arquitectura e utilização. Em [184] estão disponíveis tutoriais vídeo demonstrando a utilização do Editor para configuração da camada de serviços REST.

A integração de um explorador de ficheiros no editor resulta da necessidade de indicar a localização de ficheiros de *script* nos *handlers* da configuração, como apresentado anteriormente. De forma a evitar que os programadores criem manualmente o FQCN correspondente ao caminho do ficheiro, o

explorador permite a selecção do ficheiro e na interface de configuração do *handler* basta clicar sobre o botão à esquerda do campo de texto *path*. O FQCN é então criado e colocado no *path*, enviando simultaneamente um pedido de actualização ao servidor.

A integração do explorador permite ainda que se trabalhe sem sair do editor sobre a estrutura de directórios, o que é frequente durante o desenvolvimento, sobretudo na organização dos *scripts* e outros recursos. Adicionalmente, o explorador depende da implementação de uma interface, *FileManager* para acesso aos sistema de ficheiros.

Esta abordagem permite não só uma implementação via API de ficheiros do Java com acesso ao sistema de ficheiros local, mas também permite a implementação de um *FileManager* por FTP[65], com acesso remoto ao sistema de ficheiros do servidor. Assim, o editor pode fornecer, na mesma interface, as capacidades de edição remota não só da configuração REST mas também do sistema de ficheiros.

Este editor permite toda a gama de acções necessárias à configuração da camada de serviços REST, desde a criação dos objectos - *application*, *module*, *resource*, *http-method*, *mime-type-handler*, *handler* e *parameter* - à activação ou desactivação dos mesmos, à modificação dos vários atributos e a configuração da sequência de processamento do pedido.

Como ferramenta, retira aos programadores todo o esforço necessário à criação e ao envio do XML, fornecendo uma interface gráfica simples de utilizar e capaz de efectuar toda a configuração necessária. A especificação da configuração e esta ferramenta permite que equipas de programadores trabalhem em conjunto e de forma paralela em módulos e/ou recursos distintos, modificando facilmente a camada de serviços da aplicação.

Como a especificação do serviço de administração é independente da plataforma, é possível implementar esse serviço em qualquer linguagem de programação. Desde que a administração siga a especificação, é então possível utilizar o editor, independentemente da plataforma ou linguagem do servidor. Adicionar suporte a esta especificação significa assim ter acesso a uma ferramenta gráfica comum para configurar os serviços REST.

Na próxima secção introduz-se uma solução para a camada de controle, especialmente adaptada ao processamento de pedidos REST.

5.3 Controle

5.3.1 Introdução

A associação de ficheiros ou classes como *handlers* à sequência de processamento de pedidos foi exposta na secção anterior através da especificação de configuração. Embora nos exemplos dados os *handlers* consistissem em implementações em Java ou Groovy[73], a especificação em si abstrai

estes detalhes, pelo que qualquer outra solução pode ser utilizada.

Embora *handlers* em Java e Groovy sejam úteis o suficiente para que se possa desenvolver serviços *stub*, um protótipo ou até uma pequena aplicação, estes não respondem a todos os requisitos identificados, nomeadamente abordagem *chain of responsibility*[223], que permitiria flexibilizar a definição dos *handlers* e a reutilização de secções de código comum. Quer o Groovy quer o Java também não fornecem mecanismos próprios para *Aspect Oriented Programming*, estando dependentes de bibliotecas adicionais que requerem manipulação do código compilado. Assim, qualquer alteração a um *aspect* força o reinício do servidor.

Foi então desenvolvida uma abordagem específica, a partir de uma análise ao processamento dos pedidos. Esta secção inicia-se com a descrição da análise efectuada, seguida da apresentação da solução desenvolvida e finalmente introduz-se uma possível abordagem a AOP.

5.3.2 Estrutura

O processamento de um pedido recebido pela camada de serviços passa, de uma forma geral, por três estados: preparação, execução e finalização.

No estado de preparação é verificado se o pedido é válido ou não, o que depende de aspectos tais como autenticação, validade de conteúdo, existência e validade de parâmetros. Também são alocados recursos necessários ao processamento do pedido.

O estado de execução processa os passos essenciais à funcionalidade do pedido, utilizando os recursos alocados anteriormente e recorrendo às APIs necessárias tais como persistência. No estado de finalização são libertados os recursos alocados durante os estados anteriores.

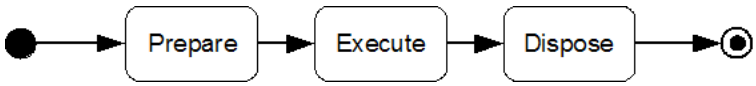

Figura 13: Diagrama de estados de processamento de pedido

Esta vista geral descrita na Figura 13 pode ser refinada através de uma análise mais detalhada de cada estado. Nos exemplos e diagramas desta secção será utilizada uma nomenclatura em inglês de forma a facilitar o reconhecimento dos estados na especificação a ser descrita na próxima secção.

O estado de preparação é composto por sub estados: teste, alocação de recursos e validação de recursos. O sub estado de teste efectua a validação do pedido verificando uma série de elementos tais como existência ou não de conteúdo e parâmetros.

O sub estado de alocação requisita e obtém os vários recursos necessários à execução, e finalmente o sub estado de verificação efectua testes de forma a validar o estado de todos os recursos e dados essenciais antes de iniciar o processamento do pedido. A Figura 14 detalha esta estrutura.

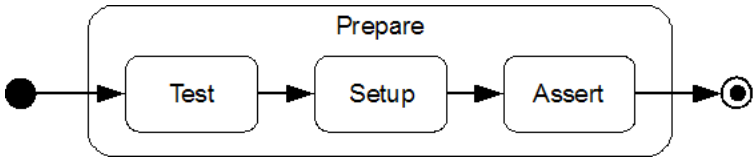

Figura 14: Composição do estado de preparação

Considere o exemplo da edição de um recurso Cliente numa aplicação REST, através de um pedido PUT a um URI fictício de *http://xpto.org/clientes/42*. A validade desse pedido pode ser verificada em dois passos:

1. Verificar que o pedido contém dados em XML, e;
2. Testar a existência de um cliente com o *id* de 42 através da API de persistência.

Considere agora o caso de um GET ao mesmo recurso. No processo de verificação o teste ao conteúdo do pedido é irrelevante, uma vez que é pretendida uma representação do cliente e não uma alteração dos seus dados. Neste caso a verificação consiste apenas em :

3. Testar a existência de um cliente com o *id* de 42 através da API de persistência.

Destas duas situações conclui-se que existem pequenas secções de código comuns a vários pedidos distintos. Seguindo o princípio de *Don't Repeat Yourself*[45], seria desejável que o programador implementasse essa secção de código uma única vez, mas que o pudesse reutilizar várias vezes.

Essa solução torna-se possível se cada estado for constituído por vários passos, em que cada passo se refere a uma secção de código que pode ser reutilizada em vários outros estados. Então visualiza-se o estado de preparação da seguinte forma (Figura 15):

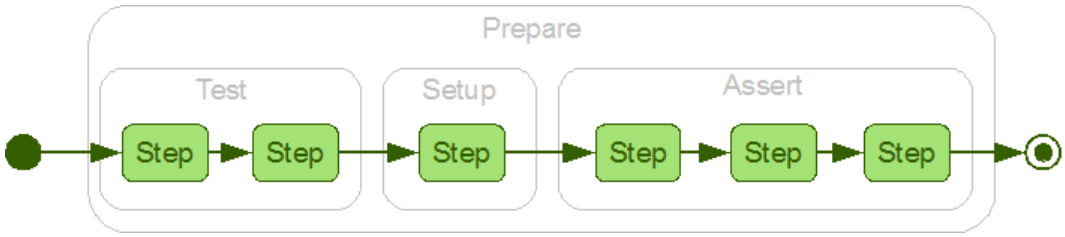

Figura 15: Composição dos estados como sequência de passos

Só é possível transitar do estado de preparação para o estado de execução se o processamento de todos os passos for bem sucedido. Cada passo corresponde a um elemento de código capaz de efectuar algum processamento. Este elemento pode ser implementado em várias linguagens de programação, pelo que a definição da estrutura deve ser abstrair a implementação específica, mas simultaneamente, permitir a sua identificação, tal como efectuado na secção anterior sobre configuração da sequência de processamento REST.

É de referir, no entanto, que esta estrutura não força, por exemplo, que durante o estado de execução não sejam feitos testes. O objectivo é estruturar o código de forma a detectar o mais cedo possível os casos inválidos e terminá-los, para que não utilizem desnecessariamente recursos e tempo de

processamento nas etapas posteriores. Então o estado de preparação deve agrupar um conjunto de testes iniciais que determinam se a máquina de estados pode ou não transitar para a execução essencial desta acção.

A máquina de estados finita que efectua o processamento de um pedido consiste numa Acção. A estrutura desta máquina reflecte a análise do processamento dos pedidos, que indica os estados e transições mais comuns. Esta análise levou a criar o seguinte esquema presente na Figura 16:

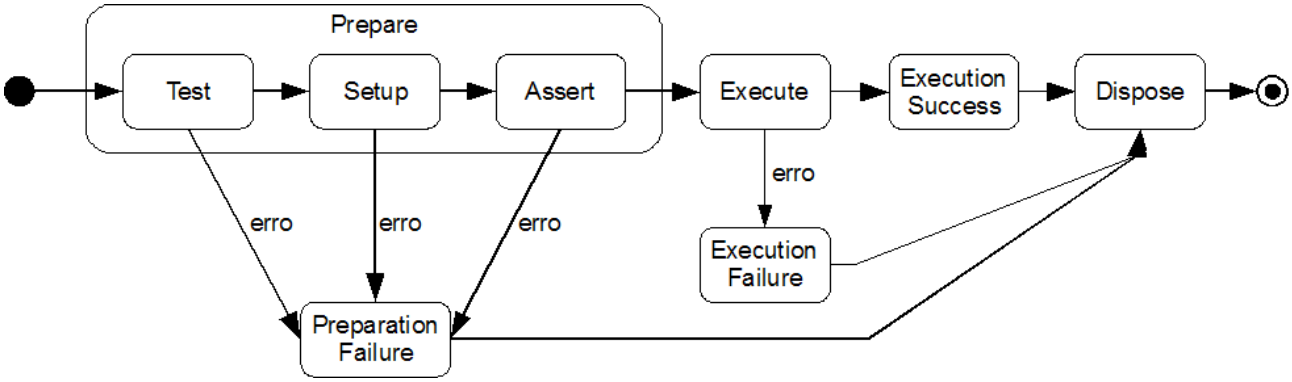

Figura 16: Diagrama de máquina de estados finita para a acção de processamento de pedido

O diagrama define um conjunto de estados e de transições principal que corresponde à execução com sucesso da acção. Pode ser visto que no caso de erro é feita uma transição para um estado específico. No caso de erro durante a preparação, existe o estado de *preparation failure* e no caso de erro durante a execução, existe o estado de *execution failure*.

Esta abordagem permite criar código para lidar com erros em estados distintos, uma vez que em cada etapa é executado código que modifica o estado dos recursos disponíveis. Assim pode ser requerido que se resolvam de forma diferente esses erros do modo a manter o sistema num estado consistente.

5.3.3 Configuração

Em relação à configuração, existem várias bibliotecas de máquinas de estados e frameworks de *Workflow[197]* tais como Enhydra Shark[51], ou State Machine Compiler [161] com grande capacidade e flexibilidade que permitem definir uma máquina de estados finita correspondente ao diagrama acima. No entanto, essas soluções apresentam vários problemas:

- Se a definição da máquinas de estados é efectuada por código, a solução força o reinício da aplicação quando há alterações;
- Se a definição é efectuada via ficheiros XML, estes apresentam uma estrutura longa e complexa, de difícil aprendizagem;
- Algumas soluções possuem ferramentas gráficas por forma a facilitar a configuração manual de XML. No entanto estas ferramentas ainda são complexas, expondo uma interface pouco simples devido às várias funcionalidades disponíveis;

- As capacidades destas soluções requerem a aprendizagem de conceitos adicionais relativamente complexos antes da sua utilização eficaz, e;
- A utilização de *Aspect Oriented Programming* nestas soluções não é trivial, e não se adapta facilmente ao processo de desenvolvimento de aplicações REST.

Estas dificuldades levam ao desenvolvimento de uma solução própria, pois para o âmbito deste trabalho é essencial fornecer aos programadores uma forma de estruturar e reutilizar secções de código conforme os estados e transições mais comuns. Deve ser fornecido ainda uma forma simples de especificar os passos para cada estado dessa máquina de estados e suportar alguma forma de *Aspect Oriented Programming*.

As situações mais complexas, que se reflectem noutra estrutura de estados e transições, podem então recorrer às várias bibliotecas de *worflow* e ferramentas associadas, pois a solução desenvolvida neste trabalho não impede de qualquer forma a sua utilização.

Novamente recorre-se ao XML como forma de definição dos passos em cada estado da Acção. Um exemplo de uma acção simples pode ser definida como :

```
<action>
  <test>
       <step source="scripts.TestRequest" />
  <test>
  <execute>
      <step source="scripts.CreateResponse" />
  </execute>
</action>
```

Neste caso é definida uma acção em que apenas o estados de *test* e *execute* possuem passos. Dada a necessidade de abstrair a configuração da implementação específica de cada passo, mas simultaneamente conseguir identificar o elemento respectivo, adopta-se a mesma abordagem utilizada na configuração dos serviços REST.

Ou seja, o atributo *source* do elemento *step* indica o FQCN correspondente ao elemento de código, seja uma classe Java, um ficheiro Groovy ou PHP, entre muitas outras possibilidades. Caso haja para o mesmo FQCN vários elementos implementados de forma diferente, cabe ao servidor definir uma ordem de prioridades e escolher que elemento utilizar nesse passo. O mecanismo de suporte a implementações deve ser desenvolvido por forma a ser relativamente simples adicionar outras soluções conforme desejado.

Esta abordagem permite recorrer, por exemplo, a *scripts* Groovy para o desenvolvimento rápido de cada passo da acção, e posteriormente, sem alterar a configuração, alterar a implementação para classes Java. Torna-se até possível utilizar na mesma acção, passos implementados de forma diferente. Descreve-se de seguida dois exemplo mais complexos : uma acção que processa pedidos GET a um recurso e retorna uma representação em XML do mesmo (Figura 17), e outra acção que processa um pedido de DELETE, removendo um recurso(Figura 18).

```
<action>
  <test>
       <step source="scripts.GetFourParameters" />
       <step source="scripts.GetObjectID" />
      <step source="scripts.GetObjectType" />
  </test>
  <setup>
      <step source="scripts.GetPersistenceManager" />
  </setup>
  <assert>
        <step source="scripts.GetObjectByID" />
  </assert>
  <preparation-failure>
        <step source="scripts.RollbackPersistenceManager" />
  </preparation-failure>
  <execute>
      <step source="scripts.ConvertObjectToXML" />
      <step source="scripts.SetXMLToResponse" />
  </execute>
  <execution-failure>
        <step source="scripts.RollbackPersistenceManager" />
  </execution-failure>
  <dispose>
      <step source="scripts.ClosePersistence" />
  </dispose>
</action>
```

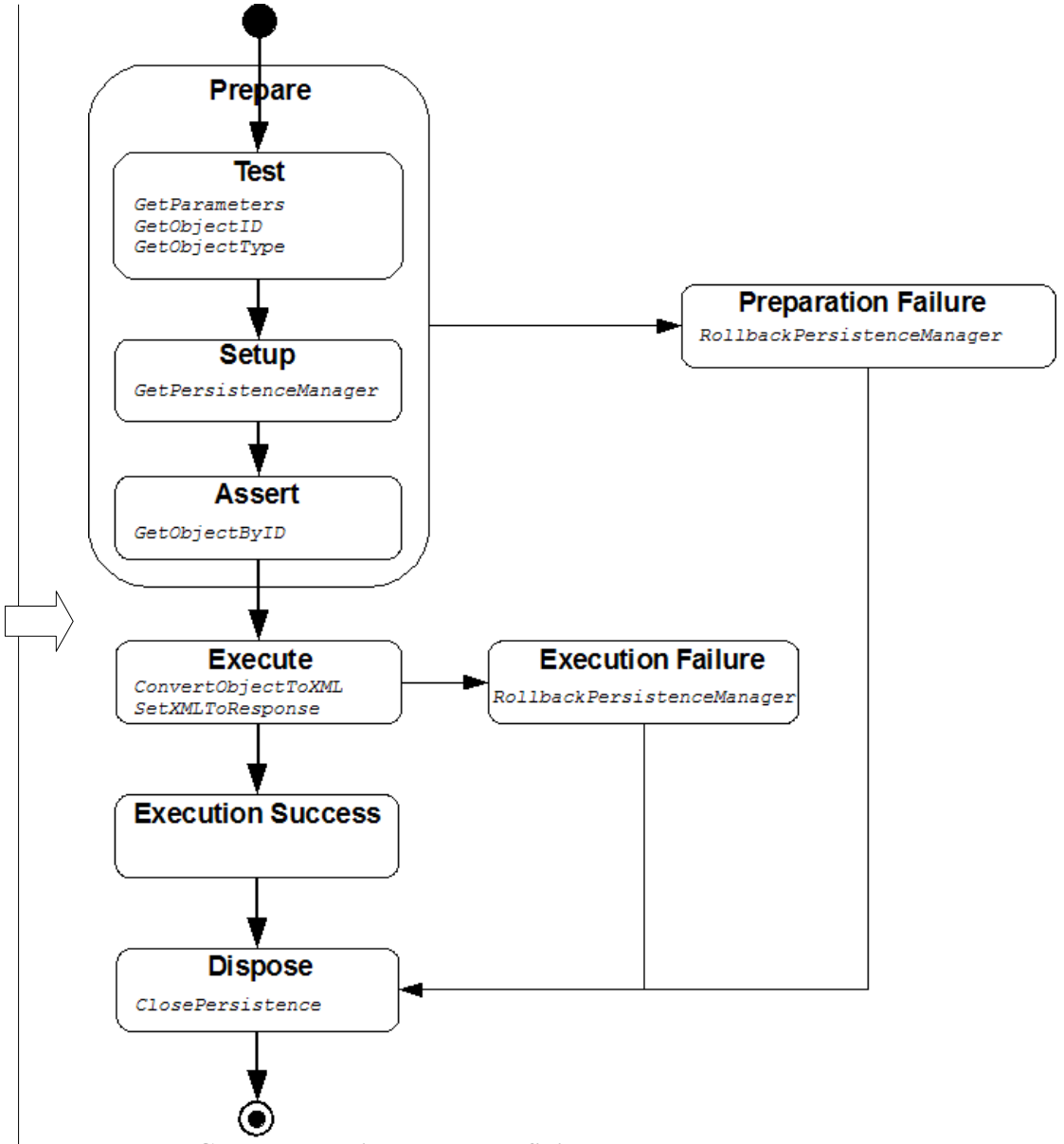

Figura 17: Definição em XML de uma Acção para processamento de GET e a máquina de estados finita correspondente

```
<action>
  <test>
       <step source="scripts.GetFourParameters" />
       <step source="scripts.GetObjectID" />
      <step source="scripts.GetObjectType" />
  </test>
  <setup>
      <step source="scripts.GetPersistenceManager" />
  </setup>
  <assert>
        <step source="scripts.GetObjectByID" />
  </assert>
  <preparation-failure>
        <step source="scripts.RollbackPersistenceManager" />
  </preparation-failure>
  <execute>
      <step source="scripts.DeleteObject" />
  </execute>
  <execution-failure>
        <step source="scripts.RollbackPersistenceManager" />
  </execution-failure>
  <dispose>
      <step source="scripts.ClosePersistence" />
  </dispose>
</action>
```

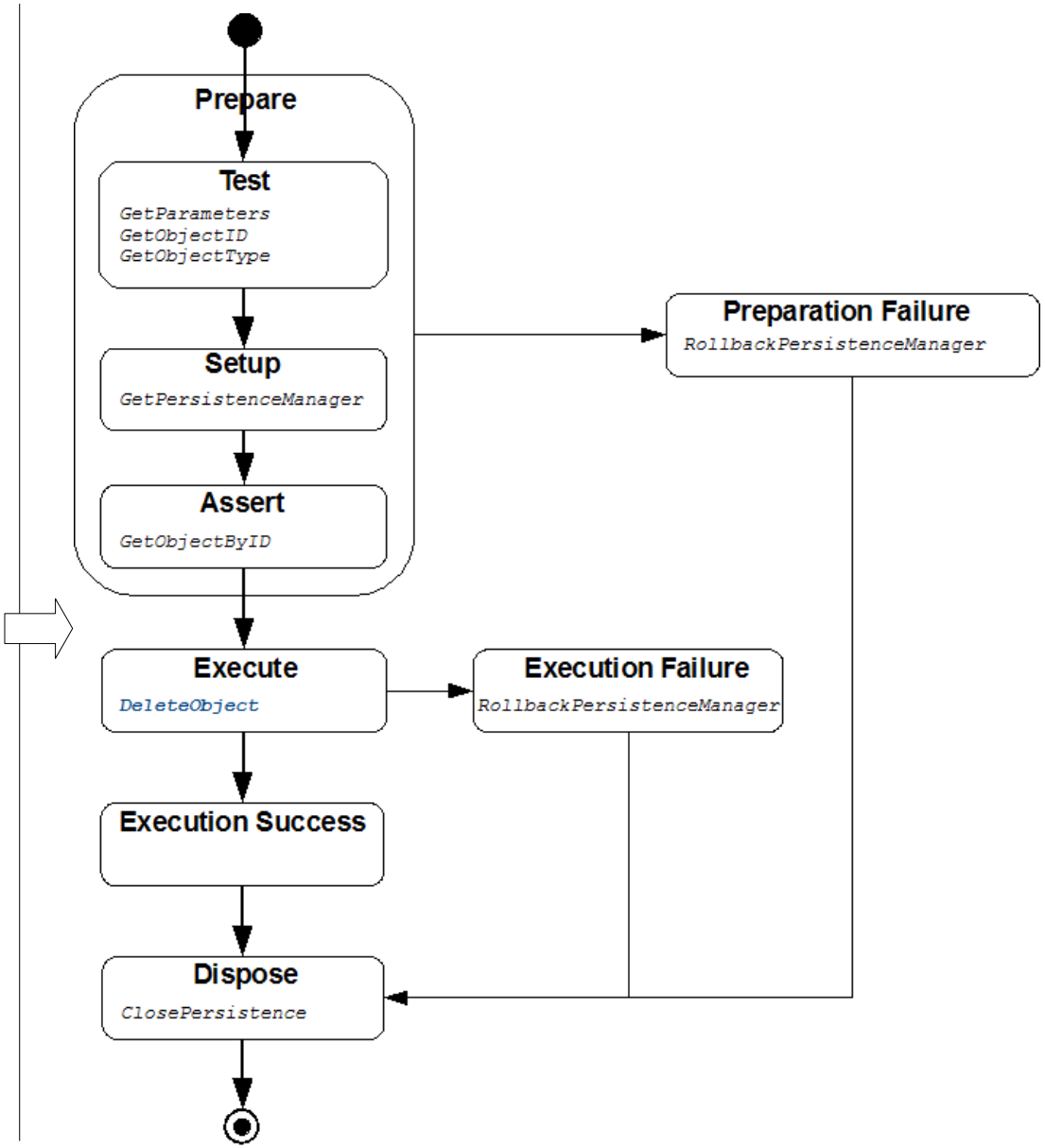

Figura 18: Definição em XML de uma Acção para processamento de DELETE e a máquina de estados finita correspondente

Os diagramas anteriores evidenciam que é possível reutilizar a maioria do código, pois as duas acções variam apenas no estado de execução. No caso do GET, o passo recorre ao elemento *scripts.ConvertObjectToXML* e *scripts.SetXMLToResponse* que cria uma representação em XML do recursos e coloca-o no conteúdo da resposta, enquanto que no caso do DELETE, o elemento *scripts.DeleteObject* remove o objecto através da API de persistência.

É visto também a adição de passos em caso de falha na preparação e execução, em que é feito o *rollback*, ou seja o cancelamento de qualquer modificação efectuada na solução de persistência.

Pode-se verificar que a grande maioria do código escrito é independente da implementação específica da camada de serviços, excepto os passos das etapas iniciais de obtenção de dados e teste do pedido bem como o passo final de configuração da resposta. Uma modificação da tecnologia da camada de serviços, por exemplo, de REST para *Big Web Services* com uso de SOAP requer apenas a alteração desses poucos passos, um processo relativamente simples e rápido, e que afectaria de imediato todas as acções onde é feita a sua reutilização.

A associação de acções à sequência de processamento de pedidos segue as condições descritas na secção anterior. Ou seja, se um *handler* pretende utilizar uma acção, então tem de se referir ao ficheiro XML que contém a definição dessa acção. Por exemplo, se ficheiro for /scripts/GetJSON.xml então o *handler* será configurado da seguinte forma:

```
<handler active="true" uri="…" >
   <path>scripts.GetJSON</path>
<handler>
```

Utiliza-se sempre o FQCN para representar o ficheiro, de forma a abstrair a implementação específica. Cabe ao servidor implementar a funcionalidade de a partir do FQCN no *path*, identificar o ficheiro XML com a descrição da acção de forma a executá-la.

O recurso a XML exige uma ferramenta gráfica de forma a retirar o esforço de criação e manutenção manual da configuração das várias acções. Esta ferramenta encontra-se em desenvolvimento e consiste num editor de acções que remove ao programador a necessidade de edição directa de XML.

A sub secção seguinte passa a descrever a solução desenvolvida para aplicação de AOP às acções.

5.3.4 Aspect Oriented Programming

O AOP é uma solução apelativa como visto no capítulo 2. As bibliotecas de AOP para Java tais como AspectJ requerem a manipulação do *bytecode*, ou seja, as classes são compiladas e então são alteradas por forma a inserir *proxys* que adicionem os *aspects* ao sistema. Implica-se então que alterações aos aspectos só podem ser aplicadas após reiniciar a aplicação, neste caso o servidor, pelo que não constituem soluções dinâmicas.

A framework desenvolvida não impede a utilização dessas soluções por parte de programadores avançados. No entanto, tendo em conta que o servidor é centrado da implementação de serviços, e no processamento de pedidos, seria útil possibilitar alguma forma de AOP adaptada a esse fim, cuja dificuldade de aprendizagem e utilização fosse inferior às soluções existentes.

A utilização de *jtActions* como máquinas de estado finitas para processamento de pedidos permite uma abordagem a AOP mais dinâmica. O AOP geralmente consiste na adição de *pointcuts* antes e/ou depois de determinados métodos, mas no caso de *jtActions*, a configuração não tem acesso aos detalhes da implementação de cada passo. No entanto, a configuração define estados aos quais estão associadas transições. Por exemplo, a transição do estado de preparação para o estado de execução pode ser visto como uma invocação de métodos:

```
prepare();
execute();
```

Com base neste ponto de vista, pode-se aplicar a metodologia de AOP em adicionar pointcuts antes e depois de cada métodos, como descrito de seguida:

```
beforePreparePointcut();
prepare();
afterPreparePointcut();
beforeExecutePointCut
execute();
afterExecutePointCut();
```

Aplicando esta lógica à máquina de estados de uma acção, obtém-se o diagrama da Figura 19:

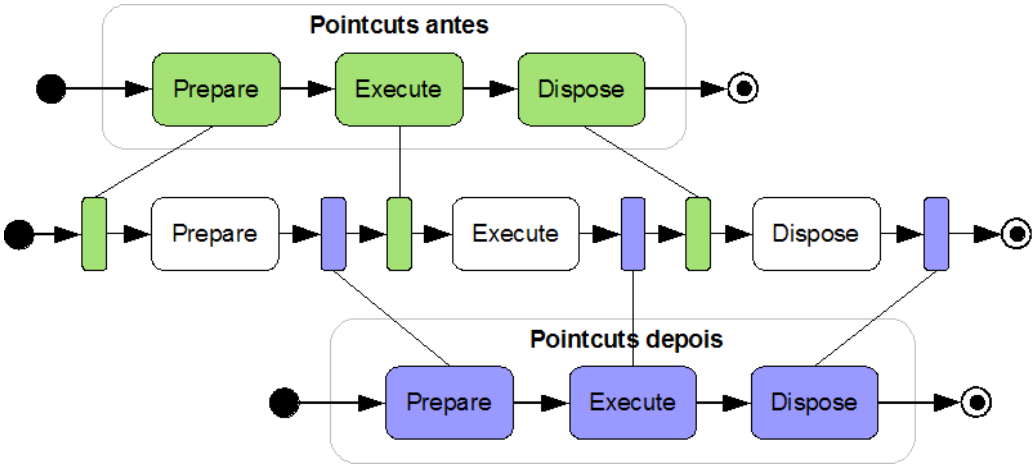

Figura 19: Definição de pointcuts como Acções

Ao agrupar desta forma os pointcuts, reconhece-se uma estruturação comum: o grupo de pointcuts antes de cada estados é definido de forma idêntica à própria *acção*. O mesmo acontece com o grupo de *pointcuts* a ser inserido após cada estrutura.

Portanto, definir os pointcuts consiste em definir *acções* que são inseridas antes e depois da *acção*-alvo. Como já foi definida uma forma de configuração das acções como declarado na secção anterior, essa especificação pode ser reutilizada aqui.

Refira-se que para efeitos de simplificação do diagrama, são esquematizados os *pointcuts* do estado geral de preparação em vez de todos os *pointcuts* para os estados de *test*, *setup* e *assert*, embora estes estejam disponíveis.

Por exemplo, pretende-se adicionar um aspecto de logging a uma acção que processa pedidos de DELETE. Para esse fim, define-se os pointcuts nas acções a adicionar antes e após cada estado do acção-alvo. O excerto seguinte define a pré-acção seguindo a especificação de configuração. Essa definição em XML é gravada no ficheiro com a seguinte localização: */scripts/aop/SignalStart.xml*

```
<action>
  <test>
      <step source="scripts.SignalStartTest" />
  </test>
  <execute>
      <step source="scripts.SignalStartExecution" />
  </execute>
</action>
```

O próximo excerto, por sua vez descreve a pós-acção, é gravado em : */scripts/aop/SignalFinish.xml*

```
<action>
  <test>
      <step source="scripts.SignalStopTest" />
  </test>
  <setup>
      <step source="scripts.SignalStopExecution " />
  </setup>
  <execute>
      <step source="scripts.SignalDisposed" />
  </execute>
</action>
```

Embora o termo *advice* tivesse sido declarado como referindo unicamente a funcionalidade adicionada, neste caso pode ser aplicado de outra forma. Note-se que a pré-*acção* e *pós-acção* definem que passos adicionar, ao que corresponde uma implementação. Então ao conjunto da pré e pós *acção* denomina-se *Advice*. Neste exemplo a definição em XML do *advice* seria :

```
<advice>
   <before>scripts.aop.SignalStart</before>
   <after>scripts.aop.SignalFinish</after>
</advice>
```

As *tags before* e *after* indicam a localização dos ficheiros XML que declaram a pré e pós *acção*, utilizando novamente a nomenclatura FQCN.

A aplicação deste *advice* leva a que os passos declarados na pré e pós *acção* sejam adicionados, respectivamente, antes e depois dos passos da *acção* principal, como visto na Figura 20. A acção resultante detém assim a funcionalidade da acção original e do *advice*.

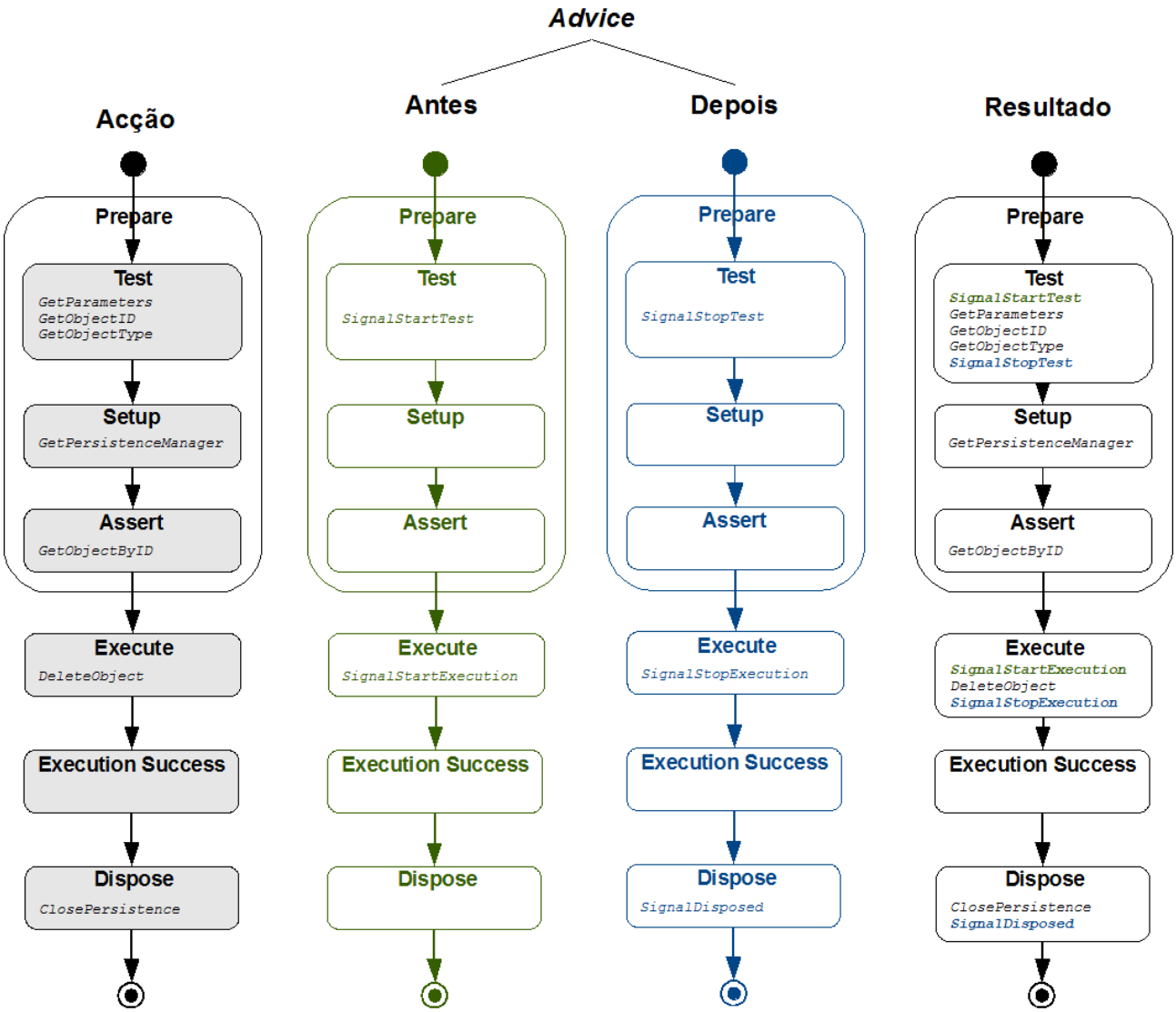

Figura 20: Exemplo da aplicação de um Advice a uma acção

Na próxima secção é apresentada uma forma de associar os *advices* às acções nas aplicações do servidor.

5.3.5 Regras de Aplicação

Visto que é possível definir uma solução de AOP para *acções*, resta o problema de associar os a*dvices* às *acções*. A solução é utilizar regras para definir o âmbito de aplicação do *advice*. Por exemplo, pretende-se adicionar um *advice* para avaliar a performance na execução de acções para:

- pedidos GET em formato JSON na aplicação com *id* de 1, e;
- pedidos POST em formato JSON para todas as aplicações.

Estas restrições podem ser definidas como uma lista de regras. A partir destas regras, o sistema deve decidir se aplica ou não o advice associado, alterando a acção a executar. As regras são definidas com base na configuração dos serviços REST, por forma a facilitar a sua integração com as aplicações a desenvolver.

O exemplo anterior pode ser mapeado para a seguinte definição:

```
<rules>
   <rule>
      <application>
         <long>1</long>
      </application>
      <http-method-names>
         <string>GET</string>
      </http-method-names>
      <mime-types>
         <string>application/json</string>
      </mime-types>
   </rule>
   <rule>
      <http-method-names>
         <string>POST</string>
      </http-method-names>
      <mime-types>
         <string>application/xml</string>
      </mime-types>
   </rule>
</rules>
```

A Figura 21 mostra as acções abrangidas pelas regras.

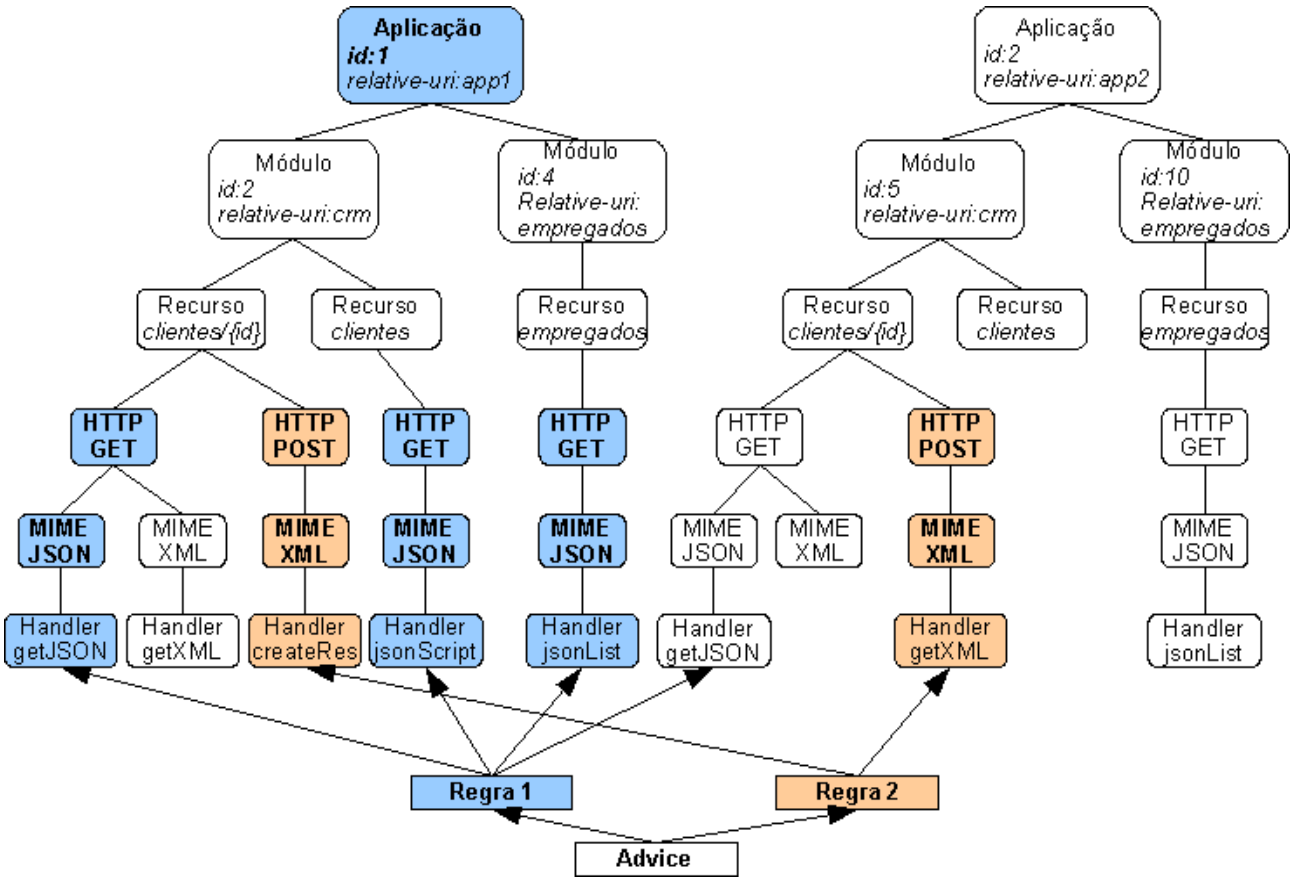

Figura 21: Exemplo de aplicação de lista de regras

Claramente, deve ser possível definir vários tipos de condições, pelo que as regras devem suportar

os atributos relevantes dos tipos utilizados para configuração da camada de serviços, tais como:

- Relative-uris de aplicações e módulos: <application-uris/> e <modules-uris>
- Target-uri de recrusos: <resource-uris/>
- Tipos de pedido HTTP: <http-method-names/>
- Tipos MIME: <mime-types/>
- Path dos handlers: <paths/>
- Identificação no serviço de administração de aplicações, módulos, recursos, http-methods, mime-type-handlers e handlers:<applications/>, <resources/>, <modules/>, <http-methods/>, <mime-type-handlers/> e <handlers/>

Com estas *tags* disponíveis para utilização da definição de regras, torna-se possível aplicar *advices* de forma tão abrangente ou específica quanto necessário. O anexo 3 pode ser consultado para outros exemplos das possibilidades deste formato.

O relacionar das regras com os *advices* é conseguida através do *aspect*. No contexto desta solução, o *aspect* é combinação de um *advice* com uma lista de regras, que define as condições da sua aplicação. Então a definição completa do caso anterior consistiria no seguinte excerto:

```
<aspect name=”logging” active=”true”>
   <advice>
       <before>scripts.aop.SignalStart<before>
       <after>scripts.aop.SignalFinish<after>
   </advice>
   <rules>
       <rule>
          <application>
             <long>1</long>
          </application>
          <http-method-names>
              <string>GET</string>
          </http-method-names>
          <mime-types>
              <string>application/json</string>
          </mime-types>
       </rule>
       <rule>
          <http-method-names>
              <string>POST</string>
          </http-method-names>
          <mime-types>
              <string>application/xml</string>
          </mime-types>
       </rule>
   </rules>
</aspect>
```

O atributo *name* permite definir resumidamente a função do aspecto de forma a ser reconhecido rapidamente pelos programadores aquando da visualização da configuração.

Esta especificação permite delinear de forma adaptada aos serviços REST as condições de aplicação dos *advices*. Pode ser argumentado que esta solução é específica a esta implementação da camada de serviços porque a definição das regras utiliza *tags* que dependem de termos da especificação REST.

É necessário ter em conta que a solução de AOP em si é abstracta desses factores. Nem a definição dos *advices* nem a implementação dos passos a adicionar dependem da camada de serviços. O componente *weaver* desta solução que processa a configuração das acções com os *advices* também não depende de forma nenhuma dos serviços. A única relação com o tipo de serviços é determinada pelas regras, o que é necessário uma vez que tratam da integração com essa camada de forma a garantir facilidade e flexibilidade de utilização.

5.3.6 Administração

De forma a potenciar uma utilização dinâmica da definição de aspectos, recorre-se à mesma abordagem utilizada em relação à definição dos serviços REST: disponibilizar um serviço de administração. O conceito é tratar todos os tipos necessários à configuração como recursos remotos disponíveis via serviços REST. Sempre que alguma alteração seja efectuada, o servidor deve aplicar imediatamente as modificações, de forma a garantir o dinamismo da administração dos aspectos.

Por exemplo, considere-se que o URI base do sistema de administração de aspectos é o seguinte: http://xpto.org/admin/aop/aspects. Pretende-se adicionar um aspecto de forma a testar a performance das acções efectuadas durante leitura, criação e edição de recursos, em que as representações são feitas em XML. Para este fim é efectuado um POST ao URI base de administração com o seguinte conteúdo em formato XML:

```
<aspect name=”perf test” active=”true”>
   <advice>
      <before>scripts.aop.performanceTestStart</before>
      <after>scripts.aop.performanceTestFinish</after>
   </advice>
   <rules>
      <rule>
         <http-method-names>
            <string>GET</string>
            <string>POST</string>
            <string>PUT</string>
         </http-method-names>
         <mime-types>
            <string>application/xml</string>
         </mime-types>
      </rule>
   </rules>
</aspect>
```

Deverá ser recebida uma resposta com o código HTTP 201 *Created* a indicar o sucesso da operação e como conteúdo terá a representação do *aspect* recém criado:

```
<aspect name=”perf test” active="true" id="3"
 uri="http://xpto.org/admin/aop/aspects/3">
   <rules item-count="1"/>
   <advice id="3" uri="http://xpto.org/admin/aop/advices/3"/>
</aspect>
```

O sistema não só deve criar o aspecto, como deve aplicá-lo imediatamente. Assim, quaisquer pedidos que correspondam à regra deste aspecto já devem ser processadas por acções com o *advice* aplicado.

A obtenção da lista de *aspects* pode ser feita por um GET a *http://xpto.org/admin/aop/aspects*, que retornará o seguinte conteúdo:

```
<paged-list page="1" item-count="3" items-per-page="20" >
   <aspect id=”1” active=”true” uri=”xpto.org/admin/aop/aspects/1”/>
   <aspect id=”2” active=”true” uri=”xpto.org/admin/aop/aspects/2”/>
   <aspect id=”3” active=”true” uri=”xpto.org/admin/aop/aspects/3”/>
</paged-list>
```

Acedendo directamente ao *aspect* recém criado pelo URI de *http://xpto.org/admin/aop/aspects/3* obtém-se:

```
<aspect id=”3” name=”perf test” active=”true”
 uri=”http://xpto.org/admin/aop/aspects/3”>
   <advice id=”3” uri=”...admin/aop/advices/3”/>
   <rules item-count=”2” />
</aspect>
```

A leitura do *advice* no URI *http://xpto.org/admin/aop/advice/3* retornaria:

```
<advice id=”3” uri=”http://xpto.org/admin/aop/advices/3”>
       <before>scripts.aop.performanceTestStart<before>
       <after>scripts.aop.performanceTestFinish<after>
</advice>
```

A listagem das regras do *advice* pelo URI específico *http://xpto.org/admin/aop/aspects/3/rules* confirma a existência de uma regra:

```
<paged-list page="1" item-count="1" items-per-page="20" >
   <rule id=”10” uri=”http://xpto.org/admin/aop/rules/10”/>
</paged-list>
```

Um pedido GET efectuado a *http://xpto.org/admin/aop/rules/10* devolve o seguinte conteúdo:

```
<rule id="10" uri="http://xpto.org/admin/aop/rules/10">
  <http-method-names item-count="3"/>
  <mime-types item-count="1"/>
  <applications item-count="0"/>
       …
  <application-uris item-count="0"/>
</rule>
```

O formato será reduzido, indicando em cada lista apenas o número de itens disponíveis. Confirma-se a existência de três itens para nomes de métodos HTTP e um item para tipos MIME. Um pedido adicional a *http://xpto.org/admin/aop/rules/10/http-method-names* confirma a configuração enviada:

```
<paged-list page="1" item-count="3" items-per-page="20" >
   <string>GET</string>
   <string>POST</string>
   <string>PUT</string>
</paged-list>
```

Uma das possibilidades destes sistema é a activação e desactivação de aspectos em *runtime*. Se não não é necessário aplicar mais este aspecto, então basta enviar um pedido HTTP de PUT ao URI de *http://xpto.org/admin/aop/aspects/3/active* com conteúdo em texto de *"false"*. A resposta confirma a desactivação do aspecto, como visto de seguida:

```
<aspect id="3" name="perf test" active="false"
 uri="http://xpto.org/admin/aop/aspects/3">
…
</aspect>
```

O servidor ao tratar este pedido não só altera o atributo *active* para o valor de *false*, como também *desactiva* o aspecto, actualizando todas as acções por forma a não incluírem o *advice* indicado. Esta capacidade flexibiliza e simplifica a aplicação e remoção de aspectos às aplicações.

O recurso ao XML e o esforço de edição manual associado deve ser mitigado pela disponibilização de ferramentas gráficas. Está em desenvolvimento um editor para criação e aplicação de regras através do sistema de administração. Todo este processo fica assim reduzido à interacção com uma interface gráfica que aplica imediatamente as alterações.

O servidor deve então suportar estes serviços de administração de AOP, garantindo que qualquer alteração efectuada por estes serviços actualize o sistema *on-the-fly*, aplicando os aspectos às acções conforme as regras.

A próxima secção apresenta a solução desenvolvida para a questão de persistência em aplicações onde ser requer um suporte a alterações *on-the-fly*.

5.4 Persistência

Para a camada de persistência é requerida uma API voltada para uma abordagem OO, pois é a tendência geral e uma forma mais natural de lidar com o acesso a dados em linguagens OO. Por exemplo, a persistência orientada a objectos permite uma navegação simples através de referências, retirando a necessidade de *queries* com vários JOINS.

A solução lógica a adoptar no caso de uma plataforma Java seria o JPA, pela sua aceitação geral, e pelas várias implementações disponíveis. No entanto, o JPA não foi desenvolvido tendo em mente alterações *on-the-fly* das classes do domínio do problema. Mesmo a framework Grails, que utiliza o Hibernate, embora adicione algumas capacidades dinâmicas, ainda força a reiniciar a aplicação na grande maioria dos casos.

A abordagem aqui apresentada foi desenvolvida devido às necessidades de dinamismo da

framework e o facto de nenhuma API actual já existente reflectir essas considerações. Outra consideração a referir é que a framework deve suportar a geração automática de aplicações com suporte a CRUD e alterações dinâmicas, logo deve fornecer código para lidar com essas acções.

Esse código que processa os pedidos de guardar, editar ou remover dados não pode depender de uma implementação específica de persistência, pois isso introduz várias dependências o que leva a um forte acoplamento de componentes. A alteração da solução nesse caso leva a grandes alterações do código e força a um grande esforço. Por forma a solucionar esse problema, e sobretudo permitir a redefinição das classes foi desenvolvida uma API simplificada de persistência orientada a objectos que abstrai a implementação base. (Figura 22)

Para gestão da persistência foi definida uma classe *Facade*[223],, a *PersistenceManagerProvider.* É nesta classe que devem ser registadas as soluções, associando uma implementação da interface *PersistenceProvider* a um nome para fácil identificação. O *PersistenceManagerProvider* mantém a lista de *PersistenceProviders* existentes e retorna os *PersistenceManagers* respectivos quando requisitados.

Cada solução de persistência deve implementar a interface *PersistenceProvider* e os métodos associados. Como soluções possíveis, pode ser utilizado JPA, simples ficheiros de texto ou XML, bases de dados orientadas a objectos como o DB4O[39] ou NeodatisODB [123], entre muitas outras. É ainda da responsabilidade do *PersistenceProvider* disponibilizar objectos do tipo *PersistenceManager* quando requisitados, bem como fornecer um conjunto de operações básicas de controle da solução, tais como parar e iniciar a base de dados ou redefinir as classes.

A classe essencial às operações comuns de salvar, editar ou remover dados é o *PersistenceManager.* Esta interface abstrai os pormenores de implementação destas operações, que depende da solução específica de persistência.

A API é básica, fornecendo um subconjunto das capacidades do JPA. Os exemplos seguintes demonstram o uso das operações mais comuns, tais como guardar um objecto :

```
PersistenceManagerProvider pmp = new PersistenceManagerProvider(“hibernate”);
PersistenceManager persistenceManager = pmp.getPersistenceManager();

Client client = new Client(“Guilherme gomes”);
persistenceManager.store(client);

persistenceManager.close();
```

Utiliza-se o nome *“hibernate”* de forma a aceder ao *PersistenceProvider* que implementa a solução de persistência. Esta implementação então retorna um objecto do tipo *PersistenceManager*, já com um transacção aberta. Estas acções são feitas através do *PersistenceManagerProvider*, sem que o programador tenha acesso a esses pormenores.

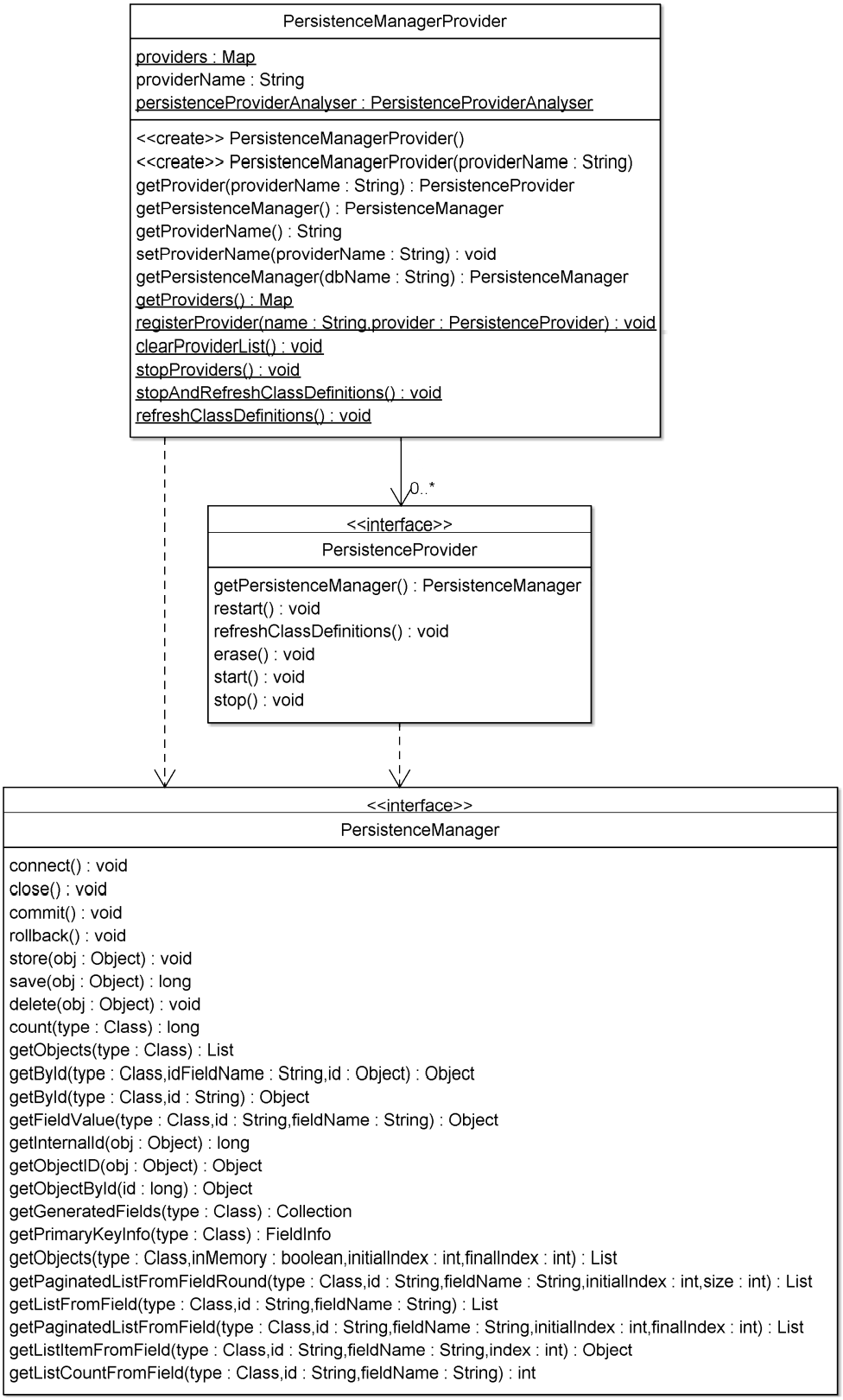

Figura 22: Diagrama de classes para a API simplificada de persistência da framework Tábula

O objecto é persistido pelo método *store* do *PersistenceManager*. A solução de persistência, neste caso o Hibernate, então trata de salvaguardar os dados. No caso de haver necessidade de cancelamento as acções realizadas através do *PersistenceManager*, basta efectuar o método *rollback*, como no seguinte excerto:

```
PersistenceManagerProvider pmp = new PersistenceManagerProvider("hibernate");
PersistenceManager persistenceManager = pmp.getPersistenceManager();
try{
   Client client = new Client("Guilherme gomes");
   persistenceManager.store(client);
   //código adicional lança excepção
}catch(Throwable t){
   //tratar excepção
   persistenceManager.rollback();
}
persistenceManager.close();
```

O método *rollback* tratar de cancelar a transacção actual, cancelando a aplicação das alterações à base de dados. Para além destes exemplos, existem vários outros métodos que disponibilizam as funções mais básicas e comuns de forma simplificada.

É de referir que a solução de persistência deve suportar a navegação entre objectos. Ou seja, o aspecto de obter informação da base de dados à medida que a árvore de objectos é navegada tem de ser implementada pela solução, quer pelo carregamento completo dos objectos relacionados, ou pela solução optimizada de *lazy loading*. O exemplo seguinte demonstra um desses casos :

```
PersistenceManagerProvider pmp = new PersistenceManagerProvider("hibernate");
PersistenceManager persistenceManager = pmp.getPersistenceManager();

Client client = persistenceManager.getById(Client.class,33);

List<Order> orders = client.getOrders();
for(Order order: orders)
{
   processOrder(order);
}

persistenceManager.close();
```

O objecto cliente retornado deve navegar para outros objectos referenciados, como no caso da obtenção da lista de ordens, sem necessitar nenhuma chamada adicional à API de persistência. Este aspecto visa a simplificação do código e aumento de produtividade do programador.

Queries

Esta API exibe uma limitação óbvia relacionada com a execução de *queries*. Esta limitação foi uma decisão deliberada pois infelizmente não existe um consenso em torno de uma linguagem única, multi plataforma para efectuar *queries* em soluções de persistência orientada a objectos. Em termos de soluções de ORM, o JPA fornece uma linguagem específica, a *Hibernate Query Language* (HQL)[80], enquanto que o JPA fornece outra solução denominada *Java Persistence Query Language* ou JPQL. Nas soluções de bases de dados orientadas a objectos, o DB4O fornece *Query By Example*, SODA e *Native Queries*, enquanto que o NeodatisODB API específica o uso de

CriteriaQueries e outra abordagem à *NativeQuery*.

Como esta área relacionada de ORM e persistência orientada a objectos é relativamente nova, ainda não foram feitos grandes esforços de uniformização. Não existe uma alternativa comum, um análogo na persistência orientada a objectos ao SQL.A solução mais interessante neste momento é a *Language Integrated Query* (LINQ) [111], disponível na plataforma .NET 3.5 como parte da suite ADO.NET[3] para acesso de dados. O LINQ existe sobre a Entity Framework[52], que é a solução de ORM da plataforma .NET 3. Esta abordagem torna as *queries* parte da linguagem de programação, pelo que é facilmente alterável e suporta *refactoring*.

Nas soluções acima citadas, isso não acontece, pois utilizam strings frequentemente para identificar as classes, nomes do atributos ou outros parâmetros. Num caso de *refactoring*, estas strings têm de ser alteradas manualmente, ou então tem de ser desenvolvido código por forma a construir as strings utilizando a informação das classes em *runtime*, o que não é nem simples nem intuitivo.

Não existe de momento uma solução semelhante ao LINQ para a plataforma Java, nem alguma iniciativa de padronizar esta abordagem em múltiplas plataformas. A especificação ou desenvolvimento de um projecto dessa natureza constitui, sem dúvida, um desafio interessante que merece atenção e trabalho adicional, mas encontra-se fora do âmbito deste projecto. O trabalho aqui apresentado centra-se na disponibilização e processamento dos serviços REST, suportando o desenvolvimento de forma dinâmica.

Assim, os objectivos relevantes à API apresentada nesta secção são abstrair as implementações de persistência, e disponibilizar os métodos para as acções mais comuns de forma simples, não só para os programadores iniciantes como para o suporte à geração automática de aplicações CRUD. É preciso ter em conta que esta abordagem não impede, de forma alguma, a utilização directa das soluções já existentes com as suas capacidades adicionais de *queries* para os programadores avançados.

Resumo

A API aqui apresentada responde às necessidades indicadas, permitindo que existam vários *PersistenceProviders*, cada um podendo ser uma implementação distinta, mas no entanto fornece uma interface uniforme e abstracta para efectuar operações comuns através do *PersistenceManager*. É relativamente simples adaptar soluções como JPA, DB4O ou NeodatisODB a esta API, através da implementação de algumas interfaces.

Como os objectivos da framework são relativos aos serviços REST, e a implementação serve de prova do conceito das especificações apresentadas, esta API na sua versão actual é suficiente para satisfazer essas necessidades. A questão das *queries* pode então ser resolvida posteriormente, de preferência com uma solução em Java semelhante e se possível, compatível com o LINQ.

5.5 Conclusão

Este capítulo definiu várias especificações relacionadas com as diferentes camadas do servidor a implementar. Este trabalho é fundamental devido à inexistência de soluções com as características de orientação aos serviços REST e requisitos de dinamismo desta framework.

As especificações servem de base para a criação da arquitectura e respectiva implementação, pois delineiam a funcionalidade geral de cada camada, orientando o desenvolvimento do trabalho. Descrevem ainda as estratégias para responder às necessidades e problemas apresentados sobretudo a nível de dinamismo, através de soluções de configuração, administração e ferramentas gráficas.

Foram apresentadas especificações para configuração e administração da camada de serviços REST de uma aplicação, bem como um editor gráfico por forma a facilitar todo este processo. O trabalho desenvolvido teve por objectivo responder às necessidades nesta área identificadas na exposição da proposta. A solução apresentada é dinâmica, multi plataforma e abstrai o tipo de implementação, fornecendo um conjunto de características que corresponde aos objectivos delineados.

Foi também apresentada uma solução para implementação de *handlers* alternativa ao uso directo de classes Java ou *scripts* em Groovy. A abordagem introduzida define acções como máquinas de estado finitas que processam os pedidos. A estrutura destas acções possibilita a reutilização de segmentos de código comuns ao processamento de vários pedidos e também abstrai a implementação desses mesmos segmentos. Permite assim manter a configuração da acção, mesmo que a implementação dos seus passos seja alterada.

A utilização de acções como *handlers* conjuntamente com a solução de AOP apresentada disponibiliza ferramentas para a estruturação e reutilização de código de forma flexível e dinâmica,

Posteriormente foi definida uma solução de AOP especificamente adaptada às acções, que torna possível aplicar *advices* através de um conjunto de regras. As regras permitem configurar de forma flexível as condições de aplicação dos *advices*.

Ainda foi introduzido um serviço de administração que expõe a configuração dos *aspects* como recursos, possibilitando a sua edição remota. Adicionalmente, qualquer alteração efectuada leva à actualização da aplicação dos aspects nas acções. Permite assim a criação, configuração e aplicação de *aspects* nas aplicações com suporte a alterações *on-the-fly*, o que é uma abordagem inovadora nesta área.A questão da persistência foi abordada com a introdução de uma API de acesso a dados orientada a objectos, que oferece funcionalidades básicas essenciais à maioria das aplicações. Esta API suporta as alterações em *runtime* via redefinição de classes. Com a introdução destas especificações neste capítulo, surge então a questão da viabilidade das soluções apresentadas. A resposta a essa questão é apresentada no próximo capítulo onde é descrito o processo de implementação da framework, com suporte a todas a especificações.

6 Implementação

6.1 Introdução

O objectivo da framework é desenvolver uma implementação das especificações e requisitos delineados em capítulos anteriores. Este capítulo descreve o processo de desenvolvimento do servidor apresentando os componentes relevantes e a sua funcionalidade. Como o projecto implementado possui uma grande dimensão, foi decidido apresentar a arquitectura de forma geral. A framework foi implementada sobre a plataforma Java devido à sua portabilidade, existência de um grande número de bibliotecas open-source e ambientes de desenvolvimento maduros.

O capítulo inicia com uma breve explicação da estrutura do servidor, seguido de uma exposição das funcionalidades oferecidas pela framework.

6.2 Estrutura geral do Projecto

A análise ao problema apresentado de criar uma framework com um servidor que apresente as características referidas na proposta levou à divisão da implementação em quatro projectos:

- **Core Rest Server** – corresponde ao núcleo do servidor, sendo a base sobre a qual os outros projectos são construídos. Os objectivos são a implementação das especificações da configuração da camada de serviços. As funcionalidades implementadas são :
  - Configuração de serviços REST;
  - Suporte à configuração dinâmica desses mesmo serviços;
  - Mecanismo para actualização da configuração dos serviços: *RESTConfigRefresh*;
  - Sequência de processamento de pedidos;
  - Suporte à associação de elementos da configuração com *handlers*;
  - Mecanismo de processamento de *handlers*, e;
  - Abstracção do acesso aos dados de configuração : interface *ApplicationProvider*.
- **Generation** – este projecto é responsável pelo aspectos de geração de aplicações automaticamente a partir da definição de classes do domínio do problema. Disponibiliza as seguinte funcionalidades:
  - Geração automática de camada de serviços e solução de persistência de uma aplicação baseada em DDD;
  - A aplicação gerada possui funcionalidades CRUD;
  - Permite o acesso remoto aos objectos, seus atributos e relações segundo a abordagem REST;
  - Suporta automaticamente o formato XML como tipo de representação dos recursos para todas as operações;

- As aplicações devem suportar uma solução de persistência por *default* sem ser necessário configuração do programador;
- Os pedidos devem ser processados por um conjunto de *handlers* genéricos;
- O processo de geração é configurável e extensível, e;
- Apresenta uma classe *Façade* para acesso às funcionalidades de geração: *ScafoldFacade*.

- **AOP** – providencia a implementação da especificação de AOP para as acções utilizadas como *handlers* no processamento de pedidos. Implementa:
  - Mecanismo de aplicação de aspects às *acções*;
  - Suporte à alteração dinâmica de aspects;
  - Abstracção do acesso aos dados de *aspects*: interface *AspectProvider*, e;
  - Mecanismo para actualização dos *aspects* no sistema através de uma classe *AOPFacade*.
- **REST Server** – este projecto é responsável pela integração dos projectos anteriores. Para esse fim, disponibiliza implementações para os componentes essenciais dos projectos e a configuração desses componentes. Obtém-se assim um servidor funcional com as características desejadas:
  - Implementação da solução de persistência pré-definida;
  - Implementação da solução de *marshall/unmarshal* de XML pré-definida;
  - Implementação do mecanismo de geração de aplicações pré-definido;
  - Implementa interfaces de componentes essenciais aos outros projectos, e;
  - Configura os componentes através de *Dependency Injection*.

Esta abordagem em quatro projectos distintos conduz à divisão do problema, permitindo restringir as responsabilidades de cada projecto a um conjunto bem definido, o que torna a implementação mais simples.

A próxima secção apresenta algumas das bibliotecas e projectos auxiliares utilizadas neste projecto.

6.3 Bibliotecas auxiliares

Foram escolhidas algumas bibliotecas para auxílio em aspectos específicos da implementação. Em casos onde não existiam alternativas, foram desenvolvidos projectos de menor dimensão por forma a disponibilizar funções úteis. Em primeiro lugar, são apresentadas as bibliotecas externas utilizadas e a seguir os projectos auxiliares desenvolvidos.

1.3.1. Bibliotecas externas

As bibliotecas externas são utilizadas para providenciar a funcionalidade específica de vários aspectos necessários ao servidor. A lista seguinte apresenta as bibliotecas:

- **Guice** – é uma solução para injecção de dependências. Dentro do projecto é utilizada por forma a configurar os diversos componentes do servidor, reduzindo as dependências e simplificando o processo de montagem do servidor. As suas funções incluem: injectar implementações de interfaces, gerir *singletons* e listas de objectos, entre outro aspectos;
- **Groovy** – consiste numa linguagem de scripting, sendo actualmente a solução de scripting mais popular na plataforma Java. É possível definir classes, compatível com a sintaxe própria do Java ou pode ser utilizada a sintaxe própria do Groovy;**XStream** – biblioteca para transformação de objectos para XML e vice-versa. Foi escolhida pela sua facilidade de utilização e configuração, embora não seja das soluções mais rápidas;
- **NeodatisODB** – é uma base de dados open source orientada a objectos. A versão actual ainda é beta, e faltam muitos aspectos básicos. Foi escolhida devido à sua facilidadede utilização, e é a única solução existente com uma licença compatível com este projecto, e;
- **RESTlet** – biblioteca que providencia a implementação dos elementos relacionados com a montagem de serviços REST, recepção de pedidos e identificação dos elementos dinâmicos dos URIs. Serve de base à funcionalidade do projecto *core rest server*.

Para além destas bibliotecas externas, foi necessário desenvolver vários outros pequenos projectos por forma a resolver problemas específicos aos quais não foram encontradas alternativas.

6.4 Projectos auxiliares

Alguns desses projectos são descritos a seguir:

- **ResourceUtils** – define várias classes que permitem aceder a recursos, classes e ficheiros através de um FQCN. As procuras são efectuadas em ficheiros jar, estruturas de directórios. e classes carregadas pela aplicação;
- **ReflectionUtils** – necessário para facilitar a obtenção de informação das classes e seus atributos em *runtime*. Agrupa vários acessos à API de *Reflection* do Java em funções mais simples;
- **Persistence** – consiste na implementação da API de persistência como definido no capítulo de especificação. Define uma classe *PersistenceManagerProvider* que gere as soluções de persistências. Cada solução deve implementar duas interfaces por forma a ser integrado nesta API;
- **GroovyUtils** – providencia um conjunto de ferramentas comuns para criação, configuração e cache de scripts *groovy*;
- **jtAction** – projecto de implementação de *acções* estruturadas como máquinas de estado finitas. É utilizado como uma das implementações de *handlers* com suporte a AOP;
- **NeodatisODB Utils** – a base de dados orientada a objectos NeodatisODB possui várias limitações, pelo que este projecto providencia soluções temporárias:
    - Gestão de ligações e servidores;

- Definição de atributos de classes como chaves primárias, com suporte a herança. Ou seja, as subclasses herdam a mesma chave. Este comportamento pode ser modificado se necessário, e;
- Geração automática de valores sequenciais para atributos de classes, com suporte a herança. As subclasses herdam a geração automática de valores de atributos. Este comportamento pode ser alterado.

Ainda em relação ao NeodatisODB, o código fonte do projecto foi acedido e alterado por forma a adicionar o suporte a :

- Persistência de classes definidas em Groovy, e;
- Redefinição de classes definidas em Groovy.

As alterações foram submetidas e aceites pelos criadores da NeodatisODB. Adicionalmente, devido à experiência prévia do autor com a DB4O, outra base de dados orientada a objectos, foi verificado que faltavam algumas capacidades que seriam úteis à Neodatis ODB e a este projecto. Neste sentido, os programadores da NeodatisODB foram muito prestáveis na criação destas novas funcionalidades e na resolução de bugs.

6.5 Arquitectura geral

A Figura 23 mostra a relação entre os diferentes elementos necessários à implementação do servidor. O projecto REST Server destaca-se pois é responsável pela implementação de várias interfaces definidas nos projectos de *core rest server*, *generation* e *aop*. Esta abordagem reduz as dependências destes projectos em relação a bibliotecas externas. A integração dos diversos componentes é então efectuada por *Dependency Injection* recorrendo ao *Guice*.

O projecto *REST Server* possui vários sub-projectos, que são introduzidos a seguir:

1. DefaultXMLSuite – implementação default para construir representações em XML. Utiliza a biblioteca *XStream* [217];
2. DefaultODBPersistence – solução default de persistência utilizando NeodatisODB;
3. DefaultHandlers – implementação de *handlers* genéricos em jtAction;
4. DefaulScafoldHelpers - define as classes utilizadas para o processo de geração;
5. DefaultApplicationProvider e DefaultAspectProvider – implementam os detalhes de acesso a dados para obtenção da configuração REST e aspectos, e;
6. DefaultBootupProcessor – implementa a sequência de acções a executar na iniciação e actualização do servidor.

A arquitectura pode ser consultada em maior detalhe no anexo 4.

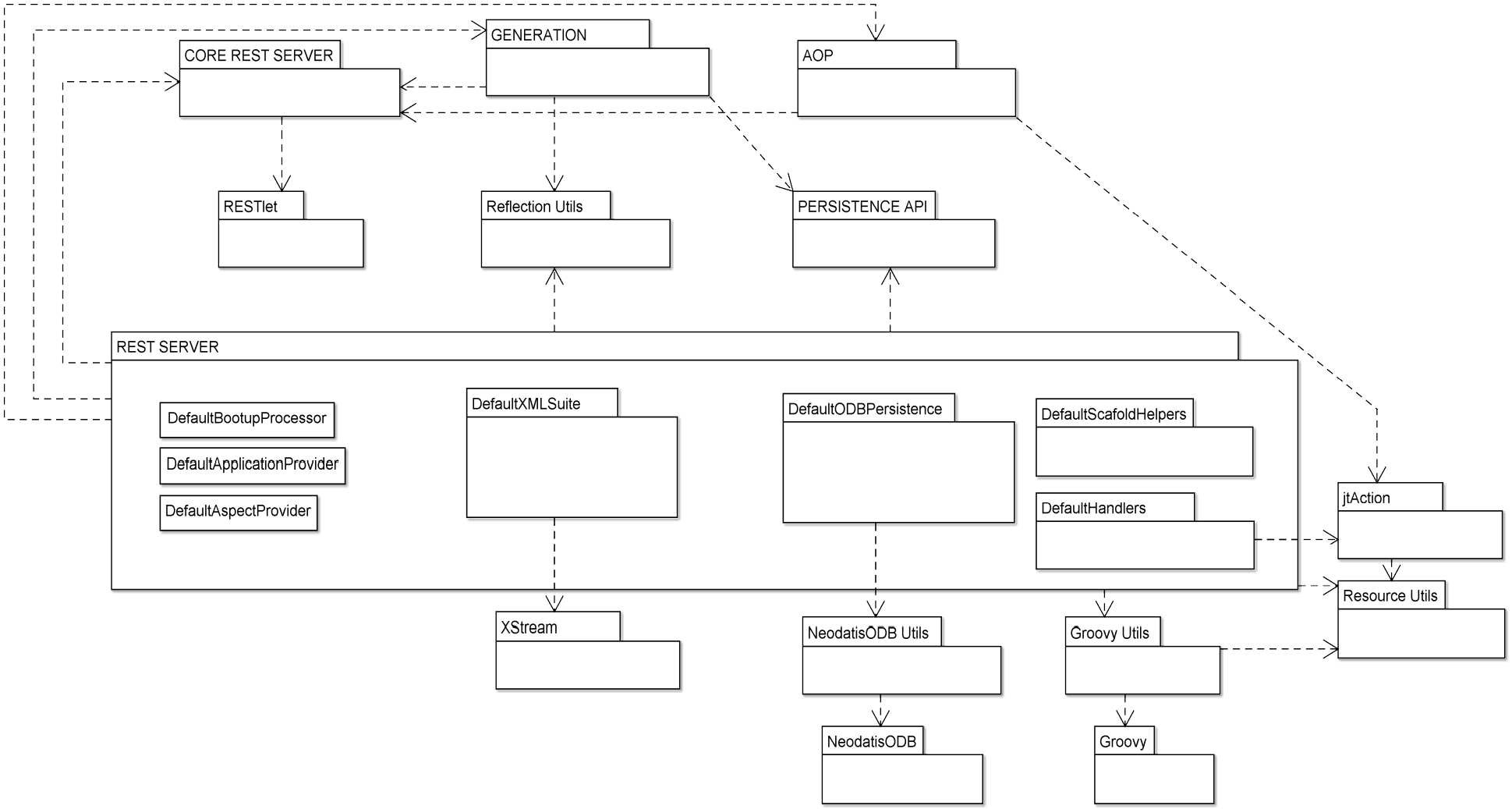

Figura 23: Componentes do servidor REST Server

6.6 Requisitos e funcionalidades

Após a apresentação dos componentes do sistema e a sua organização geral, apresentam-se agora as funcionalidades implementadas. Inicialmente, é feita uma breve introdução à geração automática de aplicações. De seguida, é explicada a abordagem utilizada para implementar o sistema de administração da configuração REST e AOP. Finalmente, apresentam-se as capacidades de configuração manual dos serviços REST.

6.6.1 Geração automática de aplicações

Esta secção apresenta a abordagem implementada na framework para geração de aplicações REST.

Definição das aplicações

Por forma a facilitar a definição das aplicações a gerar, recorre-se à estrutura de directórios. O directório *apps* do servidor é analisado sempre que o servidor é actualizado ou iniciado. Os directórios aí encontrados indicam as aplicações existentes. Por exemplo, para definir uma aplicação *test*, basta criar um directório com esse nome dentro do directório *apps*.

De forma similar, para definir os módulos dessa aplicação, primeiro é criado um directório *modules* dentro de *test*. Esse directório *modules* vai conter todo os módulos dessa aplicação. Para criar um módulo *crm* então basta criar um directório com esse nome dentro de *modules*. Para adicionar um outro módulo chamado *billing*, basta criar um directório também com esse nome (Figura 24).

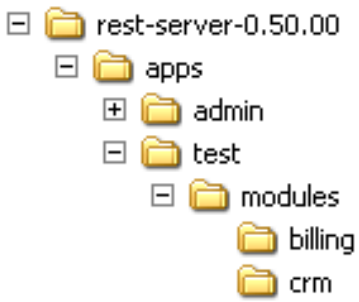

Figura 24: Estrutura de directório de uma aplicação

Os nomes dos directórios são usados como o atributo *relative–uris* da *Application* e *Module* da configuração REST. Portanto, todos as classes relativas ao módulo de crm, por exemplo uma classe *Client*, será referenciada pelo seguinte URI: */test/crm/clients*. De forma similar, a classe *Invoice* do módulo *billing* terá o seguinte URI: */test /billing/invoices*.

Definição das classes

Para indicar que classes utilizar em cada módulo como modelo do domínio do problema, em primeiro lugar, cria-se em cada módulo um directório chamado *entities*. A definição das classes

pode ser efectuada de duas formas:

1. Criando classes groovy directamente no directório *entities*, e;
2. Referenciado classes externas ao directório.

No caso 1, por exemplo, basta criar em */test/modules/crm/entities* um ficheiro de nome *Client.groovy* com o seguinte conteúdo :

```
package apps.test.modules.crm.entities;

public class Client{
   String name;
   Date birthday;
   float height;
   boolean male;
}
```

No caso 2, para referenciar classes externas, como por exemplo classes Java definidas numa biblioteca *jar* ou classes groovy definidas noutro directório, é necessária outra abordagem, mas igualmente simples. Basta definir no directório *entities*, um ficheiro *classes.txt*.

Nesse ficheiro indica-se que classes utilizar no módulo indicando o seu FQCN. Por exemplo, se existir uma classe *Client* em Java no pacote *org.softmed.utils*, basta escrever nesse ficheiro a seguinte linha: *org.softmed.utils.Client*. Se quisermos importar outra classe, por exemplo, *Address* do mesmo pactote, basta adicionar na linha abaixo o seguinte texto: *org.softmed.utils.Address*.

```
org.softmed.utils.Client
org.softmed.utils.Address
```

Se é necessário importar um grande número de classes, em vez de as indicar individualmente, pode-se importar todas as classes definidas num pacote. O caso anterior pode ser substituído por apenas uma linha: `org.softmed.util.*`

É um mecanismo simples, e mais exemplos estão disponíveis no tutorial vídeo 2.14 em [183].

Actualização e geração

Durante a geração da aplicação, o servidor identifica as classes e associa-as ao módulo. Por forma a iniciar o processo de geração é necessário recorrer à consola de controle, como visto na Figura 25:

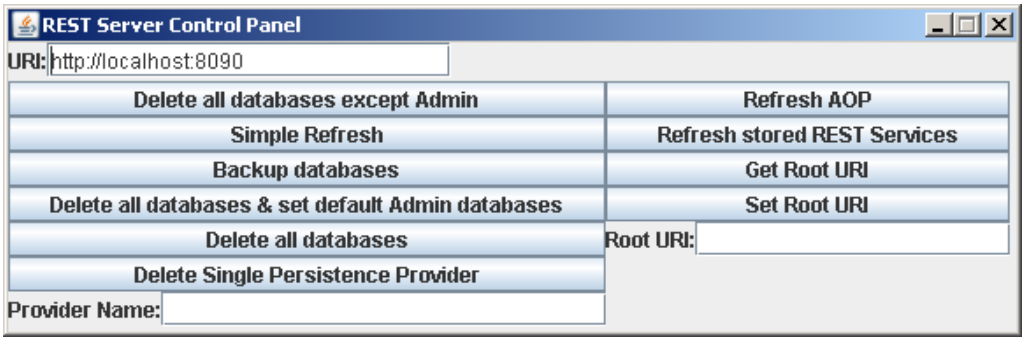

Figura 25: Interface de controle do servidor

Basta executar o comando *Simple Refresh* para forçar a actualização do servidor. Durante a actualização, são detectadas as aplicações e gerada as funcionalidades de acesso remoto a objectos e persistência. A partir desse momento é possível então criar, aceder, modificar e apagar objectos das classes definidas nos módulos via serviços REST.

No caso da classe cliente, são criados os recursos com os uris relativos, método http e tipos mime representados na Tabela 3:

Tabela 3: Configuração REST criada para a classe Client

Target-uri	Métodos HTTP				Tipos MIME	
	GET	POST	PUT	DELETE	XML	TEXT
/test/crm/clients	x	x			x	
/test/crm/clients/{id}	x	x	x	x	x	
/test/crm/clients/{id}/name	x	x	x	x		x
/test/crm/clients/{id}/birthday	x	x	x	x		x
/test/crm/clients/{id}/height	x		x			x
/test/crm/clients/{id}/male	x		x			x

O recurso */test/crm/clients* representa a lista de clientes em geral. Suporta pedidos GET para obter a lista ou pedidos POST para criar um novo cliente. Os outros recursos são de acesso a clientes individuais e seus atributos. Conforme os atributos são tipos primitivos ou objectos, os métodos suportados variam. O campo *birthday*, como é um objecto do tipo *Date*, pode ser apagado, enquanto que um campo *boolean* ou *float* é primitivo e por isso está sempre alocado, não podendo ser efectuadas operações de criação (POST) ou remoção (DELETE), apenas de leitura (GET) e alteração (PUT).

Os mapeamento de classes e seus atributos para recursos e métodos segue esta metodologia, adoptando os mesmo nomes para construção dos URIs.

Persistência

Pode ser verificado que após a actualização, foi criado um ficheiro chamado *persistence.groovy* no directório da aplicação. Isto é porque a aplicação não indicou nenhuma solução própria, pelo que o servidor forneceu uma solução usando as configuração por default, que recorre à base de dados NeodatisODB. O conteúdo desse ficheiro é grande, mas a linhas relevantes são as seguintes:

```
provider = new ODBPersistenceProvider();
name = "test-provider";
```

A primeira linha cria uma nova base de dados Neodatis que implementa as interfaces da PersistenceAPI, a API de persistência apresentada no capítulo de especificação. A segunda linha define uma variável que é o nome que vai estar associado àquela base de dados.

O servidor durante a actualização, executa este script, obtém estas duas variáveis e regista a base de

dados com o nome `test-provider`. Durante a geração da configuração REST, os *handlers* passam a ter um parâmetro a indicar aquele nome. Assim quando é processado um pedido para criar um cliente, o código executado no *handler* recebe o parâmetro e consegue então aceder a esta solução de persistência para guardar o cliente.

Em relação à identificação de objectos, a NeodatisODB gera automaticamente um número para cada objecto na base de dados. Por definição, esse é o numero utilizado na identificação do objecto e construção do seu URI. No entanto, a NeodatisODB tem algumas limitações, pelo que o projecto NeodatisUtils oferece mais capacidades, como a definição de campos como chaves primárias e a geração serial de valores em atributos.

Esta configuração pode ser efectuada no ficheiro *persistence.groovy*, com uma API simples como demonstrado no tutorial vídeo 2.21 em [183].

XML

O tipo de representação predefinida na framework é em XML. A framework cria automaticamente o mecanismo de mapeamento de objectos para XML e vice-versa, tendo como base as classes dos objectos. No exemplo da classe *Client*, a representação de um *Client* será, por exemplo:

```
<client id="1" uri="http://xpto.org/test/crm/clients/1">
   <name>guilherme gomes</name>
   <birthday>30-05-1979</birthday>
   <height>1.8</height>
   <male>true</male>
</client >
```

Note que as *tags* reflectem o nome da classe e o nome dos seus campos. Os atributos *id* e *uri* são adicionados automaticamente, pois são essenciais para identificar o objecto na aplicação. De notar que a *framework* suporta nomes definidos usando a forma *CamelCase*. Se a classe *Client* tiver um campo chamado *workAdress*, a *tag* correspondente será <*work-address*>.

Esta forma de representar os objectos apresenta um problema nas situações em que o programador utiliza classes externas ou classes *legacy* que não podem ser alteradas, mas no entanto precisa de um mapeamento diferente do criado automaticamente.

Para resolver esse problema, é possível criar um ficheiro *aliases.groovy* no directório do módulo. Nesse ficheiro configura-se o mapeamento como se vê no seguinte exemplo:

```
cfg.addAlias(Client.class,"costumer");
cfg.setAttributeField(Client.class, "name");
cfg.addOmitedFields(Client.class, "male");
cfg.addFieldAlias(Client.class,"name","full-name");
cfg.addFieldAlias(Client.class," birthday ","bday");
```

Com este código no aliases.groovy, basta efectuar uma actualização ao servidor, e o mapeamento é alterado. A representação do cliente anterior mudaria para :

```
<customer full-name="guilherme gomes" id="1"
uri="http://xpto.org/test/crm/costumer/1">
   <bday>30-05-1979</bday>
   <height>1.8</height>
</client >
```

De notar que o URI do objecto também muda. Essa é outra capacidade da *framework*. Os URIs são sempre gerados e não guardados nos objectos. Assim pode-se copiar ou mover uma base de dados de uma aplicação para outra, ou mudar de servidor, sem que isso force a modificações dos dados. A ideia é que a framework não pode forçar à camada de persistência aspectos relacionados com serviços.

É também possível mudar o URI do servidor ou mesmo do módulo sem ser necessário reiniciar o servidor, basta uma actualização.

É de referir que a configuração efectuada no ficheiro *aliases.groovy* não é específica à representação em XML. Estes dados ficam guardados para uso na geração da aplicação, independentemente do tipo de representação suportada. Se de futuro for adicionada uma representação em JSON, a geração pode na mesma aceder a estes dado para formatar o mapeamento dos objectos.

Note que nos script *aliases.groovy* e *persistence.groovy* não é necessário adicionar instruções de *import* das classes dos módulos. A *framework* quando executa estes scripts, já efectuou o levantamento das aplicações, dos seus módulos e classes associadas. Antes de executar estes *scripts*, a *framework* adiciona essas instruções de importação pelo programador, por forma a reduzir o esforço. Para mais detalhes consultar o tutorial vídeo 2.13 em [183].

Desenvolvimento incremental

O servidor permite adicionar novas classes à aplicação, requerendo apenas uma actualização para voltar a gerar o acesso aos objectos. Classes em módulos distintos podem possuir referências entre si. Por exemplo, adicione uma classe *Invoice* ao módulo *billing* na aplicação criada anteriomente, criando um ficheiro *Invoice.groovy* no directório /apps/test/modules/billing/entities com o seguinte conteúdo:

```
package apps.test.modules.billing.entities;

import apps.test.modules.crm.entities.Client;

public class Invoice{
  Date date;
  Double amount;
  Client customer;
}
```

Após uma actualização ao servidor, pode ser criados objectos do tipo *Invoice* enviando por exemplo um *POST* a */test/billing/invoices* com o seguinte conteúdo (notar que o mapeamento do *client* foi alterado para customer):

```
<invoice >
   <customer uri= "http://xpto.org/app1/crm/customer/23123"/>
   <date>25-12-2008</date>
   <amount>249.99</amount>
</invoice>
```

A framework suporta o uso de referências a objectos entre classes. Aliás o propósito da *framework* é exactamente suportar este tipo de desenvolvimento gradual da aplicação. Tal como podem ser adicionadas classes, também podem ser retiradas ou alteradas. O programador vai então testando os serviços, configurando os mapeamentos, e actualizando o servidor por forma a aplicar estas alterações, sempre sem ser necessário reiniciar a aplicação. Estas situações estão demonstradas nos tutoriais vídeo de 2.8 a 2.11 em [183].

A base de dados NeodatisODB ainda tem limitações a nível de *refactoring*, pelo que no caso de alterações de classes com objectos já persistidos, o processo de actualização deve ser outro. Para estes casos a solução adoptada é efectuar a cópia das bases de dados para um directório *backup*, e então apagá-las. Após a actualização, elas são recriadas. Este processo pode ser efectuado pelo *control panel*, como demonstrado nos tutorial vídeo 2.10 [183]. Não se perdem assim os dados originais e é possível continuar a desenvolver a aplicação.

Configuração Extra

A maior parte dos casos de desenvolvimento são abrangidos pela funcionalidades descrita anteriormente. No entanto, pode ser requerida maior controle sobre o processo de geração. Para esse fim, a *framework* suporta ficheiros adicionais que passam a ser apresentados:

- **xml.groovy** - permite ao programador definir a implementação utilizada para mapear os objectos para e de XML. A solução *default* utiliza uma série de classes com base no Xstream;
- **handler.groovy** – permite definir o conjunto de *handlers* que tratam os pedidos aos recursos das aplicações geradas. O conjunto pré-definido consiste numa série de *handlers* implementados em jtAction, e;
- **module.groovy** – permite definir o componente que gera os serviços. Torna-se assim possível mudar completamente todo o processo de geração, mapeamento de objectos e seus atributos para recursos, a configuração dos *handlers*, dos recursos gerados, entre outros.

A *framework* oferece um conjunto de componentes e configurações pré-definidas, adaptadas aos casos mais comuns de utilização. Disponibiliza ainda formas de alterar vários aspectos do processo de geração como os nomes a utilizar no mapeamento XML e construção de URI, sem a definição das chaves primárias ou geração de valores.

Para caso mais avançados, os programadores podem controlar completamente o processo de geração, definindo as implementações para conversão objecto-XML, os *handlers* a utilizar e até mesmo o próprio mecanismo de geração dos recursos.

Esta configuração é feita través de ficheiros *groovy*, que podem ser alterados *on-the-fly,* requerendo apenas a actualização do servidor. Adicionalmente, é possível definir este ficheiros a nível da aplicação, a nível do módulo ou em ambos. Neste último caso, as definições da aplicação são utilizadas para todos os módulos com excepção dos que possuem ficheiros de configuração próprios. Assim é possível numa mesma aplicação, modificar completamente o mecanismo de geração de módulo para módulo.

A *framework* oferece assim, simplicidade para os programadores novatos e flexibilidade para os programadores experientes que desejem maior controle sobre o processo de geração. Para mais pormenores sobre a arquitectura utilizada na geração de aplicações, consulte o Anexo 4.

6.6.2 Serviços de administração de REST e de AOP

Um dos requisitos desta framework é a implementação dos serviços de administração da configuração REST e AOP. Em essência, o objectivo é permitir o acesso remoto aos objectos de configuração de acordo com a abordagem REST e a aplicação de alterações on-the-fly.

Ou seja, é necessário mapear as classes a um conjunto de recursos associados a um acesso REST. O problema é que são muitos recursos a definir para cada classe, atributo, relação entre classe, listas e seus itens.

A definição manual destes recursos está fora de questão devido ao seu grande número. Qualquer alteração à especificação levaria a uma grande quantidade de alterações. A solução escolhida é utilizar as capacidades de geração de aplicações do projecto *Generation* para criar a gestão e acesso remoto à configuração REST e AOP.

Assim, é criado uma aplicação *admin* com dois módulos *rest* e *aop*, usando a estrutura de directórios.

No módulo *rest* adiciona-se o directório *entities* com um ficheiro *classe.txt.* Esse ficheiro indica que a aplicação importa as classes que representam a configuração REST, definida no projecto c*ore rest server* no pacote *org.softmed.rest.config.**. De forma semelhante, são importadas no módulo *aop* as classes que representam os *aspects*, *advices* e *rules*, mas nesse caso o pacote é o org.softmed.aop.*.

Cada módulo deve possuir a sua própria base de dados, pois assim, é possível copiar a configuração REST ou os *aspects* de um servidor para outro, de forma individual. É uma abordagem mais flexível do que colocar todos os objectos juntos. Para esse caso, definem-se ficheiros *persistence.groovy* em cada módulo, em vez de ser para a aplicação *admin*. Nesses ficheiros configura-se os campos a usar como identificação dos objectos e a geração serial desses campos. Adicionalmente, define-se o nome a associar a cada base de dados. No caso do módulo *rest* o nome é `"rest-admin-provider"` e no caso do *aop* é `"aop-provider"`.

Estes nomes são configurados respectivamente, nas classes *DefaultApplicationProvider* e *DefaultAspectProvider* do projecto *rest server.* São estas as classes utilizadas para obter a configuração REST manual e os *aspects* a cada base de dados, durante a iniciação e actualização do servidor.

Recorde que os serviços de administração não devem ser alterados durante as actualizações. Para esse fim, o projecto *Core Rest Server*, responsável pela instalação dos recursos, permite associar a um nome um conjuntos de recursos. Durante a actualização, o servidor retira todos os recursos existentes, lê a configuração manual através do *DefaultApplicationProvider* e finalmente instala os novos recursos. No entanto, a excepção são os recursos associados com o nome "admin". Esses recursos nunca são removidos durante as actualizações. Essa configuração é efectuada através de DI e é facilmente alterada.

Portanto, a configuração gerada para a aplicação *admin* e os seus dois módulos *rest* e *aop* deve ser associada a esse nome. A forma de efectuar essa configuração é criar um ficheiro *config.groovy,* colocado na raíz da aplicação. Este ficheiro define uma variável c*ontext, do tipo String* que indica o nome a associar aos recursos gerados nesta aplicação. Neste caso, o conteúdo desse ficheiro será:

```
context = "Admin";
```

Com estes dados, após a actualização, o servidor gera a aplicação permitindo o acesso remoto a objectos, a sua criação, edição e remoção. Resta apenas um problema: Até agora só se consegue o suporte à persistência. Quando se cria ou se altera um objecto do módulo *rest* ou *aop*, essa informação é guardada, mas o servidor não actualizou a configuração REST nem os *aspects* existentes. Em vez de forçar a uma actualização manual do servidor, pode-se recorrer aos próprios *aspects* para automaticamente aplicar as alterações.

Portanto, um aspecto foi definido manualmente, gravado na base de dados do módulo aop. Relembre que no processo de boot o *AOPFacade* carregou já todos os aspectos existentes através do *DefaultAspectProvider*, por isso este *aspect* está activo desde o início do servidor. O *aspect* assim criado possui como regra de aplicação qualquer pedido *POST*, *PUT* e *DELETE* efectuado à aplicação de *relative-uri admin* e módulo de *relative-uri rest.* Resumidamente, o aspecto é aplicado a qualquer acção que modifica qualquer objecto do módulo *rest*. O *advice* deste *aspect* é o seguinte:

```
<advice>
  <after>apps.admin.modules.refresh.actions.refresh-aop</after>
</advice>
```

Os *handlers* genéricos *default* do tipo *jtAction* que efectuam alterações vão assim adicionar os passos indicados nesta pós-acção. O valor de *after* identifica o ficheiro XML que define os passos a adicionar:

```
<action>
  <execution>
      <step source="apps.admin.modules.refresh.scripts.RefreshREST" />
  </execution>
</action>
```

Existe um passo a adicionar ao estado de execução. O atributo *source* identifica um *script groovy* em *apps/admin/modules/refresh/scripts/RefreshREST.groovy* com o seguinte conteúdo:

```
import org.softmed.rest.server.core.restlet.RESTConfigRefresher;
println "trying to refresh the REST Admin service";
RESTConfigRefresher.refreshConfig();
println "done";
```

Ou seja, sempre que seja recebido um pedido para criar, alterar ou apagar dados do módulo rest, após execução desse pedido, a configuração manual REST é actualizada por intermédio da *facade RESTConfigRefresher*. O *RESTConfigRefresher* utiliza o *DefaultApplicationProvider* para obter os dados de configuração da base de dados correcta, associada ao nome *"rest-provider"*.

O sistema de administração do AOP é criado da mesma forma. A base de dados do aop já possui um outro *aspect* definido previamente que se aplica às alterações dos objectos do módulo aop. Neste caso, o passo adicionado é diferente, invocando o método refresh da classe *AOPFacace*, que actualiza os aspectos existentes no sistema. Ou seja, usa as capacidades de AOP do servidor, para conseguir providenciar o suporte às alterações dinâmicas nos serviço de administração REST e do próprio AOP.

As únicas tarefas manuais requeridas para criar todo o sistema de administração foi a criação dos directórios, configuração de *aliases.groovy, persistence.groovy*, e *config.groovy* e finalmente a definição de dois aspectos para forçar a actualização em caso de alterações.

Esta é uma solução muito mais simples do que definir manualmente todos os recursos. Adicionalmente, esta abordagem tem a mais-valia de que qualquer alteração às especificações requer apenas o *refactoring* das classes que as representam no servidor.

Demonstra a flexibilidade da *framework*, em que a próprios sistemas de administrção são gerados pela *framework*.

A construção dos serviços de administração REST e AOP demonstra como resolver o problema da funcionalidade limitada das aplicações geradas.

Qualquer aplicação gerada automaticamente pelo servidor possui apenas funcionalidade CRUD e nada mais. À primeira vista parece uma solução muito limitativa. Por exemplo, numa aplicação de gestão de clientes, se é necessário iniciar um processo de *workflow* quando um cliente é criado, como resolver esta situação?

A solução é recorrer ao serviço de AOP. No exemplo acima referido, adiciona-se um aspecto que para as acções to tipo *POST* ao *uri /test/crm/costumers*, seja adicionado um passo que inicialize esse *workflow*. Esta configuração é feita dinamicamente, sem reiniciar a aplicação. O aspecto é aplicado e o próximo pedido de criação já vai executar o *workflow*.

Foi o mesmo princípio utilizado aqui na criação do serviço de administração. Os aspectos permitem integrar nas aplicações geradas automaticamente, qualquer funcionalidade adicional requerida, para além das operações CRUD por default.

Demonstra-se assim a flexibilidade e potencial dos mecanismos de geração dinâmica de aplicações e do serviço de AOP. Por forma a evitar a edição manual de XML, na definição de *jtActions* e *Aspects*, um editor gráfico está em desenvolvimento.

6.6.3 Configuração Manual

Foram apresentadas as capacidades de geração automática de aplicações a partir de classes. No entanto, o servidor suporta a definição manual da configuração REST através do serviço de administração. É possível criar manualmente aplicações, módulos, recursos e toda a informação necessária, em alternativa à solução gerada. Para facilitar esse processo, pode ser utilizado o REST Editor. Existem tutoriais vídeo disponíveis em [183] demonstrando o seu uso.

Resta a questão de como programar o *handlers* para as aplicações definidas manualmente.

Capacidades de programação :

Esta *framework* suporte de base a execução de *handlers* em Java, classes *Groovy*, *scripts Groovy* e *jtAction*. As soluções mais relevantes são os *scripts groovy* pela facilidade e *jtAction* pela estruturação, que se passa a explicar resumidamente:

- Groovy Script Handlers

No REST Editor, pode-se indicar criar toda a configuração de uma aplicação manualmente, e no casos dos *handlers* para essa aplicação, indica-nos no atributo *path* o FQCN que identifica o código a executar. Se for encontrado um *script groovy* que corresponde a esse FQCN, então esse *script* vai ser configurado e executado.

É de relembrar que o *script* tem de processar o pedido. Para esse fim, é necessário receber vários parâmetros. Esse parâmetros são colocados no *script* como variáveis já existentes. Com os seguintes nomes:

- *request* – objecto que representa o pedido efectuado pelo cliente;
- *response* – objecto que representa a resposta a retornar ao cliente;.
- *uriParameters* – é uma estrutura do tipo *map* com os segmentos dinâmicos do URI;
- *queryParameters* – é uma estrutura do tipo *map* com as secções de query no URI, ou seja, os segmentos adicionados ao URI após o separador '?';
- *application*, *module*, *resource*, *httpMethod*, *mimeTypeHandler* – são os objectos da configuração manual REST, e;

- *sequenceParameters* – estrutura *map* com as variáveis passadas de um *handler* para outro ao longo da sequência de processamento resource->httpMethod->mimetype.

É de referir também que as variáveis do *SequenceParameters*, e quaisquer parâmetros do *handler* definidos na configuração do REST estão disponíveis directamente no código. Adicionalmente, qualquer variável definida como *"global"* nos scripts *groovy* passa a ser parte do *sequenceParameter*, logo vai ser passada para os próximos *handlers* na sequência de processamento.

Uma configuração manual possível de um recurso é a seguinte:

```
<resource>
    <target-uri>/clients/{id}</target-uri>
    <handler active="true" path="scripts.getClient"/>
        <parameters>
            <parameter name="database" value="app1" />
        </parameters>
    </handler>
    <http-methods>
        <http-method name="GET">
            <mime-type-handlers>
                <mime-type-handler>
                   <mime-type>application/xml</mime-type>
                   <handler active="true" path="scripts.CreateXML" />
                </mime-type-handler>
                <mime-type-handler>
                   <mime-type>application/pdf</mime-type>
                   <handler active="true" path="scripts.CreatePDF" />
                </mime-type-handler>
            </mime-type-handlers>
        <http-method/>
    </http-methods>
</resource>
```

O FQCN scripts.`getClient` é mapeado para o ficheiro /scripts/`getClient`.groovy com o seguinte conteúdo:

```
def tempVar ="só existe no âmbito deste script";
println "Usando a base de dados " +  database;
pm = new PersistenceManagerProvider(database).getPersistenceManager();
chave = uriParameters.get("id");
entity = pm.getById(Client.class,chave);
```

Pode-se ver a utilização directa do parâmetro criado na configuração para aceder à base de dados correcta. Define-se ainda uma variável com âmbito limitado através da palavra chave *def*. A variável *entity*, no entanto, é global, ficando assim definida no *sequenceParameteres* e é passada aos próximos *handlers* na sequência.

No caso de um pedido *GET* para obter uma representação em XML, o *getObject.groovy* seria executado, seguido da execução de scripts.`CreateXML`, mapeado para o ficheiro /scripts/`CreateXML`.groovy com o seguinte conteúdo:

```
xml = xmlMarshaller.getXML(entity);
response.setEntity(xml,MediaType.APPLICATION_XML);
```

Pode-se ver a referência directa à variável *entity* que foi criado pelo script *getObject* anterior.

No caso do pedido ter sido para uma representação em PDF, então o ficheiro a executar seria */scripts/CreatePDF.groovy*:

```
import ByteArrayRepresentation;
pdf = pdfCreator.createPDF(entity);
response.setEntity(new ByteRepresentation(pdf.getBytes(),
MediaType.APPLICATION_PDF));
```

Ambos os scripts utilizam a variável *entity* definida em *getObject*. Esta abordagem permite programar a sequência de processamento como se fosse linear, deixando para a *framework* as decisões sobre que *handlers* executar. Basta implementar o código dos vários *handlers* como se tratasse de um único *script*.

Esta forma de implementação possui ainda o benefício de que cada *script* pode ser reutilizado. Por exemplo, os *scripts* `CreateXML` e `CreateXML` podem ser reutilizados por outros recursos, por exemplo, para representar facturas. Nenhum deles possui código específico à classe do objecto a representar, nem de onde obter esse objecto. O *handler* do recurso é que trata desses pormenores.

Mais exemplos podem ser consultados nos tutoriais vídeo 1.2,1.3, 2.4 e 2.5 em [183].

-jtAction Handlers

Se a partir do atributo *path* do *handler* for encontrado um ficheiro xml com a definição de um *jtAction*, então essa acção vai ser configurada e executada. Como visto, as acções podem conter vários passos e cada passo pode ser implementado de forma diferente. A *framework* suporta passos implementados como classes Java, classes *Groovy* e *scripts groovy*.

Os *handlers* genéricos usados para lidar com os serviços das aplicações geradas automaticamente utilizam *jtAction* com passos implementados em groovy. Esta implementação dos passos em *Groovy* tem a mais valia de conseguir inserir variáveis no espaço de memória do *script*, portanto o código deve ser escrito como se as variáveis já existissem e tivessem sido definidas anteriormente.

Tal como com os *handlers* em *groovy*, os passos do *jtAction* implementados em *groovy* também vão possuir o mesmo conjunto de variáveis relacionadas com o processamento do pedido. O elemento que se destaca na utilização de *jtActions* com passos *groovy* em relação ao uso directo de simples *scripts groovy* é que cada *handler* pode ser implementado em várias secções.

Por exemplo, o *handler* anterior /scripts/`getClient`.groovy se definido em *jtAction* seria:

```
<action>
   <execute>
      <step path=”scripts.GetDatabase” />
      <step path=”scripts.GetClientByID” />
   </execute>
</action>
```

O FQCN *scripts.GetDatabase* é mapeado para o ficheiro /scripts/GetDatabase.groovy com o seguinte conteúdo:

```
def tempVar =”só existe no âmbito deste script”;
println “Usando a base de dados ” +  database;
pm = new PersistenceManagerProvider(database).getPersistenceManager();
```

O FQCN *scripts. GetClientByID* é mapeado para o ficheiro /scripts/GetClientByID.groovy:

```
chave = uriParameters.get(“id”);
entity = pm.getById(Client.class,chave);
```

Com esta abordagem, o problema de obter um objecto é dividido em secções. O *script GetDatabase* pode ser reutilizado em vários recursos, pois a sua única função é obter um *PersistenceManager* a partir do nome da solução de persistência. Um *handler* para processar *GET* de Factura pode ser definido :

```
<action>
   <execute>
      <step path=”scripts.GetDatabase” />
      <step path=”scripts.GetInvoiceByID” />
   </execute>
</action>
```

onde o *GetInvoiceById* é :

```
chave = uriParameters.get(“id”);
entity = pm.getById(Invoice.class,chave);
```

O recurso a *jtActions* com passos implementados em *groovy* oferece assim as mesmas facilidades de programação do que *handlers* em *groovy*, mas com a capacidade adicional de segmentar o problema em pequenas secções reutilizáveis. Quer a definição da acção quer os passos em groovy podem ser alterados *on-the-fly* sem ser preciso reiniciar o servidor.

6.6.4 Dinamismo

Por forma a potenciar o dinamismo da *framework*, esta suporta ainda dois mecanismos adicionais que devem ser mencionados:

Carregamento automático de JARs - É comum no desenvolvimento de uma aplicação recorrer a bibliotecas externas. O processo de integração consiste em terminar a aplicação, adicionar os ficheiros JAR dessas bibliotecas, implementar o código que utiliza as APIs importadas, e finalmente reiniciar a aplicação.

Na *framework tábula*, é possível adicionar bibliotecas sem reiniciar o servidor. Existe um directório *jars* que é monitorizado de 3 em 3 segundos, em procura de novos ficheiros jars. Se algum for encontrado, essa biblioteca é carregada no *SystemClassLoader* do servidor. Todas as classes e recursos nesse ficheiro ficam assim disponíveis.

O código Java não pode ser alterado sem reiniciar o servidor, mas como os *handlers* e os *jtActions*

suportam implementação por *Groovy*, isso não é necessário. Os *scripts groovy* podem efectuar *imports* das novas classes e utilizar assim as APIs adicionadas imediatamente. Nem é preciso actualização manual, pois é tudo automático.

O processo de utilizar bibliotecas externas consiste assim em adicionar os JARS necessários no directório jar, e continuar a programar a aplicação em *groovy*, sem ser necessário reiniciar. Assim reduz-se a necessidade de terminar o servidor, adicionar *jars*, configurar manualmente o *classpathe reiniciar a* aplicação . Tudo pode ser efectuado *on-the-fly* como demonstrado no tutorial vídeo 2.7 em [183].

Processo de boot - Regra geral o processo de boot de um servidor é determinado pela implementação, recorrendo a ficheiros XML com a configuração inicial. Este mecanismo é limitativo, pois o comportamento durante a iniciação já é pré-determinado.

Na *framework Tábula*, o servidor não tem código de iniciação fixo. Existe um directório *bootup* que é verificado durante os processos de iniciação e actualização do servidor. Quaisquer *scripts groovy* existentes nesse directório são executados em ordem. Esta ordem pode ser determinada utilizando um número como prefixo do ficheiro e/ou pela ordem alfabética. Assim o processo de *boot* não é fixo, e pode ser alterado rápida e facilmente. Por *default*, já existem os seguintes *scripts*:

- **10_jar_loading.groovy** – carrega as bibliotecas no directório jars durante iniciação. A partir daí o directório é monitorizado automaticamente;
- **20_cache_reset.groovy** – efectua a limpeza das caches relacionadas com optimizações do servidor. Este aspecto é explicado no capítulo de testes;
- **30_scafold_refresh.groovy** – inicia o mecanismo de geração de aplicações;
- **40_rest_service_refresh.groovy** – efectua o carregamento das aplicações definidas manualmente utilizando a classe *RESTConfigRefresher*, e;
- **50_aop_refresh.groovy** - efectua o carregamento das carregamento de *aspects* utilizando a classe *AOPFacade*.

Após esta sequência, o servidor está pronto a receber os pedidos. É de notar que estes ficheiros são essenciais ao funcionamento do servidor tal como descrito neste capítulo, pelo que não devem ser retirados.

No entanto, os programadores podem criar novos scripts, configurando assim o processo de boot com funcionalidade adicional. A próxima actualização aplicará esses *scripts*. Este é um mecanismo flexível e com grande potencial para os programadores avançados alterarem a framework.

Note, no entanto, que este mecanismo de utilização de *scripts groovy* no *boot* não é fixo. Esta foi a solução desenvolvida por *default*, mas que pode ser completamente substituída por outra qualquer. Para mais detalhes consultar o Anexo 4.

6.7 Dificuldades

Esta secção apresenta as dificuldades mais relevantes no decorrer da implementação da framework.

Arquitectura

Em várias ocasiões durante a implementação foram identificados pontos na arquitectura susceptíveis de serem alterados, o que levou a alterações, algumas vezes extensas por forma a garantir a sua extensibilidade. Sem dúvida que código fixo é mais fácil de implementar, mas a solução flexível é muito mais poderosa, pelo que o esforço adicional foi compensado.

As arquitecturas desenvolvidas são extensas, mas respondem a estas necessidades, permitindo a configuração completa da *framework* a programadores avançados. É relativamente simples alterar os processos de geração de aplicações, adicionar suporte a novos métodos HTTP, novas formas de representação, modificar as implementações default de persistência e conversão XML, entre vários outros aspectos.

Conversão objecto-XML default

A maior dificuldade foi sem dúvida, a implementação do mecanismo *default* de conversão objecto-XML utilizando XStream. Foi necessário desenvolver algoritmos genéricos capazes de lidar com:

- Qualquer tipo de classes;
- Uma grande variedade de tipos de atributos;
- Alteração do mapeamento por configuração em *aliases.groovy*;
- Uso de URIs como identificação dos objectos;
- Utilização da *PersistenceAPI* por forma a abstrair as implementações de bases de dados;
- Referências no XML a outros objectos, mesmo de módulos distintos, e;
- Suportar todos os processos de criação e edição de objectos.

Destaca-se o problema da configuração no ficheiro *aliases.groovy* e como isso afecta completamente a construção de URIs e o mapeamento XML.

Outro obstáculo especialmente complicado foi o suporte à edição parcial de objectos. Por exemplo, se um objecto cliente já existe e queremos apenas alterar dois campos desse objecto, é permitido enviar uma representação XML apenas com esses dois valores indicados, em vez de reenviar a representação completa com todos os atributos.

Foram necessárias várias semanas de trabalho em desenho de arquitectura, implementação e testes até conseguir uma solução flexível e estável que respondesse a todas as necessidades. É impossível entrar nos detalhes dos algoritmos desenvolvidos no âmbito deste problema por requerem uma apresentação longa numa dissertação já de si extensa. No entanto como oferecem soluções a questões importantes, comuns a este tipo de aplicações, esses algoritmos serão alvo de trabalhos posteriores.

Redefinição de classes

Outra grande dificuldade foi garantir a redefinição de classes em *Groovy*. Quando o servidor gera uma aplicação, as definições das classes são associadas a vários aspectos, desde recursos ao mapeamento XML, à configuração da persistência. Quando se actualiza o servidor é necessário remover completamente todas as referências às classes *groovy*.

Só então podem os ficheiros ser recompilados e as classes definidas. Este processo complica-se porque as classes criadas têm o mesmo nome, embora na máquina virtual Java, sejam objectos distintos, ou seja são instâncias diferentes da classe *Class*.

Foi necessário alterar os mecanismos de carregamento de classes durante o processo de geração, bem como as formas de associar as classes às configuração da aplicações, e até alterar a base de dados NeodatisODB.

6.8 Conclusões

A implementação da *framework* responde a todos os requisitos do capítulo de proposta, apresentando uma solução flexível e potente. O desenvolvimento do servidor prova ainda a viabilidade das especificações apresentadas como solução aos problemas de definição e processamento de serviços REST.

A *framework* suporta a definição de aplicações REST através da geração automática de aplicações, ou então por configuração manual, recorrendo ao serviço de administração e à ferramenta REST Editor.

Em relação à geração automática, providencia um conjunto de soluções integradas para persistência, mapeamento objecto-XML e processamento de pedidos capazes de lidar com aplicações definidas de forma genérica. Oferece ainda mecanismos simples de configuração para os casos mais comuns, permitindo aos programadores avançados tomar controlo completo de todos os aspectos do processo de geração.

Esta framework disponibiliza uma ferramenta flexível e potente para a criação rápida de aplicações REST com interfaces gráficas por tecnologias RIA.

7 Testes e performance

7.1 Introdução

A implementação introduzida no capítulo anterior validou as especificações do capítulo 5 como uma solução aos problemas identificados no estado da arte e estudo de serviços, e como resposta aos requisitos da proposta apresentada.

O objectivo principal da implementação como prova de conceito das soluções propostas foi conseguido. No entanto, resta saber se é possível implementar as especificações e ainda assim garantir uma boa performance. Não basta apenas solucionar o problema, é necessário fazê-lo de forma eficiente. Uma solução flexível e dinâmica, mas que apresenta problemas graves de performance acaba por se tornar numa não-solução, uma vez que não pode ser adoptada.

Por forma a completar este trabalho, efectuou-se então uma série de testes de performance, de modo a caracterizar a implementação apresentada em relação à plataformas comuns disponíveis actualmente.

Este capítulo inicia com a descrição das optimizações aplicadas à framework, seguida dos testes efectuados. A seguir é apresentado um estudo detalhado dos resultados conseguidos nas várias plataformas e finalmente são retiradas várias conclusões relevantes.

7.2 Optimização

Durante a implementação da framework, foram feitos vários testes, por forma a testar a funcionalidade das aplicações enquanto se desenvolvia a framework. Desde o início notou-se alguma lentidão no processamento de pedidos. Este aspecto não era preocupante, uma vez que nos estádios iniciais do desenvolvimento de qualquer aplicação o foco é a implementação da funcionalidade básica.

Sem dúvida que desde o início é requerida uma implementação que tenha em conta aspectos básicos de arquitectura que potenciem uma boa performance. Mas o processo de optimização é relegado para uma altura posterior. Esse processo geralmente implica várias pequenas alterações, embora possa também levar a modificações na arquitectura básica da aplicação.

À medida que a framework foi sendo terminada, o problema da performance ganhou relevância. Os primeiros testes revelaram que o servidor era bastante lento, levando cerca de 100 segundos para completar uma sequência de 500 pedidos de GET a um objecto da base de dados, o que corresponde a 5 pedidos por segundo.

Estes resultados põem em causa a viabilidade das especificações e a arquitectura da implementação, uma vez que mostram uma performance inaceitável para qualquer servidor de aplicações. Surge

então a questão : a baixa performance deve-se a falhas inerentes à especificação e arquitectura da implementação ou apenas a falta de optimizações básicas ?

O problema reside em se os objectivos de suporte a serviços REST e alterações *on-the-fly* levou à criação de abordagens que pela sua natureza, forçam uma baixa performance. É de se esperar compromissos entre performance e flexibilidade, mas não de tal forma que a utilização do servidor se torne insuportável. Tornou-se então imperativo determinar as causas da lentidão de processamento e se possível, optimizar o servidor por forma a poder ser considerado uma alternativa viável.

7.2.1 Aperfeiçoamentos

Através de uma análise ao código e efectuando testes adicionais a pequenos excertos do código foi possível isolar e identificar alguns pontos susceptíveis de optimização.

Compilação e execução de *scripts* em Groovy

Em relação ao Groovy, é sabido que a sua performance situa-se abaixo do Java compilado. No entanto tem vindo a melhorar nas ultimas versões e os responsáveis pelo projecto têm delineado trabalhar mais nesse aspecto na próxima versão. Mesmo assim, a performance obtida no servidor estava demasiado abaixo do esperado. Após testes foi isolado a secção de código que causava o atraso.

Uma vez que é pretendido um sistema dinâmico, onde o programador altere os scripts e isso seja reflectido imediatamente no sistema, cada vez que um pedido é recebido, o script groovy associado era compilado e depois executado. A fase de compilação requer bastante processamento, pois é necessário fazer o *parsing* de todo o código no script. O resultado era uma drástica quebra de performance.

A solução adoptada foi fazer o caching dos scripts, deixando que o sistema verifique se os ficheiros foram alterados entretanto, permitindo a sua recompilação apenas em caso de modificação.

Acesso a ficheiros

O acesso a ficheiros para verificar se foram alterados introduz uma quebra de performance. Para evitar ao máximo que o servidor aceda ao sistema de ficheiros, é introduzia uma classe FileUtil. Esta classe é utilizada em vez das classes próprias do Java para operações I/O com ficheiros.

O FileUtil não só localiza ficheiros a partir de um FQCN, mas também efectua a cache temporária de certos ficheiros conforme a sua extensão. Por definição os ficheiros XML e Groovy são guardados temporariamente. Quando é feito um pedido para leitura do ficheiro, é devolvido o conteúdo da cache.

A cache de um ficheiro torna-se inválida ao fim de 3 segundos. Um pedido efectuado após esse período força o FileUtil a aceder ao ficheiro real e verificar o *timestamp* de alteração desse ficheiro. Se foi alterado, o conteúdo é lido e colocado em cache. Caso não tenha sido alterado, a cache é revalidada durante mais 3 segundos.

Na prática, se uma alteração foi feita a um script, ela só é propagada pelo sistema no pior dos casos após 3 segundos. As extensões que determinam os ficheiros a guardar e o tempo de espera entre acesso a ficheiros são uma variável que podem ser configuradas.

Esta abordagem permite evitar múltiplos acesso ao sistema de ficheiros da plataforma, o que é crucial sobretudo em momentos de maior carga.

XML

Em relação ao *marshalling* e *unmarshalling* de objectos para XML, existiam dois problemas: o primeiro é que a criação de objectos da classe XStream é bastante pesada, levando bem mais tempo que o processo de *marshalling*/*unmarshalling*. Em segundo lugar reside a configuração complexa dos objectos XStream, antes de serem utilizados. Essa configuração é necessária para identificar que campos de cada classe são expostos, com que nome, se são atributos ou não, entre várias outras operações. Adicionalmente este código é genérico, uma vez que pode ser aplicado a qualquer classe Java. Consequentemente, possui bastante código com vários testes, acessos a dados das classes, configuração das aplicações existentes, cuja execução adicional só provoca mais atrasos.

Similarmente ao problema com o Groovy, a solução adoptada foi uma vez mais, o caching dos objectos XStream que efectuam o processo de transformação entre objecto e XML. Para potenciar ainda mais esse caching, os objectos são guardados após a sua configuração, e mapeados à classe a que foram configurados. Assim, nos pedidos posteriores para obter uma representação de um objecto em XML, basta pedir à cache um objecto XStream já configurado à classe desse objecto, e executar o método necessário à sua função. Evita-se assim a criação e a configuração do XStream para cada pedido recebido.

Reflection

Finalmente, temos a utilização da API de *Reflection* do Java. Esta API permite obter dados sobre as classes em runtime, permitindo aceder aos campos e métodos declarados da classe, invocar métodos, entre várias outras capacidades. O problema surge devido a esta API ser muito usada no servidor, uma vez que a solução genérica requer muita informação sobre cada classe para funcionar correctamente.

Assim, cada pedido recebido pelo servidor despoleta várias sequências de código para obtenção dos dados das classes usando a *Reflection* API. Novamente usamos o cache destes resultados, usando a classe *ReflectionUtil*. Não só esta classe permite agrupar várias funções comuns que usam o *Reflection* durante o funcionamento do servidor, mas também guarda as informações recolhidas

para cada classe. Assim nos pedidos posteriores, em vez de efectuar toda uma sequência de chamadas à API de *Reflection*, ele simplesmente retorna os resultados já guardados.

As optimizações introduzidas podem ser resumidas à seguinte lista:

- Caching de scripts Groovy;
- Caching de objectos XML configurados;
- Caching de informação sobre classes, e;
- Redução de acesso a ficheiros;

A secção seguinte apresenta os resultados obtidos a partir destas alterações à framework.

7.2.2 Resultados

Foram feitos testes à medida que as optimizações foram sendo aplicadas. Estes testes consistiam numa sequência de 6 conjuntos de 100 pedidos de GET ao servidor. Os tempos foram registados e uma média é criada para o total dos 6 testes. A performance é então calculada em termos de pedidos por segundo como visualizado na Figura 26:

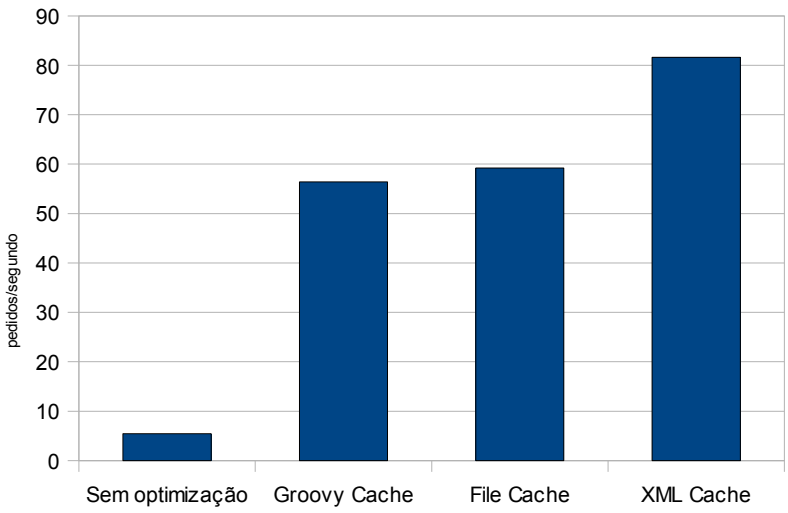

Figura 26: Aumento de performance por optimização introduzida

A criação de uma cache de scripts groovy resultou num aumento de 900%, 9 vezes mais rápido do que a versão original não optimizada.

A adição da cache de ficheiros, ou seja, o processo de permitir que se verifique a data de modificação do ficheiro apenas de 3 em 3 segundos levou a um melhoramento de 5%, pois leva a menos operações de I/O.

Finalmente, a cache dos objectos XStream já configurados aumentou em 38% a performance. Temos no total um aumento de 1500%, 15 vezes mais rápido do que a versão não optimizada.

Um teste alargado usando os quatro métodos HTTP (POST, GET, PUT e DELETE) pode ser visto na Tabela 4, detalhando melhor as diferenças de performance, em termos de pedidos por segundo:

Tabela 4: Comparação da framework com e sem optimizações

	Sem Optimização	Optimizado
POST	4,17	56,56
GET single	5,11	75,87
GET list	1,99	11,48
PUT	3,7	51,49
DELETE	4,64	105,71
TOTAL	**19,61**	**301,11**

Graficamente, as diferenças resultam no resultados expostos na Figura 27:

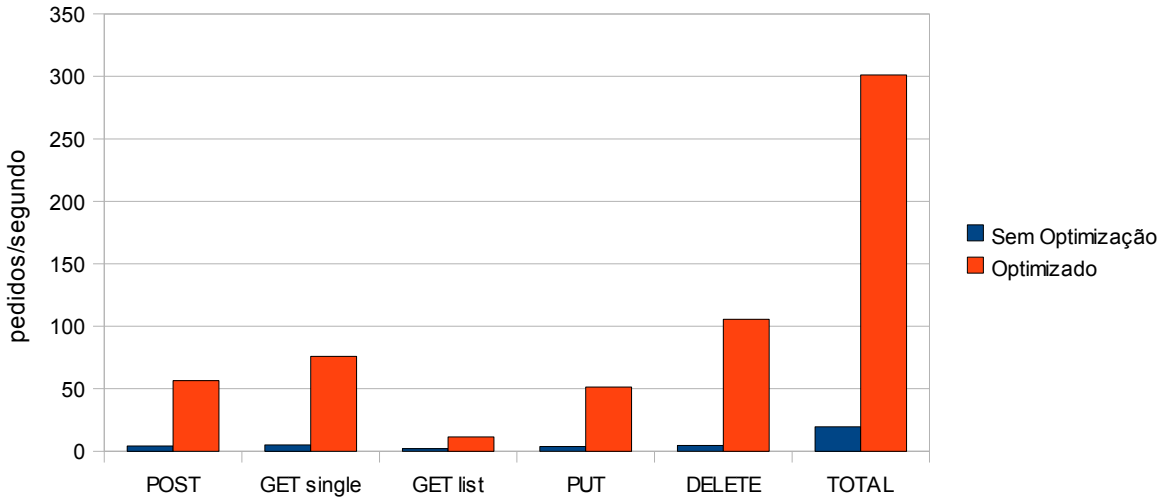

Figura 27: Comparação entre framework com e sem optimizações

Com este teste completo, vemos que no geral a versão optimizada é 1435%, ou seja, cerca de 14 vezes mais rápida, o que valida o processo de optimização.

Pode-se concluir que a baixa performance da versão inicial do servidor devia-se a vários aspectos da implementação relacionados com o processamento de pedidos. Com a introdução de optimizações, o servidor passou a responder muito mais rapidamente aos pedidos. Resta agora determinar se a framework é uma alternativa viável às soluções existentes, o que é apresentado na seguinte secção.

7.3 Benchmarks

Os benchmarks foram usados para dar uma ideia geral posição deste projecto em relação a soluções mais comuns, e em especial, como as escolhas para implementação de cada camada do servidor afectam a performance. Nesse sentido foram criadas diferentes plataformas, desde o servidor REST até ao Tomcat, que diferem gradualmente nalguns aspectos, de modo a avaliar a sua performance.

Para esta comparação não é necessário um teste extremamente detalhado, uma vez que o objectivo

desta implementação nunca foi a performance pura, mas sim validar a concepção da arquitectura.

Passa-se então a apresentar as condições e natureza dos testes, a descrição de cada plataforma seguida finalmente dos resultados e estudo dos mesmos.

7.3.1 Hardware

A máquina usada possui as seguintes características: Athlon X2 64 @ 2.51 Ghz, 4 Gb Ram ddr2 800 mhz (3GB detectados pelo Windows XP 32 bit), Disco rígido de 320 GB 8 MB de cache 7200 rpm, Windows XP Pro SP2 32-bits.

7.3.2 Testes

A aplicação usada para comparar a performance consiste num sistema muito simples de gestão de clientes, onde é possível efectuar as típicas acções CRUD – Create, Read, Update e Delete. Tendo em conta que estas acções são mapeadas para os quatro métodos HTTP - POST, GET, PUT e DELETE, respectivamente – os testes efectuam-se à execução desses mesmos métodos.

O procedimento de comparação efectua cinco testes para cada método HTTP suportado, calculando o tempo que leva a realizar cada teste. A média dos cinco testes é usada como resultado final para a performance desse método HTTP.

O conceito é testar as capacidades de cada plataforma em termos de tempo de resposta.

Para além dos 5 testes para o PUT, DELETE, GET e POST, foram efectuados mais cinco para um GET que retorna uma lista de 20 items, ao contrário do GET simples que retorna apenas um item.

Cada teste representa uma sequência de 500 pedidos, seguido de uma pausa de 20 segundos, e finalmente outra sequência de 500 pedidos. É considerado apenas o tempo que leva a executar a segunda sequência como resultado deste teste.

O propósito da primeira sequência é forçar o servidor a criar todos os recursos necessários para responder aos pedidos, ou seja, fazer um *warm-up*, esperar um pouco e finalmente executar a segunda sequência.

Na prática, e na maior parte dos casos, nota-se um melhoramento significativo na performance da segunda sequência, sendo entre 1 a 3 segundos mais rápido. Num caso específico, chegou a ser 10 segundos mais rápido. O objectivo desta estratégia é que o tempo que leva a executa a segunda sequência seja mais aproximado da performance de um servidor que estivesse a correr a aplicação há já algum tempo.

A sequência dos testes foi : POST, GET single, GET list, PUT e DELETE.

Os testes têm de ser efectuados nas mesmas condições, incluindo o estado da base de dados. É necessário considerar que a performance de utilização de uma base de dados varia com a dimensão desta. Por forma a garantir condições idênticas, adoptaram-se os seguintes procedimentos:

- Entre cada teste de POST, a base de dados é limpa por forma a não acumular todos os clientes criados, o que pode afectar os testes seguintes.
- Para os testes de GET a base de dados contém sempre 1000 clientes.
- Para o teste de PUT, a base de dados é recuperada às condições antes do teste, de modo a anular as alterações efectuadas ao 1000 clientes.
- Para o caso do DELETE o teste inicia com 1000 clientes. Como este clientes deixam de existir no fim do teste, a base de dados tem de ser recuperada.

No teste de "GET single", cada pedido pede um cliente diferente do anterior, em sequência, isto é, a primeira sequência pede os clientes de 1 a 499 e a segunda pede os clientes de 500 a 1000. Este método garante que se estivermos a usar uma base de dados orientada a objectos como a NeodatisODB, ou soluções ORM como o JPA com o Hibernate, não estamos apenas a retornar um objecto que já foi lido e esteja residente na cache. Assim desencadeamos efectivamente uma leitura da base de dados. O mesmo procedimento é usado para os testes de PUT e DELETE.

Devido à criação de um grande número de sockets durante os testes, e como no Windows XP há um limite ao número de sockets abertos simultaneamente, podem surgir erros de ligação entre os vários componentes do teste. Não só entre a aplicação cliente que efectua os pedidos e o servidor, como também entre o servidor e a base de dados, como no caso do MySQL[121]. Para solucionar esse problema, pode ser consultada a página da Microsoft que explica como mudar esse limite em [224]

7.3.3 Limitações

Existem algumas limitações em relação à base de dados NeodatisODB, utilizada como solução predefinida de persistência da framework Tábula que têm de ser considerados no contexto de análise de performance. Uma das limitações é não suportar a geração de chaves específica a classes. No entanto, possui a geração automática de um identificador único de objecto. Este valor é usado como "chave primária".

A limitação mais importante no contexto deste trabalho é a falta de suporte a *pooling* de ligações. Como tal, foi decidido usar também o MySQL sem pooling de ligações em vários casos, de forma a comparar as bases de dados em situações idênticas. Como o *pooling* de ligações é o padrão de facto, outras plataformas servem como base de comparação para esse caso, bem como o da utilização de tecnologias de ORM tais como o JPA via Hibernate com MySQL, também com pooling de ligações.

7.3.4 Plataformas

Foram criadas 11 plataformas distintas para as comparações. Cada uma consiste numa escolhas de diferentes componentes para cada camada do servidor, tais como Java ou Groovy como linguagem usada para implementar a camada de controle, NeodatisODB ou MySQL como solução de persistência entre outras possibilidades.

Cada plataforma é apresentada já de seguida e as suas características são resumidas num quadro comparativo.

A) **REST Server + Groovy + Neodatis ODB**: consiste no uso da solução genérica de geração de camada de serviços REST e persistência que já existe na plataforma. Como tal, usa código genérico e não optimizado para a aplicação em causa.

B) **REST Server + Groovy + Neodatis ODB**: Semelhante à solução anterior, excepto que não usa a solução genérica, mas sim código escrito manualmente e específico à aplicação em causa. O objectivo é obter resultados sobre o custo da solução genérica da plataforma A.

C) **REST Server + Groovy + MySQL**: Semelhante à plataforma anterior, excepto no uso do MySQL como solução de persistência ao invés da Neodatis ODB. É de referir que neste caso não é feito o pooling de ligações, de modo a melhor se equiparar à plataforma Neodatis ODB. O objectivo é podermos comparar a performance do MySQL contra a da Neodatis ODB, nas mesmas condições.

D) **RESTlet + Groovy + MySQL**: Semelhante à plataforma anterior, excepto que o servidor não usa o REST Server, mas sim a biblioteca RESTlet directamente. O servidor implementado neste projecto usa a biblioteca RESTlet, mas sobre esta foi implementado código adicional necessário para tornar possível a configuração *on-the-fly* da camada de serviços. Comparando esta plataforma com a plataforma anterior C é possível avaliar o impacto desse código.

E) **RESTlet + Java + MySQL**: A única diferença em relação à plataforma D é o uso de Java em vez do Groovy para implementação dos *handlers*, para avaliar a performance do scripting face a performance do java compilado.

F) **RESTlet + Java + JPA + Hibernate + MySQL**: As tecnologias ORM são extremamente populares actualmente, pois tornam mais fácil a interacção com a bases de dados relacionais, ao mesmo tempo que abstraem da implementação específica do SGBD. No entanto, constituem uma camada adicional, com uma baixa em performance correspondente. Assim também podemos comparar como uma aproximação ORM se situa face uma abordagem completamente orientada a objectos ao problema da persistência de dados.

G) **RESTlet + Groovy + JPA + Hibernate + MySQL**: A utilização do Groovy como linguagem de scripting permite nova comparação com a mesma solução em Java dada pela plataforma F.

H) **Tomcat + Java + MySQL**: Semelhante à plataforma E, excepto que o servidor passa agora a ser o Tomcat[180], de longe a solução mais popular actualmente para servidores Java.

I) **Tomcat + Java + MySQL**: Neste caso usamos o pooling de ligações à base de dados. Não só permite avaliar a baixa de performance ao não usar o pooling, como também serve de referência última de comparação, pois é muito difícil ultrapassar a performance desta solução no ambiente de desenvolvimento Java para aplicações Web. Usamos código Java compilado, um servidor estável, optimizado e comprovado por anos de utilização intensiva na indústria, e finalmente uma base de dados conhecida e utilizada a nível global. Como quase todas as frameworks estudadas na secção de estado da arte consistem geralmente em camadas adicionadas a um servidor Java como o Tomcat, com a função de facilitar a vida ao programador, todas elas serão invariavelmente mais lentas que esta solução. Daí o uso da plataform I como referência última em termos de performance.

J) **Tomcat + Java + JPA + Hibernate + MySQL**: O uso do JPA com a implementação do Hibernate e o MySQL é a escolha mais popular em termos de soluções ORM em Java. Comparando com a plataforma I podemos obter o custo em performance destas tecnologias.

K) **Tomcat + Java + Neodatis ODB**: em vez do ORM, é utilizada uma aproximação completamente orientada a objectos na persistência. O objectivo é avaliar a performance da base de dados Neodatis ODB contra o MySQL sem pooling, com pooling, e finalmente contra a solução ORM mais popular, o JPA com recurso ao Hibernate e MySQL.

A Tabela 5 sumariza os pontos relevantes na caracterização das plataformas:

Tabela 5: Caracterização das plataformas

	Descrição					
	Servidor	**Linguagem**	**Código Genérico**	**Base de Dados**	**Pooling de ligações**	**Persistência**
A	REST Server	Groovy	Y	Neodatis ODB	N	OO
B	REST Server	Groovy	N	Neodatis ODB	N	OO
C	REST Server	Groovy	N	MySQL	N	SQL
D	RESTlet	Groovy	N	MySQL	N	SQL
E	RESTlet	Java	N	MySQL	N	SQL
F	RESTlet	Java	N	JPA/Hibernate/MySQL	Y	ORM
G	RESTlet	Groovy	N	JPA/Hibernate/MySQL	Y	ORM
H	Tomcat	Java	N	MySQL	N	SQL
I	Tomcat	Java	N	MySQL	Y	SQL
J	Tomcat	Java	N	JPA/Hibernate/MySQL	Y	ORM
K	Tomcat	Java	N	Neodatis ODB	N	OO

A implementação de cada plataforma, bem como o código fonte e todos os recursos adicionais estão disponíveis no site do projecto, bem como tutoriais vídeo em [21] mostrando como utilizar esses recursos para montar cada plataforma de teste.

7.4 Resultados

Após a realização dos testes para todas a plataformas obtemos os seguintes valores, apresentados na

Tabela 6, em termos do tempo levado para concretizar os 500 pedidos:

Tabela 6: Resultados dos testes - tempos totais de execução em segundos

	Plataformas										
	A	B	C	D	E	F	G	H	I	J	K
POST	8,84	7,28	9,21	8,78	8,16	5,32	5,68	6,33	3,15	4,7	3,61
GET single	6,59	4,92	7,58	6,8	6,87	7,32	7,75	4,62	1,76	7,48	2,1
GET list	43,55	24,13	26,82	20,11	19,82	19,74	26,38	4,85	1,97	7,77	2,56
PUT	9,71	7,39	8,99	8,2	9,05	4,95	5,22	5,76	2,67	5,15	3,82
DELETE	4,73	4,26	6,64	6,57	6,08	3,36	3,89	4,8	1,81	4,08	2,34

Uma forma mais prática de avaliação da capacidade de resposta de cada plataforma é calcular ema sua performance em termos de pedidos por segundo:

Tabela 7: Resultados dos testes - valores em pedidos/segundo

	Plataformas										
	A	B	C	D	E	F	G	H	I	J	K
POST	56,56	68,68	54,29	56,95	61,27	93,98	88,03	78,99	158,73	106,38	138,5
GET single	75,87	101,63	65,96	73,53	72,78	68,31	64,52	108,23	284,09	66,84	238,1
GET list	11,48	20,72	18,64	24,86	25,23	25,33	18,95	103,09	253,81	64,35	195,31
PUT	51,49	67,66	55,62	60,98	55,25	101,01	95,79	86,81	187,27	97,09	130,89
DELETE	105,71	117,37	75,3	76,1	82,24	148,81	128,53	104,17	276,24	122,55	213,68
TOTAL	**301,12**	**376,06**	**269,81**	**292,42**	**296,77**	**437,44**	**395,82**	**481,28**	**1160,14**	**457,21**	**916,48**

A última linha da Tabela 7 consiste no somatório de todos os testes, dando assim um valor final à capacidade de cada plataforma em responder aos vários tipos de pedidos.

Dada a descrição dos testes e a caracterização de cada plataforma, passa-se agora para os resultados obtidos e estudos dos mesmos na próxima secção.

7.5 Análise

O processo de análise está estruturado da seguinte forma: em primeiro lugar, são mostrados os resultados dos testes. Depois são feitas comparações da framework Tábula em relação a plataformas semelhantes em termos de metodologias de implementação de aplicações. Finalmente é feito um estudo de como a performance é alterada conforme as soluções escolhidas para cada camada. Este estudo só é possível devido ao número de plataformas e à variação das características entre elas.

7.5.1 Resultados

Uma primeira análise aos resultados dos testes revela desde logo alguns dados relevantes, apresentados nesta secção. Pode ser feita uma avaliação global da performance das frameworks recorrendo aos resultados totais, que são o somatório do total de todas as operações efectuadas. (Figura 28). A exposição dos testes individuais pode ser consultada no anexo 5, sendo aqui apresentadas as conclusões.

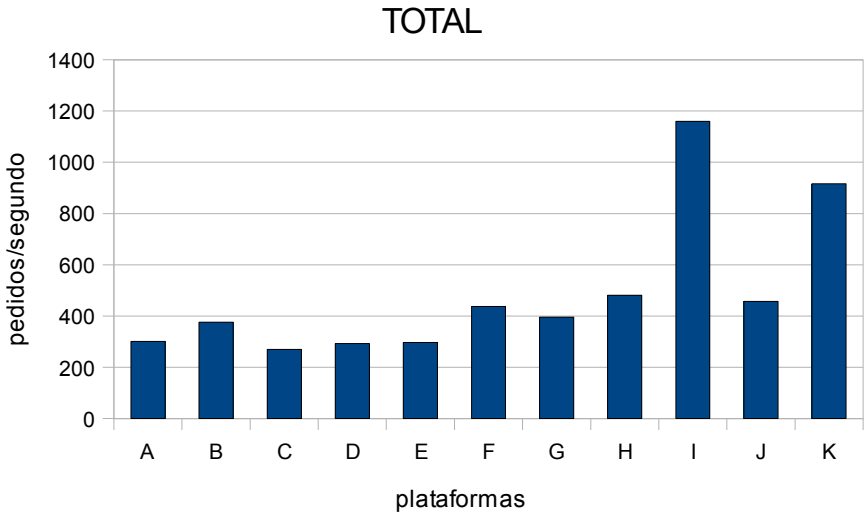

Figura 28: Comparação da performance geral das frameworks

A nível geral destacam-se claramente as plataformas I (Tomcat+MySQL+pooling) e K (Tomcat + NeodatisODB). Pode-se concluir finalmente :

- A predominância do Tomcat sobre o RESTlet, mostrando a sua maturidade e anos de optimização;
- A utilização de ORM tem um forte impacto negativo na performance, e;
- O pooling de ligações à base de dados tem um grande impacto positivo.

é de referir que no teste de *GET list*, as plataformas com base em RESTlet são muito lentas. A entrega de respostas com um corpo já com alguma dimensão parece estar na origem do problema. Já foi indicado que no geral o RESTlet é mais lento do que o Tomcat, mas nestas condições a diferença é drasticamente maior.

É possível que exista algum bug ou falta de optimização na realização de pedidos em que a resposta já é mais extensa do que algum número de bytes. Seriam necessários testes adicionais para verificar exactamente o que se passa, sobretudo se a performance descai gradualmente com o aumento do tamanho da mensagem, ou se há algum valor específico a partir do qual a performance cai abruptamente. Este é um aspecto que revela a juventude de um projecto como a biblioteca RESTlet.

7.5.2 Estudo Específico

A construção das várias plataformas tinha por objectivo não só a análise feita anteriormente como também o estudo mais detalhado do impacto em performance recorrente de utilização de certas soluções numa aplicação.

Esse estudo é feito comparando plataformas cujas diferenças se limitem apenas a um factor, obtendo assim o impacto na performance desse elemento variável. As plataformas foram desenvolvidas com soluções distintas para cada camada da arquitectura, e serão esses os elementos a analisar. Este estudo pode ser consultado no anexo 6, sendo aqui apresentadas as conclusões do

impacto em performance decorrente do uso de várias tecnologias:

- O Código Genérico da framework Tábula é 20% mais lento do que código escrito manualmente;
- Groovy é 11% mais lento que Java;
- REST Server é 8% mais lento que o uso directo de RESTlet;
- RESTlet é entre 4% a 38% mais lento que o uso de Tomcat;
- MySQL sem pooling é 141% mais lento do que com pooling;
- JPA/Hibernate/MySQL com pooling é 154% mais lento do que MySQL com pooling;
- Neodatis ODB sem pooling é 21% mais lento do que MySQL com pooling;
- Neodatis ODB sem pooling é 90% mais rápido do que MySQL sem pooling, e;
- Neodatis ODB sem pooling é 100% mais rápido do que JPA/Hibernate/MySQL com pooling.

Estes resultados são importantes para a determinação de que tecnologias e abordagens utilizar para uma aplicação tendo em conta o seu impacto na performance. Esses foram os objectivos delineados para esta secção e só foram possíveis pela abordagem escolhida nos testes, nomeadamente a construção de um grande número de plataformas com características diferentes.

A próxima secção introduz uma análise menos abrangente, limitada às plataformas semelhantes ao Tábula no que se refere a condições e metodologias.

7.5.3 Comparações com plataformas standard

A plataforma standard de referência para estes testes é sem dúvida a J (Tomcat + JPA + Hibernate + MySQL), pois recorre a componentes muito utilizados sendo a solução mais comum para persistência orientada a objectos.

Note que que todas as frameworks em Java estudadas no estado da arte basicamente adicionam camadas de funcionalidade sobre estas bases de ORM e servidor de aplicações, tais como geração automática de aplicações, ou gestão de componentes de lógica. Essas frameworks podem sem dúvida ser utilizadas com acesso directo a JDBC e expressões SQL, mas a produtividade conseguida com as soluções de ORM torna-as na abordagem preferencial.

A plataforma J configurada com as soluções mais populares para persistência (JPA+Mysql) e para servidor (Tomcat), sem mais nenhuma camada adicional de funcionalidades é mais rápida do que as restantes frameworks com código adicional. Se as plataformas com base no REST Server estiverem perto da performance da plataforma J, então estarão também dentro da gama de performances das outras frameworks.

A diferença de performance entre as plataformas com REST Server e as restantes frameworks

nestas condições de ORM e servidor, serão sempre menores do que a diferença de performance em relação à plataforma J. A comparação com a plataforma J serve então de base de referência, indicando o pior caso possível de diferenças de performance. Em relação a todas as outras frameworks, a diferença será menor.

O gráfico da Figura 29 mostra uma comparação mais detalhada entre a J, A e a B e :

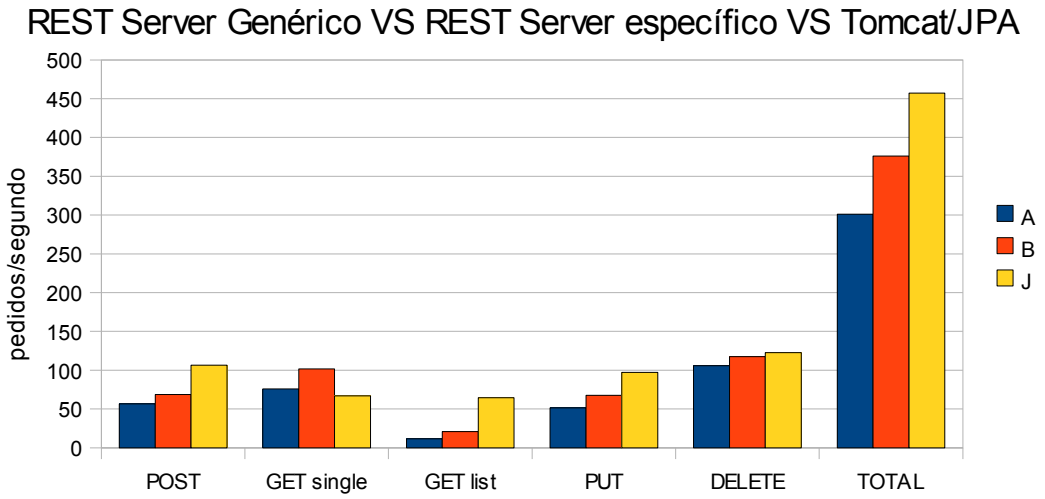

Figura 29: Comparação da performance geral entre as plataformas A, B e J

Podemos verificar uma predominância da plataforma J em quase todas as operações, com excepção do GET single. Foi visto nos testes de GET single e GET list que o impacto de performance do uso de ORM é muito maior do que nos outros testes. Isto deve-se ao custo de instanciar os objectos, efectuar uma leitura da base de dados e finalmente colocar a informação nos objectos.

Confirma-se a baixa performance do RESTlet no envio de listas de clientes, onde o Tomcat da plataform J é quase 5 vezes mais rápido que a A e 2.2 vezes mais rápido que a B.

No geral, o REST Server com solução genérica fica a 34% da performance do J com ORM, e o servidor com a solução de código específico fica a 18%.

Resumo

Baseado nos resultados dos vários testes, torna-se possível finalmente classificar a framework Tábula em relação às soluções existentes. No entanto, há que em conta os seguintes factores antes de tirar conclusões:

O REST Server:

- Foi implementado em 8 meses;
- Usa bibliotecas recentes e não optimizadas tais como RESTlet, Groovy e Neodatis ODB;
- A base de dados Neodatis ODB não tem actualmente suporte para pooling de ligações;

- Suporta alterações em *runtime* da camada de serviços, controle e persistência.
- Possui um Editor gráfico para configuração de serviços REST;
- O código Groovy pode ser alterado em runtime;
- É capaz de gerar automaticamente uma aplicação com capacidades CRUD e camada de serviços REST para acesso remoto aos objectos;
- O processo de optimização durou um dia, e;
- Está ainda em versão alpha.

Sobre a plataforma típica J(Tomcat + JPA + Hibernate + MySQL):

- As bibliotecas existem à anos;
- Possuem vários programadores activos;
- Foram optimizadas extensivamente;
- Já lançaram várias versões estáveis, e;
- Pequenas alterações forçam o reiniciar do servidor.

Considerando todos estes factores, sobretudo a flexibilidade do desenvolvimento no REST Server contra a performance fixa da solução J(Tomcat + JPA + Hibernate+ MySQL), a solução apresentada neste projecto não fica muito a perder.

Há sem dúvida espaço para aperfeiçoamento, e muitos pontos a melhorar, mas no geral há uma grande satisfação com os resultados actuais, sobretudo para uma implementação cujo objectivo inicial são era a velocidade mas ser apenas uma prova de conceito.

7.6 Conclusão

O trabalho extenso efectuado neste capítulo foi essencial para validar as especificações e a framework desenvolvida no âmbito desta dissertação. Tornou possível também estudar o impacto de várias tecnologias à performance de uma aplicação.

A análise detalhada da primeira versão do REST Server levou à identificação de secções de código responsáveis por atrasos consideráveis na resposta ao pedidos efectuados. Foram delineadas estratégias para reduzir esses atrasos, que após implementação, levaram a um aumento geral na ordem dos 1700%, ou seja, a versão optimizada é 17 vezes mais rápida.

O estudo dos resultados das várias frameworks foi trabalhoso, mas importante e esclarecedor. Tornou possível caracterizar a performance de cada plataforma e dos seus componentes. No entanto, determinar qual destas plataformas é a melhor não é uma questão que possa ser respondida apenas com estes dados de forma isolada. É necessário contextualizar a aplicação, saber os requisitos funcionais e não funcionais, bem como as necessidades de desenvolvimento.

Por exemplo, e em relação à persistência, sem dúvida que a plataforma I (Tomcat + Java + MySQL

com pooling) é a mais rápida, mas excepto em casos de aplicações muito simples, a produtividade dos programadores na implementação desta solução com SQL escrito manualmente vai ser muito inferior a qualquer solução que recorra a qualquer tipo de ORM ou bases de dados orientadas a objectos.

Da mesma forma, as plataforma J é mais rápida do que qualquer das soluções utilizando o REST Server, mas a produtividade dos programadores na implementação de aplicações centradas em serviços é muito inferior, pois a sua definição em J é estática, forçando a *restarts* para qualquer tipo de alteração. O REST Server, suportando a definição e configuração dinâmica de serviços através de um editor gráfico, oferece uma eficiência muito superior.

Em relação à performance do REST Server, o que seria de esperar num projecto neste estado actual seria uma grande diferença em relação a soluções semelhantes, provavelmente algumas vezes mais lento.

Surpreendentemente, a implementação deste projecto situa-se entre 35% a 18% da performance de uma solução típica como a plataforma J (Tomcat + JPA + Hibernate + MySQL+pooling), conforme seja usada a solução genérica ou código específico. Pode-se afirmar que a performance não está muito longe da solução mais comum da indústria.

Considerando as características do REST Server contra a plataforma J, nomeadamente o suporte extensivo a alterações dinâmicas e a utilização de bibliotecas open source recentes e não optimizadas, pode-se concluir que este resultado é muito satisfatório e valida as estratégias desenvolvidas a nível de definição de especificações, implementação e optimização.

8 Conclusões e Perspectivas Futuras

8.1 Conclusões

As aplicações web detém actualmente uma grande importância por fornecerem todas as funcionalidades disponíveis na Internet que permitem a troca e processamento de informação. Como tal, a sua criação é uma área de desenvolvimento intenso, com uma variedade de soluções disponíveis.

Nesta dissertação efectuou-se um estudo extensivo de várias frameworks para criação de aplicações web. Adicionalmente, foi feito uma análise das tendências nesse campo, com a indicação de que existem novas abordagens com grande impacto nesta área. Destacam-se a arquitectura SOFEA, o estilo arquitectural REST e as ofertas comerciais de cloud computing. Concluiu-se que as ferramentas actuais não oferecem funcionalidades adaptadas aos cenários permitidos por estas novas abordagens.

Neste trabalho foi proposta uma solução consistindo de:

- Uma especificação de configuração de aplicações REST;
- Descrição do processamento de pedidos;
- Solução para camada lógica baseada em acções como máquinas de estados finitas;
- Abordagem de AOP para *acções* com integração REST via regras, e;
- Especificação de serviços de administração dinâmico de REST e AOP.

Esta proposta foi implementada na framework Tábula, com suporte às especificações e requisitos delineados. Esta ferramenta permite criar aplicações REST por dois mecanismos: geração automática da aplicação ou configuração manual.

A geração automática pode ser facilmente definida e configurada pelos programadores, oferecendo soluções pré-definidas de persistência, mapeamento objecto-XML e processamento de pedidos. A framework é orientada à produtividade do programador, oferecendo configurações pré-definidas adaptadas aos casos comuns de desenvolvimento. Aos programadores avançados são disponibilizados mecanismos para alteração dos componentes da geração.

Adicionalmente, a solução de AOP da framework possui um sistema de administração dinâmico que permite combinar as capacidades de geração com as necessidades de funcionalidade específica.

Para o caso de definição manual da aplicação, é disponibilizada uma ferramenta gráfica para a configuração dinâmica de aplicações REST. Quanto ao processamento de pedidos, este pode ser efectuado de várias formas, com destaque para scripts Groovy ou acções estruturadas com passos implementados também em Groovy. A combinação de scripting com o carregamento automático de bibliotecas oferece um ambiente optimizado para a implementação rápida de aplicações.

A arquitectura do servidor é estruturada em camadas, com abstracção de componentes-chave o que garante a sua flexibilidade e extensibilidade. O uso de Guice para injecção de dependências conduziu a um fraco acoplamento entre componentes.

A implementação valida assim as especificações propostas, provando a sua viabilidade como forma de implementação de aplicações REST. A framework também demonstra que estas especificações, com a sua flexibilidade e dinamismo, podem ser implementadas sem uma perda drástica de performance.

A framework oferece assim uma solução completa nesta área, alcançando com sucesso os objectivos propostos.

8.2 Perspectivas Futuras

A realização de qualquer trabalho deixa sempre possibilidades de evolução em esforço futuros, tais como:

- Lançamento de versões estáveis das ferramentas gráficas para configuração de jtAction e administração de AOP;
- Integração das diversas ferramentas entretanto desenvolvidas;
- Maior abstracção do sistema de regras de AOP da camada de serviços REST;
- Suporte à solução de AOP a handlers, independentemente da implementação destes;
- Mecanismo mais fácil para adicionar suporte a novos tipo de representação nas aplicações geradas automaticamente;
- Suporte uma solução de linguagem query à PersistenceAPI, que possa ser facilmente mapeada para as implementações existentes de bases de dados;
- Suporte a campos em classes do tipo *Map* na geração automática;
- Melhoramento da performance do servidor;
- Suporte na geração automática a campos binários nas classes para guardar imagens, áudio e vídeo;
- Suporte à configuração por anotações dos campos a usar como identificador dos objectos, e a geração de valores para campos, e;
- Suporte a refactoring de classes no NeodatisODB sem ser necessário reconstruir a base de dados.
- Geração automática da interface para aplicações com base numa tecnologia RIA.

9 Referências bibliográficas

[1] ActionScript, www.actionscript.org
[2] ActiveRecord - , wiki.rubyonrails.org/rails/pages/ActiveRecord
[3] ADO.NET - , http://msdn.microsoft.com/en-us/library/aa286484.aspx
[4] Adobe, www.adobe.com
[5] Adode AIR, www.adobe.com/products/air/
[6] AJAX - asynchronous JavaScript and XML, http://en.wikipedia.org/wiki/AJAX
[7] Ajax4JSF, https://ajax4jsf.dev.java.net/
[8] Amazon, www.amazon.com
[9] Amazon EC2, , Amazon Elastic Compute Cloud, aws.amazon.com/ec2/
[10] Amazon S3 - Amazon Simple Storage Service, aws.amazon.com/s3/
[11] Amazon SQS - Amazon Simple Queue Service, aws.amazon.com/sqs/
[12] AMI - Amazon Machine Image, http://aws.amazon.com/ec2/
[13] Kiczales, Gregor et al, Aspect-Oriented Programming, Proceedings of the European Conference on Object-Oriented Programming, vol.1241. pp.pp.220–242, 1997
[14] Apache Axis 1, ws.apache.org/axis
[15] : Apache CXF - , cxf.apache.org
[16] AppDrop, http://appdrop.com/
[17] Applet, java.sun.com/applets
[18] AspectJ, www.eclipse.org/aspectj
[19] Azure, www.microsoft.com/azure/
[20] Back-end, http://en.wikipedia.org/wiki/Front-end
[21] Bechmarks da framework Tábula, http://tabula.softmed.org/?page_id=65
[22] : blinksale - , www.blinksale.com
[23] Blue Cloud, www.ibm.com/press/us/en/pressrelease/22613.wss
[24] BTP - Business Transaction Protocol, www.oasis-open.org/committees/business-transactions/
[25] Brian K.; Dennis R., The C Programming Language, 1st ed., Englewood Cliffs, NJ: Prentice Hall. ISBN 0-13-110163-3, 1978
[26] Cdyne, www.cdyne.com
[27] CERN - Centre d'Etudes et de Recherche Nucléaires, www.cern.ch
[28] CGI - Common Gateway Interface, http://www.w3.org/CGI/
[29] Client Server Architecture, http://en.wikipedia.org/wiki/Client-server
[30] Cloud - Cloud Computing, http://en.wikipedia.org/wiki/Cloud_computing
[31] Codebehind - plugin experimental ao suporte de Web Services, struts.apache.org/2.0.6/docs/codebehind-plugin.html
[32] COM - Component Object Model , http://www.microsoft.com/com/
[33] Convention over Configuration, http://en.wikipedia.org/wiki/Convention_over_Configuration
[34] CORBA - Common Object Requesting Broker Architecture , www.corba.org
[35] W. Stevens, G. Myers, L. Constantine, Structured Design, IBM Systems Journal, 13 (2), 115-139, 1974
[36] CRUD - Create Read Update Delete, http://en.wikipedia.org/wiki/Create,_read,_update_and_delete
[37] McCabe T. J., A Complexity Measure, IEEE Transactions on Software Engineering, vol. se-2, nº.4, 1976
[38] DAO - Data Access Object, http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html
[39] DB4O, www.db4o.com
[40] Evans, E., Domain-Driven Design - Tackling Complexity in the Heart of Software, Addison-Wesley, 2004
[41] Definição de Web Service pelo W3C, http://www.w3.org/TR/ws-gloss/

[42] DI - Dependency Injection, http://en.wikipedia.org/wiki/Dependency_injection
[43] DOM, W3C, Document Object Model, www.w3.org/DOM
[44] Drools, www.jboss.org/drools/
[45] Hunt, A.;Thomas, D., The Pragmatic Programmer: From Journeyman to Master, ISBN 0-201-61622-X , Addison-Wesley, 2000
[46] DWR - Direct Web Remoting, directwebremoting.org/
[47] ebXML - Electronic Business using eXtensible Markup Language, www.ebxml.org
[48] Eclipse, www.eclipse.org
[49] EJB - Enterprise JavaBeans, java.sun.com/products/ejb/
[50] EL - Expression Language, java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPIntro7.html
[51] Enhydra Shark, shark.enhydra.org
[52] Entity Framework, http://msdn.microsoft.com/en-us/library/aa697427(VS.80).aspx
[53] Enunciate, enunciate.codehaus.org/
[54] ESB - enterprise service bus, http://en.wikipedia.org/wiki/Enterprise_service_bus
[55] Facebook, www.facebook.com
[56] Facelets, https://facelets.dev.java.net
[57] Fat client, http://en.wikipedia.org/wiki/Fat_client
[58] Flex, Adobe, , http://www.adobe.com/products/flex/
[59] Flickr, www.flickr.com
[60] FQCN - Fully Qualified Class Name, http://www.onjava.com/pub/a/onjava/2005/01/26/classloading.html
[61] Framework - Software Framework, http://en.wikipedia.org/wiki/Software_framework
[62] FreeMarker, freemarker.sourceforge.net
[63] : Frevvo - , www.frevvo.com
[64] Front-end, backend
[65] FTP - File Transfer Protocol , http://tools.ietf.org/html/rfc959
[66] GIOP - General Inter-ORB Protocol, http://www.omg.org/spec/CORBA/3.1/
[67] GlassFish, https://glassfish.dev.java.net/
[68] Google - , www.google.com
[69] Google App Engine, code.google.com/appengine/
[70] Google Calendar, www.google.com/calendar
[71] GORM - Grails Object/Relational Mapper, grails.org/GORM
[72] Grails, grails.org
[73] Groovy, groovy.codehaus.org
[74] GSP - Groovy Server Pages, docs.codehaus.org/display/GRAILS/Developer+-+Groovy+Server+Pages
[75] Guice, http://code.google.com/p/google-guice/
[76] Hibernate, www.hibernate.org
[77] Hivemind, hivemind.apache.org
[78] Hollywood Principle, http://en.wikipedia.org/wiki/Hollywood_Principle
[79] Hot Swap - , pt.wikipedia.org/wiki/Hot_swapping
[80] HQL - Hibernate Query Language, http://www.hibernate.org/hib_docs/v3/reference/en/html/queryhql.html
[81] HTML - HyperText Markup Language, W3C, http://www.w3.org/MarkUp/
[82] HTTP - Hypertext Transfer Protocol, http://www.w3.org/Protocols/
[83] httpd, CERN, HTTP Daemon, http://www.w3.org/Daemon/
[84] Ibatis, ibatis.apache.org
[85] IBM, www.ibm.com
[86] ICEfaces, www.icefaces.org/
[87] IDL - Interface Definition Language, http://en.wikipedia.org/wiki/Interface_definition_language

[88] IIOP - Internet Inter-Orb Protocol , http://www.omg.org/library/iiop4.html
[89] IoC - Inversion of Control, http://en.wikipedia.org/wiki/Inversion_of_control
[90] J2EE, SUN, Java 2 Platform Enterprise Edition, asdasd
[91] J5EE, SUN, Java 5 Enterprise Edition, http://java.sun.com/javaee/technologies/javaee5.jsp
[92] Java, java.sun.com
[93] Java Reflection API, http://java.sun.com/docs/books/tutorial/reflect/
[94] JavaFX, SUN, , www.sun.com/software/javafx/
[95] JAX-RS - Java API for RESTful Web Services , https://jsr311.dev.java.net/
[96] JAX-WS - Java API for XML Web Services, https://jax-ws.dev.java.net/
[97] JAXB - Java Architecture for XML Binding, https://jaxb.dev.java.net/
[98] JAXR - Java API for XML Registries, java.sun.com/webservices/jaxr/index.js
[99] JBoss microcontainer, labs.jboss.org/jbossmc
[100] JBoss Portal, www.jboss.org/jbossportal/
[101] jBPM, www.jboss.com/products/jbpm
[102] JDBC, java.sun.com/javase/technologies/database/
[103] JDO - Java Data Objects, java.sun.com/jdo/
[104] Jersey - implementação open source de JAX-RS, https://jersey.dev.java.net/
[105] JMS - Java Message Service, java.sun.com/products/jms/
[106] JPA - Java Persistence API, http://java.sun.com/javaee/technologies/persistence.jsp
[107] JPQL, SUN, Java Persistence Query Language , http://java.sun.com/mailers/techtips/enterprise/2006/TechTips_Oct06.html
[108] JSF, java.sun.com/javaee/javaserverfaces/
[109] JSON - JavaScript Object Notation, www.json.org
[110] JSP, SUN, Java Server Pages, https://java.sun.com/products/jsp
[111] LINQ - Language Integrated Query, msdn.microsoft.com/en-us/netframework/aa904594.aspx
[112] Mashup, http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)
[113] Metro - open source java web service stack, https://metro.dev.java.net/
[114] Microsoft, www.microsoft.com/
[115] MIME - Multipurpose Internet Mail Extensions, http://en.wikipedia.org/wiki/MIME
[116] MOM - Message Oriented Middleware, http://en.wikipedia.org/wiki/Message_Oriented_Middleware
[117] Moonlight, www.mono-project.com/Moonlight
[118] Mosaic, http://en.wikipedia.org/wiki/Mosaic_(web_browser)
[119] MVC, Trygve M. H. R., Model View Controller, http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html
[120] MXML , http://www.adobe.com/devnet/flex/articles/paradigm.html
[121] MySQL, www.mysql.com
[122] Naked Objects, www.nakedobjects.org
[123] NeodatisODB, odb.neodatis.org/
[124] NetBeans, www.netbeans.org
[125] OASIS - Organization for the Advancement of Structured Information Standards, Organization for the Advancement of Structured Information Standards
[126] OASIS - Organization for the Advancement of Structured Information Standards, www.oasis-open.org/
[127] OGNL - Object Graph Navigation Language, www.ognl.org
[128] on demand - , en.wikipedia.org/wiki/On-demand
[129] OOP - Object Oriented Programming, http://en.wikipedia.org/wiki/Object-oriented_programming
[130] OpenSymphony, http://www.opensymphony.com/
[131] ORM - Object Relational Mapping, http://en.wikipedia.org/wiki/Object-relational_mapping
[132] PARC - Palo Alto Research Center, http://www.parc.com/

[133] Perl, www.perl.org
[134] PHP - PHP:Hypertext Processor, www.php.net
[135] Picocontainer, www.picocontainer.org
[136] pipes, http://en.wikipedia.org/wiki/Pipe_(Unix)
[137] POJO - Plain Old Java Object, http://en.wikipedia.org/wiki/POJO
[138] Poster - plugin para o Firefox, https://addons.mozilla.org/en-US/firefox/addon/2691
[139] Prototype - , www.prototypejs.org/
[140] REST - Representational State Transfer, Roy T. Fielding, http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
[141] REST Client, code.google.com/p/rest-client/
[142] RESTlet,
[143] rHTML, wiki.rubyonrails.org/rails/pages/UnderstandingViews
[144] RI - Reference implementation, http://en.wikipedia.org/wiki/Reference_implementation
[145] : RIA - Rich Internet applications, http://en.wikipedia.org/wiki/Rich_Internet_application
[146] RIFE, rifers.org
[147] : RMI - Remote Method Invocation,
[148] RoR - Ruby on Rails, http://www.rubyonrails.org/
[149] RPC - Remote Procedure Call , http://tools.ietf.org/html/rfc1057
[150] Ruby - , ruby-lang.org/
[151] SAAJ - SOAP with Attachments API for Java, https://saaj.dev.java.net
[152] SaaS - Software as a Service, http://en.wikipedia.org/wiki/Software_as_a_service
[153] script.aculo.us - , script.aculo.us
[154] SEAM, www.jboss.com/products/seam
[155] SecureConversation - , docs.oasis-open.org/ws-sx/ws-secureconversation/200512/ws-secureconversation-1.3-os.html
[156] Server farm, http://en.wikipedia.org/wiki/Server_farm
[157] Servlet 1.0, http://java.sun.com/products/servlet/
[158] Silverlight, Microsoft, , http://silverlight.net/
[159] SLOC - Source Lines of Code, http://en.wikipedia.org/wiki/Source_lines_of_code
[160] Smalltalk, www.smalltalk.org
[161] SMC - State Machine Compiler, smc.sourceforge.net
[162] SMTP - Simple Mail Transfer Protocol, http://tools.ietf.org/html/rfc772
[163] SOA - Service Oriented Architecture , http://en.wikipedia.org/wiki/Service-oriented_architecture
[164] SOAP - Simple Object Access Protocol, W3C, http://www.w3.org/TR/soap/
[165] SOAP UI, www.soapui.org
[166] SOFEA, Prasad, G.; Taneja, R.; Todanka, V., Service-Oriented Front-End Architecture, http://wisdomofganesh.googlegroups.com/web/Life%20above%20the%20Service%20Tier%20v1_0.pdf
[167] SOUI, Nolan Wright, Service Oriented UI, http://www.appcelerant.com/mvc_is_dead.html
[168] Spring, http://www.springframework.org/
[169] Spring-WS - Spring Web Services , static.springsource.org/spring-ws/sites/1.5/
[170] Springsource - SpringSource Application Platform, http://www.springsource.com/products/suite/applicationplatform
[171] Stripes, mc4j.org/confluence/display/stripes/Home
[172] Struts 1, struts.apache.org
[173] Struts 2, struts.apache.org/2.x/
[174] SUN, www.sun.com
[175] SWING, java.sun.com/docs/books/tutorial/uiswing/
[176] Tabula web framework, http://code.google.com/p/tabulasoftmed/
[177] Tapestry, tapestry.apache.org

[178] Thin client, http://en.wikipedia.org/wiki/Thin_client
[179] Tiles, tiles.apache.org
[180] Tomcat, tomcat.apache.org
[181] TopLink, http://www.oracle.com/technology/products/ias/toplink/index.html
[182] Trails, www.trailsframework.org
[183] Tutoriais Vídeo da framework Tabula, http://tabula.softmed.org/?page_id=45
[184] Tutoriais Vídeo da framework Tabula, http://code.google.com/p/tabulasoftmed/
[185] UDDI - Universal Description, Discovery and Integration, uddi.xml.org/
[186] UI, http://en.wikipedia.org/wiki/User_interface
[187] UN/CEFACT - United Nations Centre for Trade Facilitation and Electronic Business, www.unece.org/cefact/
[188] URI - Universal Resource Indicator, http://tools.ietf.org/html/rfc3986
[189] Utility computing - , http://en.wikipedia.org/wiki/Utility_computing
[190] Velocity - , velocity.apache.org/moving.html
[191] W3C - , www.w3.org
[192] WADL - Web Application Description Language, https://wadl.dev.java.net/
[193] Web 2.0, http://en.wikipedia.org/wiki/Web_2.0
[194] Webstart, java.sun.com/javase/technologies/desktop/javawebstart/index.jsp
[195] WebWork 2, http://www.opensymphony.com/webwork/
[196] Wicket, wicket.apache.org
[197] Workflow - , http://en.wikipedia.org/wiki/Workflow#Historical_development
[198] WPF - Windows Presentation Foundation , windowsclient.ne
[199] WS - Web Services, http://www.w3.org/2002/ws/
[200] WS-Atomic Transaction, http://docs.oasis-open.org/ws-tx/wstx-wsat-1.1-spec-os/wstx-wsat-1.1-spec-os.html
[201] WS-BP - WS-I Basic Profile, www.ws-i.org/Profiles/BasicProfile-1.0.html
[202] WS-Coordination, http://docs.oasis-open.org/ws-tx/wscoor/2006/06/
[203] WS-I - Web Services Interoperability Organization, www.ws-i.org
[204] WS-ReliableMessaging, http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.1-spec-os-01.pdf
[205] WS-Security, www.oasis-open.org/committees/wss/
[206] WS-SECURITYPOLICY, WS-SECURITYPOLICY
[207] WS-Trust - , docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.html
[208] WSDL - Web Service Description Language, www.w3.org/TR/wsdl
[209] WSIT - Web Services Interoperability Technology, java.sun.com/webservices/reference/tutorials/wsit/doc/index.html
[210] WWW - World Wide Web, http://en.wikipedia.org/wiki/World_Wide_Web
[211] XAML - Extensible Application Markup Language, http://windowssdk.msdn.microsoft.com/en-us/library/ms752059.aspx
[212] Xfire, xfire.codehaus.org/
[213] XML, W3C, eXtensible Markup Language, w3c.org
[214] XML-RPC, www.xmlrpc.com
[215] XMLHttpRequest, www.w3.org/TR/XMLHttpRequest/
[216] XSLT - XSL Transformations, www.w3.org/TR/xslt
[217] XStream, xstream.codehaus.org
[218] Yahoo, www.yahoo.com
[219] YouTube, www.youtube.com
[220] ZohoCRM, crm.zoho.com/
[221] Hunt, A.; Thomas, D., The Pragmatic Programmer: From Journeyman to Master, ISBN 0-201-61622-X, Addison-Wesley, 2000
[222] Kiczales, Gregor et al, Aspect-Oriented Programming, Proceedings of the European

Conference on Object-Oriented Programming, vol.1241. pp.pp.220–242, 1997
[223] Erich Gamma; et al , Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995
[224] Microsoft's Q196271, http://support.microsoft.com/kb/196271
[225] .NET - , www.microsoft.com/net/
[226] Moritz, F., *Rich Internet Applications (RIA): A Convergence of User Interface Paradigms of Web and Desktop Exemplified by JavaFX*, Diploma Thesis, University of Applied Science, Kaiserslautern, Germany, Janeiro 2008
[227] Darwin, I. F., *Java Web MVC Frameworks:Background, Taxonomy, and Examples*, M. Sc., Staffordshire University, 2004

Anexo I - Descrição de Frameworks

➢ Struts 1.1

Lançado em Julho 2001, o Struts [172] é uma framework open source Java para aplicações web que segue a metodologia MVC. É a framework dominante de momento, embora em declínio. Foi desenvolvida por especialistas da área de aplicações web, nomeadamente Craig McClanahan, da SUN, e foi doada à fundação Apache em 2000. A sua última versão é a 1.3.9.

O seu uso leva à separação de responsabilidades típica do MVC, embora o Struts em si consista na parte de lógica de negócio da aplicação. O objectivo do Struts foi reduzir a complexidade de configuração do J2EE, deixando de usar os *Enterprise Java Beans* (EJB)[49] e apostando numa arquitectura de lógica baseada em acções.

Lógica
O Struts aplica o padrão *Front Controller* em que um pedido de um cliente passa por um Servlet central que verifica o URL do pedido e vai buscar um *RequestProcessor* associado especificamente a esse URL. Esse *RequestProcessor* por sua vez vai obter a acção associada ao pedido, implementada por um *ActionBean*. O *ActionBean* é executado e devolve uma indicação de que página devolver como resposta.

Apresentação
É possível usar Servlets, JSP, templates para Velocity[190], XSLT[216], embora o mais comum seja utilizar JSPs. Possui várias bibliotecas de tags JSP que permitem utilizar aceder à funcionalidade da framework a partir da interface gráfica, com suporte à JSP 2.0 Expression Language (EL) [50]:

- HTML - elementos como botões, *checkbox*, etc;
- Bean - aceder a JavaBeans e suas propriedades;
- Logic - geração condicional de texto como *loops;*
- Nested - permite que as *tags* interiores não precisem de declarar tantos atributos, já que foram declarados pelas *tags* acima. O objectivo é reduzir o código;
- Template - uso de *templates* JSP, e;
- Tiles - mecanismo avançado de *templates*. Suporta *layouts* através de *tiles*.

Acesso a dados
Não possui suporte específico. Depende das escolhas para a aplicação. Por exemplo, pode ser usado JPA, Hibernate, JDO[103], JDBC[102]ou outros.

➢ Spring MVC

A versão 1.0 foi lançada em Março 2004, embora a primeira implementação tivesse sido publicada no final de 2002 por Rod Johnson [168].

Foi referido anteriormente que o Spring implementou o que é considerado uma boa prática essencial hoje em dia, que é a injecção de dependências (DI)[42] ou inversão de controle (IoC) [89].

Para além da DI a framework Spring adiciona várias utilidades adicionais, tais como uma camada de abstracção JDBC que permite executar SQL facilmente com apenas algumas linhas de código, bem como lidar com as excepções inerentes.

Foi adicionado o suporte a AOP através de *proxies* – no caso de interfaces – ou de manipulação de *bytecode,* ou seja, classes já compiladas. Esta abordagem permite interceptar a execução de qualquer método, e adicionar funcionalidades antes ou depois da sua execução. Na prática estão a ser adicionados aspectos à aplicação usando o AspectJ[18].

Em termos de MVC, o Spring é semelhante ao Struts, mas usa controladores em vez das acções. Um dos seus aspectos positivos é usar interfaces para os controladores em vez de forçar herança como no caso do Struts e também suporta interceptores de forma semelhante ao Struts 2.

De notar que o Spring cresceu de tal forma, sobretudo com a framework MVC que é hoje considerado uma forte alternativa ao Java EE 5. A aquisição recente da companhia *open source* Covalent, que providencia suporte aos produtos da *Apache Software Foundation* só veio reforçar ainda mais essa ideia. Uma aplicação web pode ser desenvolvida com Spring MVC + Tomcat com garantias de boa arquitectura e suporte, em alternativa ao servidor completo e mais pesado para Java EE 5. A aquisição da Covalent coloca à disposição do Spring um grande potencial em termos de integração, performance e suporte com o Tomcat.

Lógica
O `org.springframework.web.servlet.DispatcherServlet` é o Front Controller que recebe os pedidos e os envia aos controladores respectivos conforme a configuração da aplicação. Os controladores implementam a interface `org.springframework.web.servlet.mvc.Controller`.

Apresentação
Embora suporte vários tipos de apresentação, o suporte é dado sobretudo aos JSPs bem como a várias tecnologias relacionadas, como Tiles[179], Velocity e FreeMarker[62].

Acesso a dados
Em termos de ORM, suporta de base standards como JPA, JDO 1 e 2, bibliotecas como Hibernate, TopLink e Ibatis[84]. A integração do Spring com estas soluções permite, por exemplo, facilitar a configuração dos ficheiros de persistência do JPA.

➢ Java EE 5

A plataforma Java EE 5[91] foi lançada em Maio de 2006 com várias melhorias à construção de aplicações, sobretudo em relação à versão anterior, J2EE.

Usando apenas anotações, é agora possível configurar a persistência, EJB, *Web Services* e injecção de dependências. A SUN na prática adicionou a sua própria versão dos aspectos mais populares de ferramentas *open source* para Java como o Hibernate e o Spring, lançando várias especificações tais como EJB 3.0, JSF 1.2, JSP 2.1, Servlets 2.5 e JPA 1.0.

De destacar a persistência, onde foi criada a especificação Java Persistence API (JPA)[106] com várias implementações tais como o Hibernate e TopLink. Foi um reconhecimento por parte da SUN do modelo de persistência desenvolvido pelo Hibernate e que pode ser usado até em aplicações de *desktop*.

Embora seja de facto uma passo na direcção correcta, e um grande avanço em relação ao J2EE, desenvolver aplicações web nesta plataforma continua a ser uma tarefa complexa, mesmo se a aplicação for de pequenas dimensões.

Java EE 5 é melhor orientado a grandes aplicações. Para tal, são necessários conhecimentos de gestão de componentes, transacções, persistência, injecção de dependências, ciclos de vida de objectos, responsabilidades do contentor da aplicação, escalabilidade e vários outros aspectos.

Muitos destes factores são geridos pelo servidor da aplicação, o que deixa ao programador a tarefa de configurar os serviços que deseja, ao invés de implementar abordagens para gestão de transacções, ciclos de vida de objectos, persistência e outros.

De facto, o uso de anotações em vez de XML tornam a construção da aplicação mais simples, ao custo da adição de meta-dados de configuração ao código. Por exemplo, a configuração da persistência leva à adição de anotações específicas à API de persistência a várias classes do modelo de objectos de domínio.

Se por um lado o uso de anotações reduz imenso o número de linhas de código necessário à configuração, por outro, adiciona dependências às bibliotecas que definem as anotações usadas. Tendo em conta que um dos objectivos de um bom código é a separação de responsabilidades, então aparentemente as anotações violam esse princípio.

Lógica

A lógica é providenciada por Enterprise Java Beans(EJB). Os EJB's na sua versão 3.0 podem ser Plain Old Java Objects (POJOs), ou seja não é necessário criar subclasses de classes da API, bastando usar anotações para configurar. De notar que existem vários tipos de EJBs, nomeadamente:

- *Entity beans* - fazem o mapeamento da base de dados para uma classe. São as classes configuradas para persistência através do JPA;
- *Stateful Session Bean* - subsiste na sessão do cliente e mantém o seu estado;
- *Stateless Session Bean* - executam a sua função, mas não mantém o seu estado. São usadas

para acções curtas. O servidor terá uma *pool* destes objectos, obtendo um, executando e devolvendo à *pool, e;*

- *Message Driven Bean* - recebem mensagens via JMS e processam-nas quando possível.

Apresentação
Embora não force nenhuma tecnologia específica, podendo ser usado JSP, JSF, ou outras tecnologias, é dado um grande foco ao JSF.

De notar que o JSF requer que exista uma classe Java associada a cada interface JSF, a que se denomina *backing bean*. É através da *Expression Language* (EL) que o JSF acede aos atributos do *backing bean*, podendo também chamar métodos desse mesmo objecto. As funções possíveis dos *backing beans* também incluem efectuar validação, lidar com eventos da interface ou determinar qual a próxima página.

Acesso a dados
Utiliza o Java Persistence API (JPA) para gerir o acesso a dados. É uma especificação da SUN, da qual há várias implementações disponíveis, como o Hibernate, TopLink, etc. Este conceito já foi introduzido na secção *ORM e Persistência*.

➢ Seam

A primeira versão de Seam [154] foi lançada em Setembro de 2005 e desde junho de 2006 está disponível a segunda versão.

O objectivo desta framework é facilitar a construção de aplicações em Java EE 5. A configuração e integração dos EJB e JSF é fortemente reduzida através do uso de anotações adicionais específicas ao Seam e bibliotecas auxiliares. Por exemplo, deixa de ser necessário criar um *backing bean* para cada página JSF. Por outro lado, de momento só suporta o JSF como tecnologia de apresentação.

A injecção de dependências é usada mais extensivamente do que no J5EE, sendo todos os componentes geridos e injectados nas páginas/objectos correctos em *runtime*. O resultado é uma framework orientada para configuração por excepção, ou convenção sobre configuração, tal como Ruby on Rails (RoR).

De forma semelhante, o Seam também possui um gerador automático de aplicações a partir de uma base de dados existente. Embora em Seam o uso de XML seja mais reduzido do que no J5EE, o XML ainda é utilizado na configuração da aplicação, no fluxo das páginas e nos processos de negócio.

Esta framework ainda coloca à disposição várias ferramentas adicionais tais como jBPM[101], JBoss Rules (a.k.a Drools)[44], JBoss Portal[100], JBoss Microcontainer[99] entre outros.

Lógica
A implementação da lógica da aplicação pode ser feita via EJBs ou POJOs. Os EJB's requerem algum esforço adicional, tal como indicar se é *statefull* ou *stateless*, e obriga a implementar uma interface. Em compensação são totalmente geridos pelo servidor, tendo acesso a vários serviços integrados.

Os POJOs são mais simples, pois não é forçada nenhuma interface adicional, possibilitando testar cada componente de forma isolada fora da aplicação. Por outro lado, os POJOs perdem uma grande quantidade de serviços e capacidades que são automaticamente geridos pelo servidor da aplicação.

Apresentação
De momento o Seam só suporta JSF como tecnologia de apresentação. Podem ser usados usados templates com Facelets[56], e AJAX através da integração de base de duas bibliotecas de componentes AJAX para JSF: ICEfaces[86] e Ajax4JSF[7].

Acesso a dados
Sendo o Seam da autoria dos criadores do Hibernate, esta é a tecnologia suportada de base, quer através da especificação JPA quer pelo uso directo da biblioteca. De referir a anotação @Name que regista a classe como *entity bean* na framework, e a torna disponível para ser usada directamente no JSF ou em registo de configuração sem nenhum intermediário adicional, como seria no caso do J5EE.

➢ Struts 2

Struts 2[173] foi lançado em Fevereiro de 2007 e resulta da combinação do Struts e do WebWork 2 [195] da OpenSymphony[130]. Mostra grandes avanços em relação ao Struts 1, sobretudo no suporte ao uso de anotações para a configuração, diminuindo a necessidade de ficheiros XML. Outros melhoramentos incluem o suporte à injecção de dependências via integração com Spring, a escolha do JSF para a vista, com suporte a AJAX e melhor suporte a testes.

O Struts 2 continua a ser uma framework orientada à acção, com mapeamento de acções a URLs. Em relação ao Struts 1, o *ActionServlet* é substituído por um filtro, o *FilterDispatcher.* As acções são subclasses de *Action* ou de *ActionSupport,* definidas em *com.opensymphony.xwork2*. A classe *ActionSupport* é uma classe utilitária com vários métodos adicionais para facilitar a implementação.

Foram adicionados interceptores, que podem ser executados antes ou depois da invocação de uma acção por forma a disponibilizar uma solução de AOP. Os interceptores são uma implementação do padrão *decorator* e pode ser considerado também o *chain of responsibility* . Existem de base vários interceptores que permitem criar mensagens de *log*, até fazer *profiling* da execução.

Os interceptores podem ser organizados em pilhas, onde a ordem de declaração é importante. Adicionalmente, cada acção pode definir seus próprios interceptores. Novos interceptores podem

ser criados através de subclasses de `com.opensymphony.xwork2.interceptor.Interceptor`.

É ainda de referir o uso de um objecto to tipo *Value Stack,* uma pilha de variáveis para guardar a acção e os parâmetros necessários para a sua execução. Como uma pilha normal, é possível adicionar e remover objectos. Uma acção a ser executada é colocada no topo da pilha. Esta pilha pode ser referenciada a partir de qualquer ponto da framework, desde os interceptores, ao resultados, ao próprio JSP usando a Object Graph Navigation Language (OGNL)[127]. A OGNLé utilizada como Expression Language (EL) nos JSPs em vez o próprio JSP EL. A função da EL é obter e colocar propriedades de objectos Java nos JSPs sem usar código Java directamente.

Lógica
A lógica na framework Struts 2 é efectuada através das acções e dos interceptores.

Apresentação
Utiliza JSP, com suporte a JSF. Pode ser integrado com motores de *templates* como FreeMarker ou Velocity. Usa ONGL como linguagem de expressão nos JSP. Suporta AJAX. Disponibiliza ainda um conjunto de tags adicionais para os JSP:

- General tags;
- Controle de fluxo;
- Manipulação de dados na *value stack*;
- Internacionalização;
- HTML tags;
- Exibição de dados nas páginas, e;
- Montagem de formulários HTML.

Acesso a dados
Não possui suporte específico. Depende da escolha para a aplicação. Por exemplo, podem ser usado JPA, JDO, Hibernate, JDBC ou outros.

➢ Ruby On Rails

Lançado em Julho de 2004, o Ruby on Rails (RoR)[148] é uma framework *open source* escrita usando a linguagem de programação orientada a objectos Ruby[150], criada no início da década de 1990. O objectivo é a criação fácil de aplicações web usando a arquitectura MVC.

O RoR é conhecido pela sua abordagem *Convention over Configuration*[33], ou seja, a framework foi desenvolvida para produzir aplicações CRUD[36], gerando quase todo o código, estrutura de directórios e configurações necessárias seguindo o padrão mais comum para este tipo de aplicações. O programador codifica as excepções às regras e/ou os detalhes adicionais. Outra abordagem utilizada consiste em *Don't Repeat Yourself* (DRY)[221], em que o objectivo é reduzir tanto quanto possível a necessidade de configurações ou implementação de aspectos idênticos ou

semelhantes por forma a reduzir esforço e maximizar a re utilização. Em conjunto com a capacidade de síntese da linguagem Ruby, o RoR oferece uma grande produtividade.

Esta framework teve um grande impacto na área de desenvolvimento de aplicações, pois ao contrário da maioria das frameworks, desde o início o objectivo foi facilitar o esforço dos programadores ao contrário de criar uma arquitectura flexível que requer muito esforço de configuração adicional.

A estratégia *Convention over Configuration* reflecte-se por toda a framework, fazendo uso de padrões comuns na organização e desenvolvimento de aplicações. É importante referir alguns aspectos onde são aplicados esses princípios:

Em primeiro lugar, uma aplicação RoR tem tipicamente a seguinte estrutura de directórios:

- Controller - todos os controladores da aplicação;
- Views - todos os ficheiros que definem a interface gráfica, e;
- Model - todos os modelos a serem usados para persistência.

Ao agrupar os ficheiros desta forma, o RoR implementa desde o início a noção de separação de responsabilidades, e permite uma fácil identificação da função destes ficheiros, o que possibilita outros automatismos da framework.

Um exemplo destes automatismos é a associação de um controlador a um modelo apenas pelo nome deste. Se existir uma classe definida num ficheiro *Client.rb* do directório *model*, o controlador que gere esses clientes estará no ficheiro *client_controller.rb* do directório *controller*. De forma similar, a cada método do controlador é associada uma vista definida num ficheiro *rhtml* guardada no directório *views/client*.

Este é um exemplo de *configuration por default*, pois o sistema está à espera que o programador siga estas convenções, pelo que não necessita de declarar a configuração nestes casos. Em contraste, a mesma aplicação em J5EE necessitaria, no mínimo, do uso de várias anotações para efectuar esta associação.

Adicionalmente o RoR possui ferramentas de consola que permitem ao utilizador gerar toda uma aplicação ou apenas os componentes MVC necessários através de comandos simples.

Finalmente o RoR suporta a criação de ambientes diferentes para as fases de desenvolvimento, teste e produção usando bases de dados distintas para cada ambiente.

Como resultado destas metodologias, uma das características desta framework é a inexistência de ficheiros de configuração em XML. A aplicação é configurada pela sua estrutura e código.

Várias outras frameworks já adoptaram esta estratégia *Convention over Configuration*, com ganhos

significativos na usabilidade e produtividade.

Como o Ruby é uma linguagem de *scripting*, a framework RoR adicionou o suporte a alterações na aplicação *on-the-fly*, desde a apresentação, os controladores ou mesmo o modelo, reduzindo assim a necessidade de reiniciar o servidor para aplicar alterações básicas.

Lógica
A lógica é implementada em subclasses de *ApplicationController. A*pós execução de um dos seus métodos, renderizam um *template* ou redireccionam para outra acção. Por definição, renderizam um *template* existente no directório *app/views* com o nome do controlador e da acção executada. Os controladores têm acesso aos dados de pedido e sessão.

Apresentação
A apresentação é definida em ficheiros *rhtml* que se assemelham a JSP, utilizando no lugar do Java o Ruby. Por convenção, o nome do ficheiro corresponde ao do método de um controlador, e se estiverem associados à acções CRUD de um modelo, estão dentro de um subdirectório de *views* com o nome do controlador a que se referem.

As vistas podem ser de três tipos : *layouts*, *templates* ou *partials:*

- Layout: código comum a todas as acções, normalmente o inicio e o fim do HTML;
- Template: código específico da acção, por exemplo o código especifico para o "*edit*" ou "*list*", e;
- Partial: código comum, utilizado em diversas acções, como por exemplo uma tabela para a estruturação de um formulário.

O RoR suporta AJAX recorrendo às bibliotecas de JavaScript Prototype[139] and Script.aculo.us[153].

Acesso a dados
É de referir o sistema de ORM suportado pelo RoR, denominado ActiveRecord[2]. *O* ActiveRecord lida com a persistência e mapeamento dos objectos para bases de dados relacionais e pode ser visto como o JPA do RoR.

Este sistema depende fortemente da convenção. Por exemplo o nome da tabela reflectirá o nome da classe. Mas é possível usar outras definições, através de configurações adicionais. Suporta validação, associações entre classes e transacções.

Um dos mecanismos interessantes disponíveis é o uso de migrações para controle de versões da base de dados. Uma migração é uma classe que executa tarefas como criar tabelas, índices, popular campos, adicionar dados, etc. Esta abordagem permite efectuar mudanças incrementais à base de dados.

É de boa prática limitar cada migração a uma tarefa específica e bem limitada, para evitar problemas e simplificar o desenvolvimento iterativo.

➢ Grails

Esta framework recente teve a sua primeira versão lançada em Fevereiro de 2008. A ideia do Grails[72]é emular o RoR mas sobre a plataforma Java usando o Groovy como linguagem de script. Este projecto usa ferramentas já existentes como Spring e Hibernate, que são considerados standards como base para facilitar a implementação da framework.

O Grails segue o exemplo do RoR, implementando a metodologia de *convention over configuration.* A estrutura de directórios é semelhante à do RoR, onde em vez de existir *app/model*, *app/controller* e *app/views existem grails-app/domain*, *grail-app/controllers* e *grails-app/views*, respectivamente.

A associação dos modelos aos controladores e as acções dos controladores à vista atraves dos seus nomes segue o mesmo princípio e tal como o RoR, o Grails suporta instruções pela linhas de comandos com a capacidade de gerar automaticamente *scafolds* ou estruturas de suporte à aplicação, tais como a estrutura de directórios para um novo projecto, ou os ficheiros para uma nova classe de domínio a ser persistida, e a partir desta, o controlador e páginas. As páginas são Groovy Server Pages (GSP)[74] onde o HTML contém código Groovy embebido.

Basicamente, mantém a capacidade do RoR de produzir aplicações simples com um mínimo de código, mas desta vez, usando Groovy.

Lógica
A lógica da aplicação é definida em scripts groovy que definem vários métodos. Após execução de um dos seus métodos, renderizam uma página ou redirecionam para outra acção.

Apresentação
A interface gráfica é definida usando GSP ou seja Groovy Server Pages. É semelhantes a JSP, excepto que suporta tags <g> que indicam o código groovy a executar, ou componentes a adicionar.

Acesso a dados
Usa o Grails Object/Relational Mapper (GORM) [71]baseado no Hibernate. Suporta alterações em runtime, desde que sejam básicas.

➢ Tapestry

Tendo a primeira versão sido lançada em 2003, a ideia central do Tapestry[177] é abstrair completamente a programação típica de aplicações web, envolvendo URLs, parâmetros, métodos HTTP como GET e POST.

A versão 5 desta framework adicionou várias novidades, tais como: suporte a alterações em tempo real, uso de anotações em vez de XML, e maior integração de componentes AJAX. Esta versão corta com a compatibilidade da versão anterior.

A framework permite escrever toda a aplicação usando apenas templates HTML - não JSPs - classes Java e ficheiros de configuração xml. As classes Java providenciam funcionalidade, o template HTML a estrutura da apresentação e o ficheiro de configuração a ligação entre os dois. A versão 4 e 5 do Tapestry reduz a necessidade de configuração das páginas usando anotações na classe Java correspondente à página.

O propósito da framework é criar aplicações de forma simples e directa. Logo é orientada para a apresentação da aplicação, não indicando nenhuma metodologia específica para implementação de componentes de lógica de negócio, ou seja não apresenta nenhuma solução semelhante aos EJBs. No entanto, é boa prática separar a lógica de negócio dos controladores da interface, por isso, no sentido de garantir uma boa arquitectura é recomendado criar classes específicas para lógica de negócio, que não dependam de nenhuma API da framework.

Tal como o Struts ou outras frameworks, o tapestry funciona através de um *Front Controller*, um servlet central que faz a gestão da aplicação. A classe *é org.apache.tapestry.ApplicationServlet* e remete os pedidos recebidos para os controladores Java respectivos.

Para construir uma página típica em Tapestry são necessários três elementos:

- Classe Java subclasss de `org.apache.tapestry.html.BasePage`: Ex. *Home.java*
- Template HTML com componentes Tapestry: Ex. *Home.html*
- Ficheiro de configuração da página: Ex. *Home.page*

A classe Java define atributos e funções a usar no *template* HTML. Este acede ao objecto criado a partir da classe via *tags* específicas. O ficheiro de configuração efectua o mapeamento dos atributos e funções definidas na classe para expressões que possam ser referidas no *template* HTML. É a partir deste *template* que é gerado o código HTML final a ser enviado ao cliente.

No Tapestry já são aplicados alguns conceitos de *Convention over Configuration*, pois se os três ficheiros acima mencionados tiverem o mesmo nome, não é necessária nenhuma configuração adicional.

De notar que na navegação entre páginas não são referidos URIs directos, mas sim o nome da página. Quando a aplicação corre, o URI correcto é gerado automaticamente. Este método é tornado possível pois a framework procura as classes que estendem a classe *BasePage* e regista-as com os nomes respectivos. O Tapestry também suporta injecção de dependências, uma vez que está integrado com o HiveMind, uma biblioteca de DI semelhante ao Spring.

Uma das dificuldades desta framework é o requisito que algumas páginas sejam abstractas, pois a framework é que vai criar as implementações das classes, com o código gerado automaticamente. São estas implementações, as classes concretas, que são usadas. Este é um aspecto do Tapestry 4 nada tradicional na programação de uma aplicação Web, que dificulta a aprendizagem, e leva a que alterações ao código forcem o reinício do servidor.

Lógica
Não força nenhuma implementação específica, sendo esta framework orientada à apresentação.

Apresentação
A interface é deinida por um ficheiro de *layout* é efectuado através de um *template* HTML com componentes Tapestry que ligam à classe Java. O controle da interface é implementado pelas classes Java correspondentes a cada página. Suporta componentes AJAX.

Acesso a dados
Não força nenhuma implementação específica, pode ser usado JPA, JDO, Hibernate, JDBC ou outras soluções.

➢ Wicket

A framework Wicket 1.0 [196] foi lançada a 21 de Junho de 2005. Tal como o Tapestry, o Wicket é dedicado mais à camada de apresentação, não apresentando alternativas aos EJB's, por exemplo.

A configuração da aplicação consiste na definição de um *Front Controller* implementado pela classe *org.apache.wicket.protocol.http.WicketFilter*. É necessário indicar que aplicação usar por default, criando uma subclasse de *org.apache.wicket.protocol.http.WebApplication,* onde é definida qual a página inicial.

Esta framework requer que cada interface seja criada através de dois ficheiros : um ficheiro HTML que define o *layout* e um ficheiro que define uma classe Java associada, não requerendo nenhuma configuração adicional em XML. Ao contrário do Tapestry, que usa a classe Java apenas para conter os dados dinâmicos referenciados pelo ficheiro HTML, no Wicket, a classe Java define os componentes da página HTML, bem como os seus valores. Ao contrário do Tapestry, não é preciso definir classes abstractas, mas é necessário associar manualmente os componentes Java com os elementos wicket colocados no HTML através dos seus ids, o que se torna trabalhoso e conduz a erros.

A associação entre página HTML e classe Java é feita por *Convention over Configuration*, nomeadamente possuírem o mesmo e estarem colocados no mesmo directório.

É de notar que o Wicket usa componentes directamente no código Java, sendo guardada uma árvore de componentes para cada página na sessão de utilizador, o que consome alguma memória.

Tal como o SWING[175], os componentes têm associados o seu modelo, ou seja os seus valores. Embora por definição esse modelo fique também guardado na sessão, o Wicket dispõe de interfaces que podem ser usadas para definir modelos *detached*, ou seja, que são usados apenas para gerar o HTML da interface, mas não são guardados na sessão, oferecendo ao programador um maior controle sobre que dados efectivamente persistir na sessão. Por outro lado, a escolha dos componentes em Java permite criar novos componentes a partir dos já existentes, e reutilizá-los na aplicação.

O Wicket contém de base vários componentes, desde formulários, painéis, a listas de itens, possibilitando a criação rápida de interfaces comuns. Suporta também AJAX através do uso de componentes que definem comportamentos em JavaScript.

Lógica
Não força nenhuma implementação específica, sendo esta framework orientada à apresentação.

Apresentação
Cada página é definida num ficheiro HTML que configura o *layout* e numa classe Java que define os componentes. O controle da apresentação depende das classes Java associadas a cada página. O AJAX também é suportado.

Acesso a dados
Não força nenhuma implementação específica. Pode ser usado o JPA, JDO ou Hibernate, JDBC ou outra solução.

➢ Stripes

O Stripes [171] teve a sua primeira versão lançada em Setembro de 2005 estando actualmente disponível a versão 1.5. A sua criação deve-se à frustração dos autores no uso do Struts e Spring, especialmente em relação às configurações requeridas. Como tal, o Stripes faz uso extensivo de convenções e anotações para eliminar a configuração via ficheiros XML.

O Stripes usa para a interface JSP com *tags* próprias e *ActionBeans* como controladores, que são classes que implementam a interface *net.sourceforge.stripes.action.ActionBean.* Os *actions beans* não são associados a um JSP por configuração via XML, mas sim por convenção. A partir do nome completo da classe, o Stripes associa um URL. É possível alterar este comportamento usando anotações.

O *Front Controller* nesta framework é o filtro *net.sourceforge.stripes.controller.StripesFilter* que é configurado com o directório raiz onde procurar classes que implementam a interface *ActionBean* que define um controlador. O Stripes associa então a cada *ActionBean* encontrando o seu respectivo URL.

São usados JSP como tecnologia de interface com recurso a tags próprias para aceder aos atributos e métodos dos *ActionBeans*.

O Stripes possui integração com o Spring, oferecendo aos *ActionBeans* a escolha de utilizar injecção de dependências.

Lógica
Não força nenhuma implementação específica, sendo esta framework orientada à apresentação.

Apresentação
Usa JSP com *tags* Stripes para associar a atributos e métodos dos *ActionBeans*. Os *ActionBeans* constituem os controladores na arquitectura desta framework. Suporta AJAX através de bibliotecas como o Prototype.

Acesso a dados
Não força nenhuma implementação específica de persistência. Pode ser usado JPA, Hibernate, JDO, JDBC, ou outras soluções.

➢ RIFE

A primeira versão do RIFE[146] foi lançada no final de 2001. Actualmente a versão estável é a 1.6.1, disponível desde Julho de 2007. Esta framework foi desenvolvida tendo em mente o máximo de flexibilidade e reutilização de componentes quanto possível.

Uma aplicação RIFE é definido centralmente num ficheiro XML denominado de repositório. Neste repositório são definidos os participantes da aplicação, ou seja, recursos como fontes de dados, ficheiros de configuração e outros. Para além do XML a configuração do site pode ser feita em Java ou Groovy.

Um site é considerando como um participante, pelo que tem de ser definido no repositório, indicando a página inicial através de um mapeamento a uma classe Java.

No RIFE, uma página é considerada um elemento e, tal como no Tapestry e Wicket, é constituída por uma classe Java e por um ficheiro HTML. A substituir a classe Java, a framework suporta linguagens de *scripting* como o Groovy, o que oferece capacidade de alteração da implementação em *runtime*.

A classe Java é subclasse de *com.uwyn.rife.engine.Element* e implementa o método *processElement()*. O ficheiro HTML serve de *template*, com *tags* próprias do RIFE, identificadas por <*r*>. Estes são elementos dinâmicos que serão geridos pela framework, que então trata de gerar o HTML final que é enviado ao *browser* do cliente.

Um dos problemas desta framework é que os atributos disponíveis a serem referidos nas tags RIFE do Template HTML são adicionadas manualmente no código da classe Java ou *script* Groovy. Ou seja, os atributos da classe Java não são mapeados automaticamente para os campos disponíveis no *template*, pelo que é necessário esforço adicional.

A lógica da navegação reside em que o site é constituído por páginas - elementos - que são ligados entre si. Cada elemento é um componente, e como tal, pode ser reutilizado na mesma aplicação ou noutras. A navegação entre páginas é definida na configuração do site do repositório, por referência aos elementos, e construindo ligações de fluxo e dados entre eles. São usadas as tags <*flowlink*> e <*datalink*> para indicar a ligação de navegação e de dados, respectivamente.

As ligações de fluxo dependem dos pontos de saída de cada elemento. Este pontos de saída, identificados por um nome, permitem efectuar ligações a outros elementos. São estas ligações que permitem a navegação entre elementos.

As ligações de dados, permitem a troca de dados entre elementos, e podem ser ligações de entrada ou saída de dados. Adicionalmente, os dados podem ser renomeados de um componente para outro, pelo que uma variável X num elemento pode ser transferida como variável Y noutro elemento. Esta capacidade melhora a flexibilidade da framework e a capacidade de reutilização dos elementos.

Como estas ligações dependem fortemente de cada elemento, no RIFE é necessário associar a cada elemento um ficheiro XML onde definimos a configuração do mesmo. Portanto, na verdade existe, para cada elemento do site, três ficheiros : a classe Java ou *script*, o *template* HTML e um ficheiro XML que define as ligações. Assim é possível definir cada elemento e as suas capacidades de interligação como um componente funcional da aplicação.

O RIFE permite portanto uma grande flexibilidade na construção da aplicação, suportando alterações em tempo real, reutilização de componentes e fácil integração de aplicações.

No entanto, possui vários pontos fracos. Entre eles, uma sintaxe das tags RIFE usada nos *templates* HTML que não é nada evidente, a documentação é muito incompleta em relação às suas capacidades e funcionalidades. Para dificultar ainda mais a aprendizagem, estas tags podem ser utilizadas de três formas diferentes, e a documentação intercala essas três formas sem grande distinção.

O RIFE suporta a geração automática de aplicações simples para CRUD a partir dos modelos do domínio do problema, uma solução própria denominada RIFE/CRUD que trata da persistência e também da criação completa do site, incluindo páginas e navegação.

Mas ao contrário do RoR ou do Grails, onde são gerados os artefactos, ou seja, toda a estrutura de directórios e ficheiros com código necessários à aplicação, no Rife nada é gerado, estando toda a configuração residente em memória.

Esta metodologia reduz a capacidade de alteração da solução gerada bem como da aprendizagem da framework, pois esta podia se basear no código criado, que serviria de exemplo de boas práticas de uso do Rife.

Lógica

Não força nenhuma implementação específica, sendo esta framework orientada à apresentação.

Apresentação

Para cada interface, existe uma página HTML que define o *layout* e uma classe JAVA ou *script* Groovy que define o comportamento dinâmico. O controle é feito através da implementação dos elementos Java, mas a configuração dos elementos é feita em ficheiros específicos e dedicados a cada elemento, enquanto que a navegação do site é determinado em ficheiro XML que pode ser alterado em tempo real. Suporta AJAX através da biblioteca *Direct Web Remoting* (DWR)[46].

Acesso a dados

A utilização do RIFE/CRUD é opcional, sendo que a framework não força nenhum modelo de persistência em particular, podendo ser utilizado com JPA, Hibernate, JDO, JDBC ou outros.

Anexo II - REST vs Big Web Services

Neste anexo é feita uma comparação entre *Representational State Transfer* (REST) e *Big Web Services* (BWS) como formas de implementação de Web Services. Foram escolhidos vários atributos relevantes por forma a melhor caracterizar cada uma das abordagens. Com base nestes dados pode então ser feita uma escolha informada sobre que solução utilizar.

Rede

O uso de BWS tenta ser abstraído da rede com o objectivo de integração transparente como uma chamada a uma API. Em vez disso, o REST utiliza directamente a rede, identificando recursos por URIs, fazendo uso de *hyperlinks* para relacionar recursos, bem como opcionalmente obter dados mais detalhados.

Normas

Existem muitas normas WS, a fim de adicionar capacidades, tais como suporte a mensagens, segurança, gestão, processos de negócios, entre outras. Mesmo que os programadores possuam a escolha de que normas utilizar numa determinada situação, existe também uma maior complexidade envolvida.

Contrariamente, REST necessita ainda de normas, embora aplique standards como HTTP, XML e URIs. Por exemplo, não existem normas para o mapeamento dos recursos para URIs, para documentação dos serviços, definição de suporte à entrega fiável de mensagens ou notificação assíncrona de eventos. Portanto, os fornecedores de serviços podem implementar soluções únicas, deixando aos programadores o esforço de compreensão das diferentes abordagens.

Registo de Serviços

Em termos de registo de serviços, os BWS podem recorrer não só ao UDDI como ao ebXML. Em REST, não existe nenhum standard para centralização de registo de serviços, cabendo a cada entidade efectuar a publicação através de documentação facultativa e utilizando *hyperlinks* a fim de expôr recursos adicionais.

Implementação

Um cliente REST necessita apenas de efectuar pedidos simples em HTTP, usando como conteúdo a representação do recurso. Em contraste, BWS necessita do SOAP e do WSDL, tornando a criação de *stubs* uma obrigação a fim de simplificar os detalhes de implementação. Portanto, é mais simples desenvolver componentes com REST do que com BWS.

Protocolos

SOAP foi desenvolvido para ser flexível, podendo ser considerado uma framework de protocolos. SOAP define uma estrutura de documento e regras de codificação, tornando-o mais pesado do que XML simples sobre HTTP, o que aumenta o tamanho da mensagem, tempo de processamento, bem como o esforço requerido para a sua utilização.

Segurança
Com BWS, é possível aplicar algumas normas WS para os aspectos de segurança, tais como *WS-Security*[205], *WS-SecureConversation*[155] e *WS-SECURITYPOLICY*[206], entre outras.

Tanto REST como BWS podem usar HTTP sobre Secure Sockets Layer para proporcionar comunicações seguras.

SOAP sobre HTTP é normalmente utilizado devido às firewalls, mas neste caso o tipo de pedido é indicado no XML, não sendo óbvio para as firewalls sem inspecção profunda de pacotes e aplicações específicas a esse fim.

O REST é mais fácil de gerir, uma vez que também depende de HTTP, mas o tipo de pedido é evidenciado pelo método HTTP utilizado. Assim, é possível gerir na firewall regras de permissão controlando o acesso a determinados URIs e acções HTTP.

Transacções
O suporte a transacções numa aplicação com acesso a múltiplos serviços remotos provenientes de vários fornecedores de serviços é uma tarefa difícil. Existem standards WS para o suporte de transacções tais como o *WS-Atomic Transaction* (WS-AT)[200], que faz parte do *WS-Coordination* [202].

No REST, existem várias estratégias para a adição de suporte a transacções, mas uma vez mais, há a questão recorrente da falta de normas.

Uma solução possível seria considerar as transacções como recursos, efectuando um POST para criar uma transacção. POSTs adicionais seriam utilizados para adicionar as várias etapas dessa transacção com os dados necessários e finalmente a transacção pode ser submetida alterando um atributo que indicada o seu estado ou cancelada via DELETE.

A necessidade de suporte a transacções numa aplicação REST pode significar uma abordagem demasiado pormenorizada na exposição dos recursos, o que levanta a possibilidade de redesenhar a aplicação de forma a utilizar um único pedido para efectuar a função desejada.

Finalmente, tanto WS como REST beneficiariam com o suporte ao *Business Transaction Protocol* (BTP)[24], uma solução baseada em XML proposta pela *Organization for the Advancement of Structured Information Standards* (OASIS)[125]. O BTP é agnóstico em relação aos protocolos de comunicação, estando definidos mapeamentos ao SOAP.

Escalabilidade
Os BWS são escaláveis como comprovado através da disponibilização de serviços web publicamente disponíveis pela Amazon[8], Yahoo[218] e Google[68], mas a escalabilidade é simultaneamente dispendiosa e complexa de garantir.

REST visa diminuir o esforço necessário para criar aplicações escaláveis, recorrendo a pedidos stateless, desenho em camadas que permite várias estratégias para melhorar o desempenho, bem como a possibilidade de guardar em cache resultados para posterior utilização.

Ainda assim, Fielding, na sua tese de doutoramento, faz a observação de que REST é um estilo arquitectural adequado para distribuição de hipermédia em massa e, portanto, pode não ser a melhor solução para aplicações de menor dimensão.

Design
Os BWS são centrados em acções, concebidos para serem utilizados como chamadas API, ao contrário de REST com a sua abordagem orientada a recursos. Logo, utilizar BWS é um problema de design de APIs enquanto que o REST consiste num problema de manipulação de recursos.

Pode ser argumentado que a criação de uma API para execução remota, é mais difícil de conceber e implementar, do que um sistema com um grupo de acções reduzido aplicado a um conjunto hierárquico de recursos que podem ser derivados de modelos do domínio do problema. [referência].

Suporte
Existe uma variedade de BWS disponíveis para uso público por parte de diversos fornecedores tais como Google, Yahoo e Amazon, entre muitas outras empresas. No entanto, o suporte a REST tem crescido com o lançamento de serviços adicionais da Yahoo e Amazon, assim como muitas outras pequenas empresas.

Testes
Com REST, é possível testar serviços com um simples web browser. Para uma solução mais completa, é recomendado usar uma ferramenta como o REST Client da WizTool[141], actualmente na versão 2.1. É capaz de realizar pedidos HTTP, especificando cabeçalhos, conteúdo, tipo de conteúdo via mime, dados de autenticação e possui a capacidade de guardar e carregar pedidos efectuados.

Para o teste de BWS, existe o SOAP UI[165], uma ferramenta muito útil no desenvolvimento de aplicações com BWS. Este programa permite invocar BWS com suporte a alguns perfis WS-I tais como WS-Security. Permite também inspeccionar e validar WSDL e XML, bem como desenvolver casos de teste e simular BWS.

Os BWS e REST podem ser testado de forma mais básica com recurso ao Poster[138], um plugin para o Firefox, que permite efectua pedidos como POST e PUT, para além dos pedidos GET que o browser efectua.

Documentação
Nos BWS, é necessário criar um ficheiro WDSL para declarar os métodos e tipos de serviço para que o cliente possa conectar-se e executar uma operação. Esta descrição é utilizada principalmente

para a criação de *stubs* em vez de realmente documentar a utilização do serviço num contexto.

REST não possui nenhum *standard* para a descrição ou documentação do serviço, embora na prática, basta um simples documento HTML explicando que URIs são utilizados, as suas relações com os recursos, que métodos HTTP e representações são suportados.

Ferramentas

Em Java o suporte a BWS é efectuado recorrendo a um conjunto de APIs ou bibliotecas *open source*. Das APIs oficiais da SUN destacam-se as seguintes:

- ***Java Architecture for XML Binding* (JAXB)** [97] - possibilita o mapeamento de classes java para representações em XML, bem como o processo de *marshal* e *unmarshal*. Não é necessário nenhum esforço de implementação, apenas de configuração do mapeamento, o que é conseguido através de anotações.
- ***SOAP with Attachments API for Java* (SAAJ)** [151] - permite construir e ler mensagens SOAP, bem como enviar mensagens através da rede.
- ***Java API for XML Web Services* (JAX-WS)** [96] - permite criar BWS em Java com recurso a anotações para simplificar o esforço.
- ***Java API for XML Registries* (JAXR)** [98] - permite às aplicações Java aceder e interagir com vários tipo de registos como por exemplo, o UDDI ou ebXML.

Em adição à definição destas APIs a SUN criou uma iniciativa denominada *Web Services Interoperability Technology* (WSIT)[209] que consiste num projecto open source com o objectivo de facilitar a criação e uso de novas tecnologias de Web Services utilizando as APIs Java referidas bem como a implementação de especificações WS-I tais como WS-Trust[207], WS-SecureConversation[155], WS-ReliableMessaging[204] entre outras.

Estas APIs e a iniciativa WSIT possuem implementações disponíveis denominadas de *Reference Implementation* (RI)[144] pois são a referência utilizada pela SUN como exemplo de implementação das especificações. Estas implementações são open source e fazem parte do projecto *Metro*[113] da SUN, uma pilha de BWS ou *WS-stack*.

Uma B*WS-stack* integra um conjunto de normas BWS e capacidades, tornando mais fácil a implementação e configuração de serviços, eliminando muito do esforço necessário à criação manual de ficheiros arquivos WSDL e stubs associados.

O *Metro* é uma stack da SUN, utilizada no projecto *GlassFish*[67], o servidor referência de aplicações J5EE. Existem muitas outras ferramentas BWS, nomeadamente, BWS-stacks tais como Metro , Apache Axis [14] e Xfire [212], ou bibliotecas open source como o Enunciate[53] e Apache CXF[15].

Existem, no entanto, pontos negativos à utilização destas ferramentas:

- Cada pilha BWS possui características diferentes tais como o conjunto de normas WS suportadas ou protocolos de rede. Isto pode levar a problemas quando diferentes pilhas BWS necessitam de comunicar entre si.
- A existência de várias versões da mesma pilha com algumas características diferentes também podem levar a implementações distintas com problemas de comunicação.
- Embora as ferramentas abstram vários detalhes dos BWS e reduzam o esforço manual, possuem um conjunto de capacidades limitadas, para além das quais o programador terá de depender dos seus conhecimentos sobre os padrões BWS e a pilha específica a ser utilizada.

A nível de IDE's existem vários plugins disponíveis para o Eclipse e Netbeans que facilitam o desenvolvimento de aplicações com BWS.

Em relação ao REST, baseia-se em padrões amplamente conhecidos e utilizados - HTTP, URI, XML - para os quais existem vários recursos disponíveis em várias linguagens de programação. No entanto, existe uma falta de ferramentas de suporte e documentação bem como aplicações que sirvam de referência.

A SUN definiu a especificação *Java API for RESTful Web Services* (JAX-RS)[95] de forma a providenciar suporte a serviços REST na plataforma Java com recurso a anotações em *Plain Old Java Objects*. A implementação referência consiste no projecto open source denominado *Jersey*[104]. A Microsoft adicionou também suporte a REST na versão 3.5. da plataforma .NET.

Outro esforço consiste na *Web Application Description Language* (WADL) [192], criada com o objectivo de providenciar uma descrição de serviços baseados em HTTP que possa ser processada por máquina. É possível gerar código para suportar aplicações REST a partir de ficheiros WADL para plataformas específicas como o Java.

Existem bibliotecas open source para o suporte a aplicações REST, destacam-se o projecto RESTlet[142] e Enunciate.

Não obstante estes esforços, ainda existem várias questões em aberto no desenvolvimento de aplicações REST:

- Embora haja um conjunto crescente de bibliotecas e ferramentas em várias linguagens de programação, não existe nenhuma especificação standard globalmente aceite pela comunidade para definir a camada de serviços de uma aplicação;
- Os princípios REST são independentes da plataforma, mas a ferramentas actuais são bastante específicas. Os programadores têm de aprender um novo conjunto de ferramentas para cada plataforma onde queiram implementar os serviços;

- Desenvolver uma aplicação requer testes exaustivos que levam a alterações. Implementar uma camada de serviços não é diferente mas as ferramentas actuais não fornecem um mecanismo de alteração dinâmico, pelo que pequenas alterações requerem que a aplicação seja reiniciada. No caso da plataforma Java, pequenas mudanças de código podem ser aplicadas pelo mecanismo de 'Hot-Swap'[79], mas qualquer alteração à camada de serviços requer um reinício. Esta situação dificulta o desenvolvimento por ser um caso de utilização muito comum, e;
- Não existe descrição de uma interface ou serviço de administração comum, independente de plataforma, capaz de aplicar mudanças dinamicamente à camada de serviços.

Frameworks

O suporte a Web Services nas diversas frameworks referidas no estado da arte é um aspecto relevante para este trabalho. Embora as frameworks tenham sido desenvolvidas para simplificar a construção de aplicações centradas em HTML por via de diferentes estratégias, a importância crescente dos Web Services tem levado à adição de capacidades de implementação destes serviços nas frameworks.

Existem três tipos de abordagem ao suporte a Web Services nestas frameworks. Estas abordagens serão descritas de seguida, bem como as frameworks que as adoptaram:

- Utilização directa de soluções sem uma integração de base na framework, tais como stacks BWS - Metro, Axis 2, Xfire -, servidores com stacks já incluidas como o Glassfish, bibliotecas open source como REStlet ou Enunciate, ou ainda bibliotecas específicas dos fornecedores de serviços. Sem o esforço de integração adicional há uma menor produtividade em contraste com as facilidades disponíveis ao desenvolvimento da camada interface ou de controle: Struts 1, Tapestry, Wicket e Stripes.
- Integração das ferramentas mencionadas acima, fazendo uso das suas capacidades no processo de desenvolvimento típico de cada framework. Ainda assim, as capacidades ficam aquém do que está disponível para a interface, nomeadamente alterações on the fly dos serviços web.
    - Java5EE - uso da stack Metro com o conjunto de APIs para BWS, e suporte a REST via JAX-RS e sua implementação pelo Jersey;
    - RIFE - integração de Xfire, e;
    - Grails - suporte a REST de base, integração do Axis , Metro ou XFire como plugins opcionais.
- Desenvolvimento de soluções próprias, ou através de APIs específicas e simples ou pela implementação de stacks completas.
    - Struts 2 - plugins experimentais Codebehind[31] e REST;
    - SEAM - recurso à stack BWS *JBoss Web Services*, configurável via anotações;

- Spring MVC - recurso à stack *Spring Web Services* (Spring-WS)[169], configurável via XML e anotações, e;
- RoR - suporte de base a SOAP e REST através da classes ActionWebService e futuramente integração de CRUD via Web Services com recurso à classe ActiveResource.

Esta exposição permite verificar que os Web Services e o seu suporte ainda são vistos como um aspecto complementar ao desenvolvimento de aplicações web, não usufruindo sequer de nenhum esforço de integração por parte de várias soluções.

É constatado que mesmo nas frameworks que efectuam alguma integração das soluções existentes, quer nas que implementam a sua própria solução, as capacidades de dinamismo na definição e alteração de serviços são reduzidas. O RoR e o Grails surgem como excepções devido à sua natureza baseada em linguagens de scripting.

No geral, a adopção de BWS é superior à adopção de REST como abordagem aos Web Services, pois apenas 4 frameworks - Grails, RoR, Struts 2 e Java5EE - possuem ou integram ferramentas para desenvolvimento de serviços REST.

Anexo III - Regras de aplicação de *advices*

Neste anexo são dados exemplos de utilização de regras por forma a definir a aplicação de advices. As regras definem restrições com base na configuração dos serviços REST. Por exemplo, pode ser pretendido aplicar um advice para analisar a performance de todos os pedidos de um módulo de gestão de clientes, cujo atributo relative-uri é "*crm"*. Este caso pode ser descrito da seguinte forma:

```
<rule>
   <module-uris>
      <string>crm</string>
   </module-uris>
</rule>
```

É preciso ter em consideração os seguintes factores:

- Diferentes aplicações podem possuir módulos com o mesmo relative-uri.Figura 30;
- O advice é aplicado a todo o tipo de pedidos efectuados aos módulos com atributo *relative-uri* de "*crm*", sejam eles GET, POST, PUT ou DELETE, como visto na Figura 30.
- Uma vez que o sistema de serviços suporta a modificação dos *relative-uri* de aplicações e módulos, se o *relative-uri* do módulo de clientes fosse alterado para "*customers*", então a regra já não aplicaria o *advice* a esse módulo. Seria necessário modificar manualmente a regra.

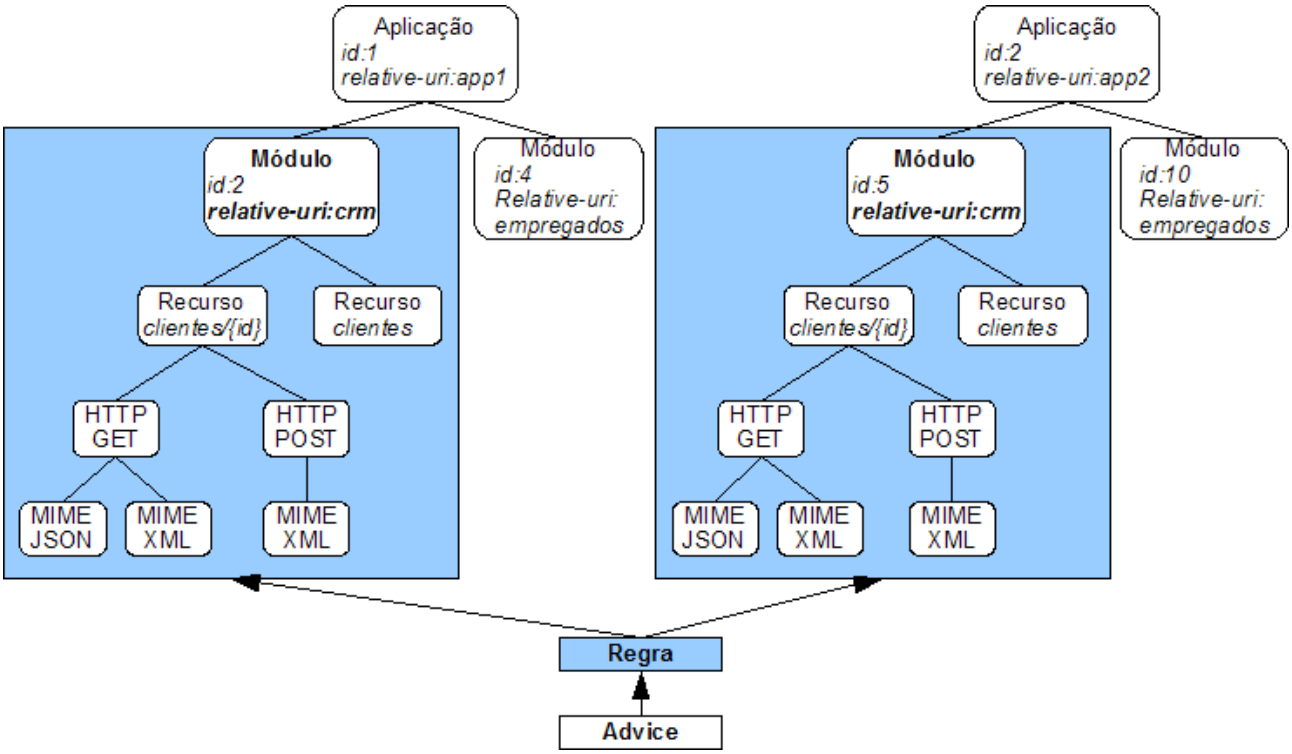

Figura 30: Exemplo de regra de aplicação de Advice pelo relative-uri de módulo

É possível definir regras em torno da identificação do módulo no serviço de administração, ou seja, do atributo *id* utilizado pelo serviço de administração. Assim, independentemente do valor do atributo *relative-uri* do módulo, este *advice* pode sempre ser aplicado.

```
<rule>
   <modules>
      <long>2</long>
   </modules>
</rule>
```

A utilização desta regra já limita a adição do *advice* apenas às acções do módulo 2, como visto na Figura 31.

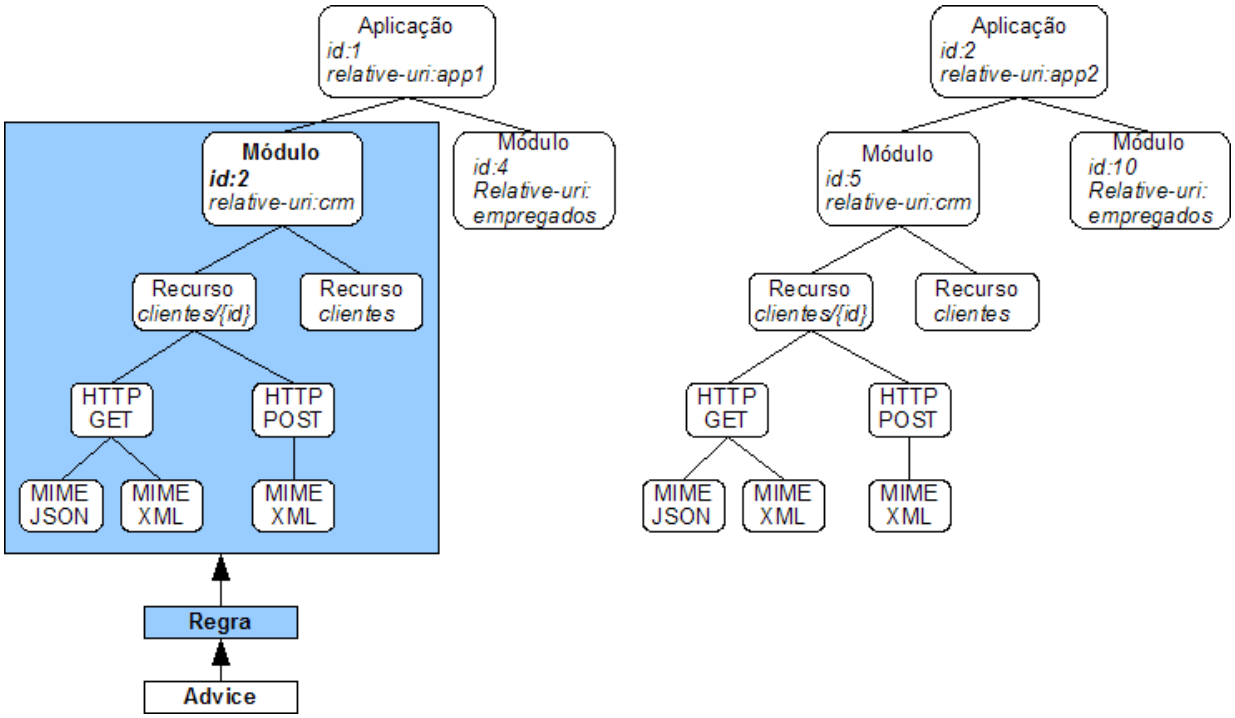

Figura 31: Exemplo de definição de regra a um módulo específico pelo seu id

É pretendido adicionar o *advice* também ao módulo de empregados da mesma aplicação, com o *id* de 4. Então a regra é alterada, como visto no seguinte excerto, abrangendo ambos os módulos, como demonstrado pela Figura 32.

```
<rule>
   <modules>
      <long>2</long>
      <long>4</long>
   </modules>
</rule>
```

Note que podem ser indicados módulos mesmo de aplicações distintas. Este mesmo mecanismo também pode ser estendido em relação aos *ids* de aplicações, recursos, handlers, a todos os tipos de dados da configuração REST. Assim consegue-se uma associação fixa do advice, independentemente da alterações dos atributos da aplicação, módulo ou recurso.

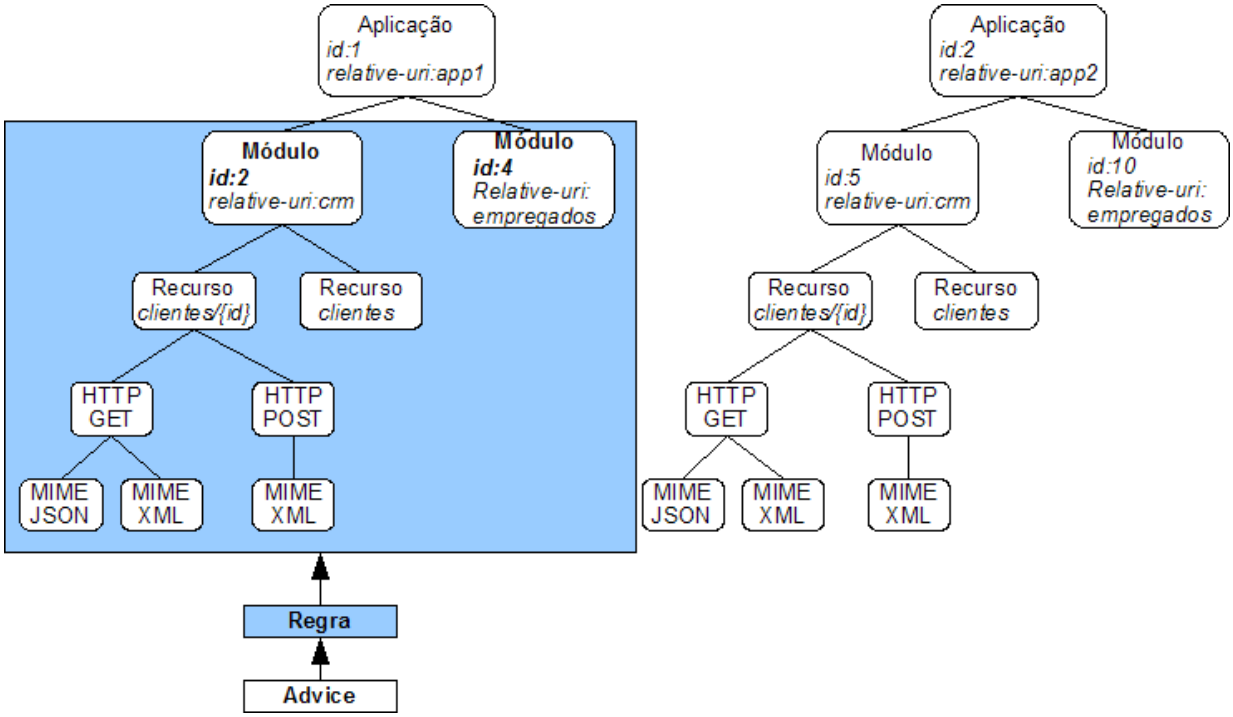

Figura 32: Configuração de regra a dois módulos específicos pelos seus ids

Pretende-se agora restringir a aplicação do *advice* apenas aos pedidos de leitura (GET). Basta adicionar uma nova condição à regra, como visto no excerto que se segue e na Figura 33:

```
<rule>
   <modules>
      <long>2</long>
      <long>4</long>
   </modules>
   <http-method-names>
      <string>GET</string>
   </http-method-names>
</rule>
```

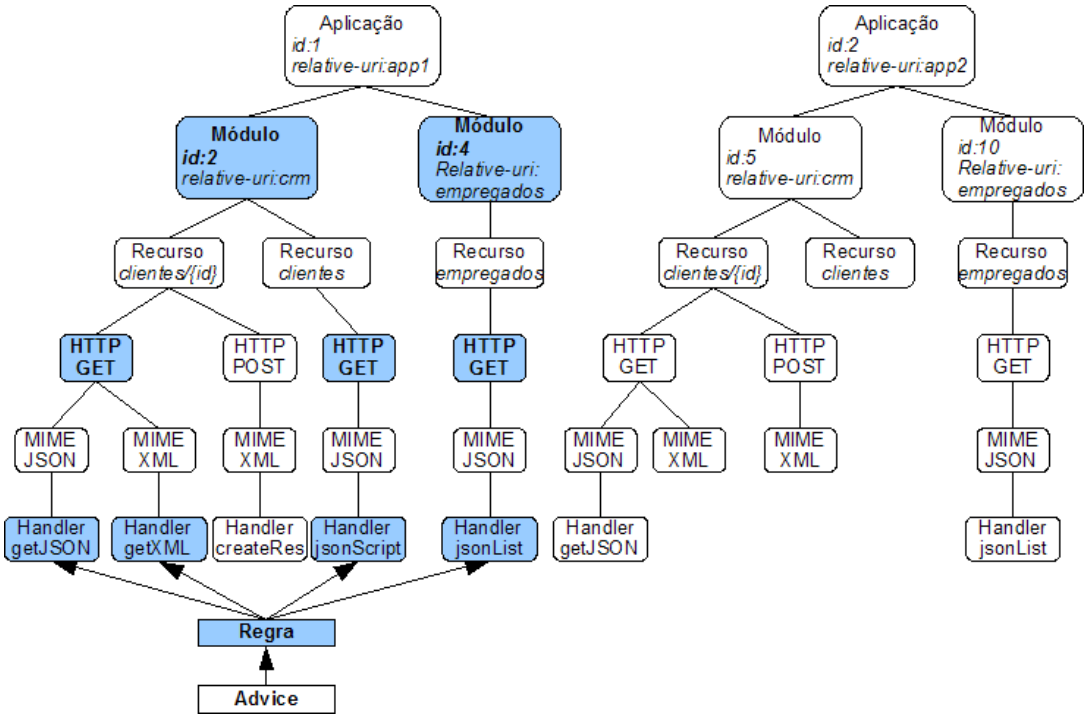

Figura 33: Exemplo de regra com restrições de id de módulos e método HTTP

É possível conceber facilmente vários cenários de aplicação de regras. Por exemplo, pode ser necessário avaliar a performance de todas as acções de leitura de dados quando a representação a retornar deve estar no formato JSON. Então o *advice* teria de ser aplicado a todos os pedidos GET, para qualquer aplicação, módulo, ou recurso do servidor, mas restringido ao tipo mime de *application/json*. Esta regra é descrita como:

```
<rule>
   <http-method-names>
      <string>GET</string>
   </http-method-names>
   <mime-types>
      <string>application/json</string>
   </mime-types>
</rule>
```

O excerto acima aplicaria o *advice* às acções como indicado na Figura 34:

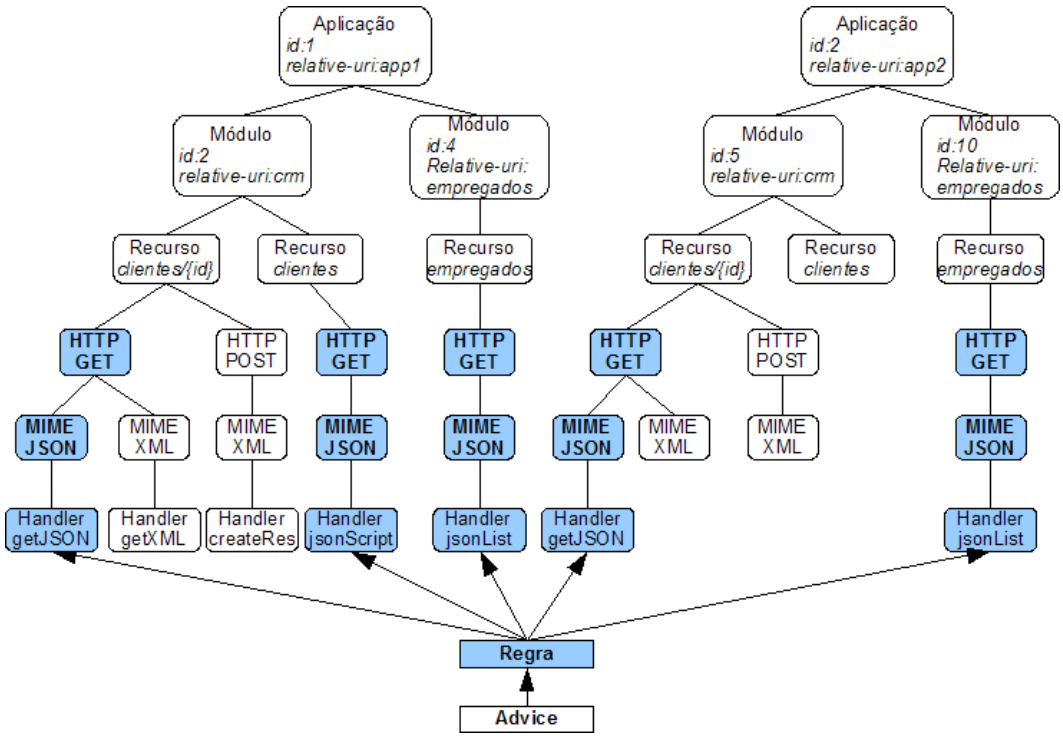

Figura 34: Exemplo de regra por método HTTP e tipo MIME

Anexo IV – Arquitectura

Este anexo descreve a arquitectura dos projectos essenciais para a implementação do servidor : core rest server, generation, aop e jtaction.

Core rest server

O papel deste projecto é definir a base do servidor, com a capacidade de configurar componentes para receber pedidos HTTP correspondentes a um determinado URI. O projecto iniciou-se com a definição de classes correspondentes à especificação de configuração de serviços REST com visto na Figura 35. Essas classes permitem organizar os recursos pelas aplicações e seu módulos, indicado que métodos HTTP são suportados e tipos de representação.

É de referir que todas as classes possuem um atributo *active* do tipo boolean por forma a que as instâncias de cada classe possam indicar se estão a funcionar no servidor ou não. As classes possuem também um campo *id* do tipo *Long* com a função de identificação de cada instância. Este campo é assim usado como "chave primária" para identificação na base de dados e no serviço de administração para gerar um URI.

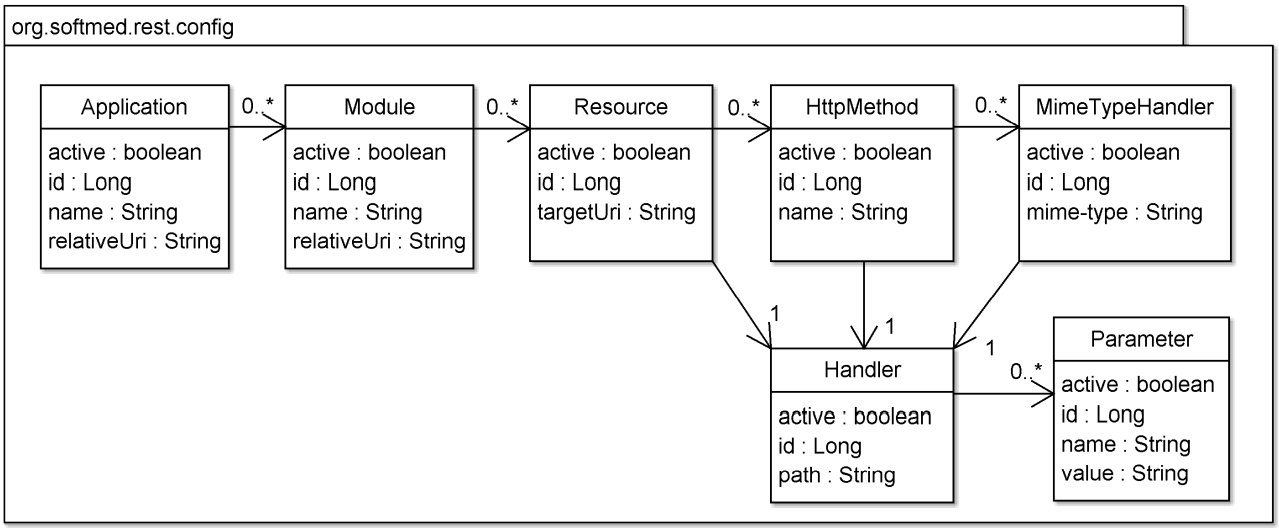

Figura 35: Diagrama de classes da configuração REST

Cada classes possui atributos de acordo com o definido pela especificação. Pode-se ver ainda que cada Resource, HtttpMethod e MimeTypeHandler podem referenciar um Handler. O Handler possui um atributo *path* que identifica o código a processar o pedido por um FQCN correspondente.

Activação de Recursos

As funções principais deste projecto são o carregamento da configuração de serviços e a activação de todos os recursos associados. Para esse fim foi utilizada a à biblioteca RESTlet, que define uma APIs para configuração e activação de componentes que processam pedidos. Esses componentes chama-se RESTlets, uma nomenclatura similar aos Servlets.

Portanto o servidor deve obter de alguma forma a configuração das aplicações e mapear cada recursos a um RESTlet. O processo de mapeamento é relativamente simples, mas um dos problemas é que a biblioteca RESTlet não é dinâmica. Embora a API defina métodos para iniciar e parar componentes, adicionar e remover RESTlets, na realidade esse comportamento não é implementado. A utilização normal da biblioteca consiste em configurar e activar os RESTlets, e qualquer alteração requer terminar a aplicação, modificar o código de configuração, compilar e voltar a executar.

Como o objectivo é tornar possível a configuração e colocação de recursos on-the-fly, foi preciso criar uma solução a este problema. A biblioteca define um componente chamado Router, que permite direccionar os pedidos a componentes de acordo com regras. Os componentes podem ser outros Routers ou até RESTlets. Através de testes, foi possível determinar que é possível colocar e remover um Router de outro Router sem terminar o servidor.

Foi decidido então esconder cada RESTlet por trás de um Router. E cada um destes Routers é colocado num Router central. Assim, quando se pretende retirar um RESTlet, o que efectivamente está a ser feito é retirar o Router específico desse RESTlet do router central.

Resolvido este problema, resta criar o servidor com base nos componentes da biblioteca RESTlet por forma a mapear Recursos a RESTlet, configurar e activá-los. A solução desenvolvida está esquematizada no diagrama de classes da Figura 36:

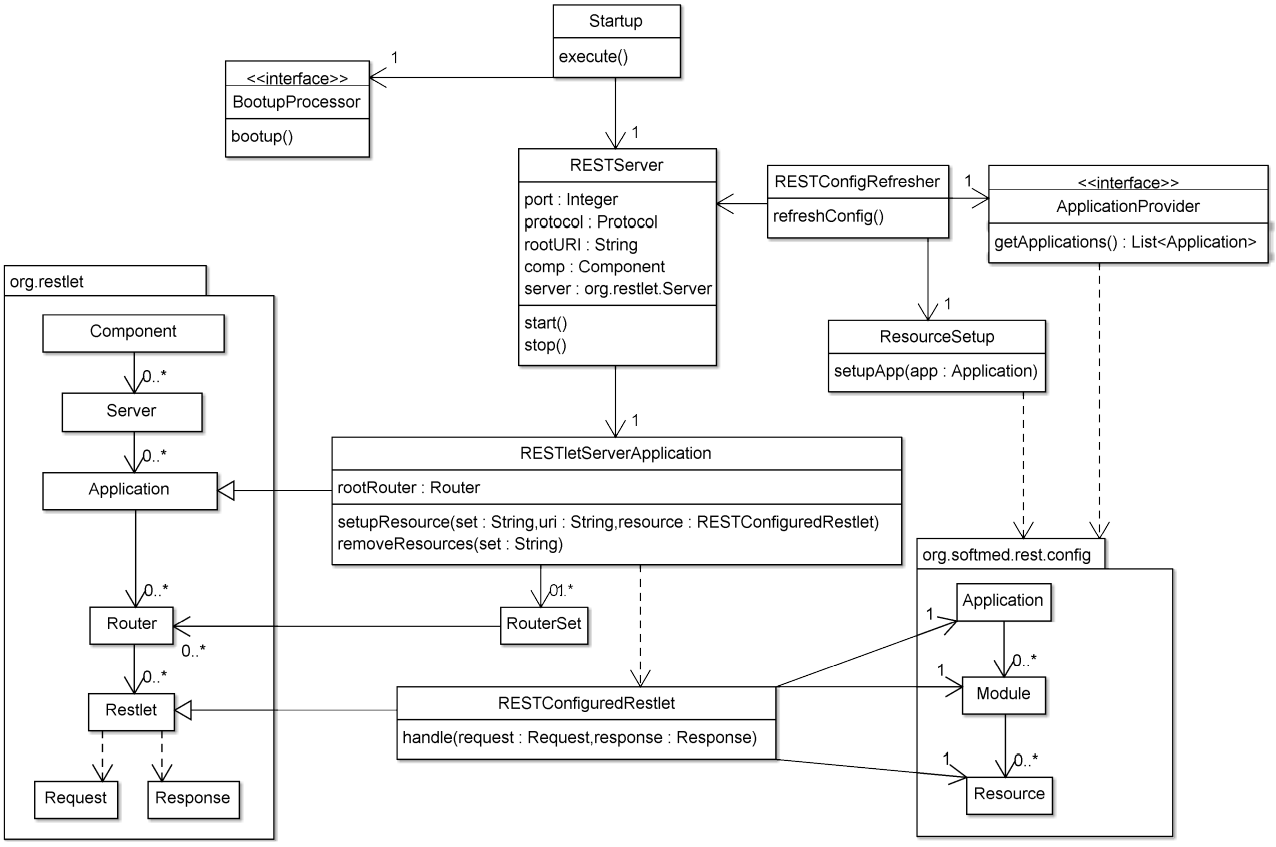

Figura 36: Arquitectura do Core rest server

A classe startup é a que inicia realmente o servidor quando é invocado do método *execute*. Nesse método a primeira acção é invocar o *bootup* do BootupProcessor. Esse método serve para executar qualquer configuração necessária ao início e refrescamento do servidor. Sendo uma interface, abstrai da implementação específica, pelo que podem ser utilizadas várias soluções. A solução por default é executar scripts Groovy existentes num directório "bootup" do servidor, mas essa solução só é implementada no projecto "REST Server".

Após o bootup, é instanciado um RESTServer, que contém atributos que indicam em que porta o servidor deve aceitar pedidos, bem como qual o URI base do servidor. O REST Server detêm um objecto do tipo Component, que é o componente básica da biblioteca RESTlet. Sobre este Component correr a aplicação-servidor, neste caso uma instância de RESTletServerAplication. Esta aplicação possui um campo *rootRouter* do tipo Router.

Este é o router central, que constitui a raíz do servidor, onde se colocam todos os outros routers. Cada recurso da configuração é representado por um RESTConfiguredReslet, uma subclasse de RESTlet, onde se indica os dados básicos da configuração nomeadamente a Application, Module e Resource correspondente a este recurso.

Um dos problemas é que o servidor deve permitir a configuração dinâmica dos serviços, ou seja, a modificação dos recursos expostos, mas é preciso manter os serviços de administração a funcionar. Então decidiu-se por um sistema que agrupe os recursos por conjuntos. Cada conjunto corresponde a uma instância de RouterSet. O RESTletServerAplication detêm um mapeamento de nomes para RouterSet, pelo que é fácil associar cada grupo a um nome. Assim todos os recursos associados à administração ficam no RouterSet de nome "admin" e quaisquer outros recursos associados a outras aplicações ficam no RouterSet de nome "applications", como visto na Figura 37.

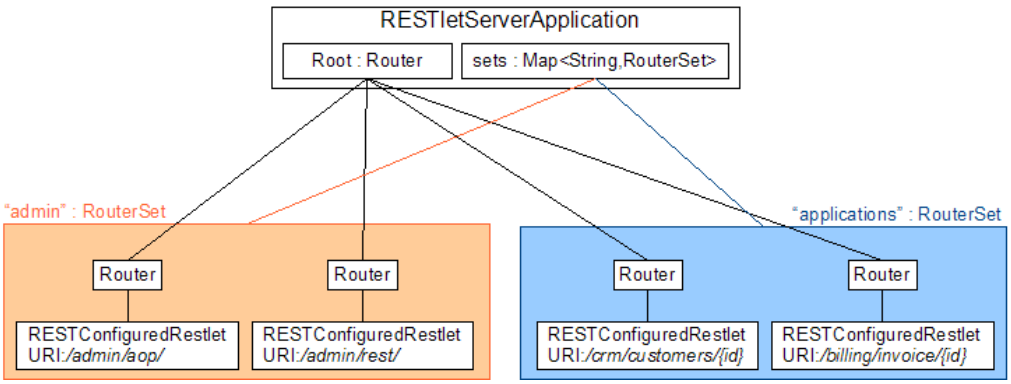

Figura 37: Organização dos recursos por diferentes RouterSet

Assim, quando é necessário efectuar alterações, basta retirar todos os recursos do grupo "applications", e colocar o novo conjunto de recursos. No caso do serviços de administração, todos seus recursos mantêm-se pois pertencem a outro grupo.

Refrescamento

O problema do carregamento inicial da configuração e seu refrescamento é resolvido por uma classe Facade, a RESTConfigRefresher. No método refresh do RESTConfigRefresher todos os recursos instalados são retirados com excepção dos recursos do RouterSet ”admin” através do método removeAllResources do RESTServerApplicaton, acessível pelo RESTServer. (Figura 38)

Depois é pedida a configuração, através do ApplicationProvider e do seu método getApplications(). O ApplicationProvider é uma interface, pelo que a forma de guardar e aceder à configuração pode ser implementada de diversas formas, desde ficheiros XML, a bases de dados relacionais, por exemplo.

Depois, para todas as aplicações retornadas pelo ApplicationProvider, o RESTConfigRefresher mapeia cada recurso a um RESTconfiguredRESTlet, que é e configurado e instalado no servidor pelo método setupResource da RESTApplication. Na instalação é pedido o URI, indicado no atributo target-uri. A partir desse momento aquele RESTconfiguredRESTlet está apto a receber pedidos.

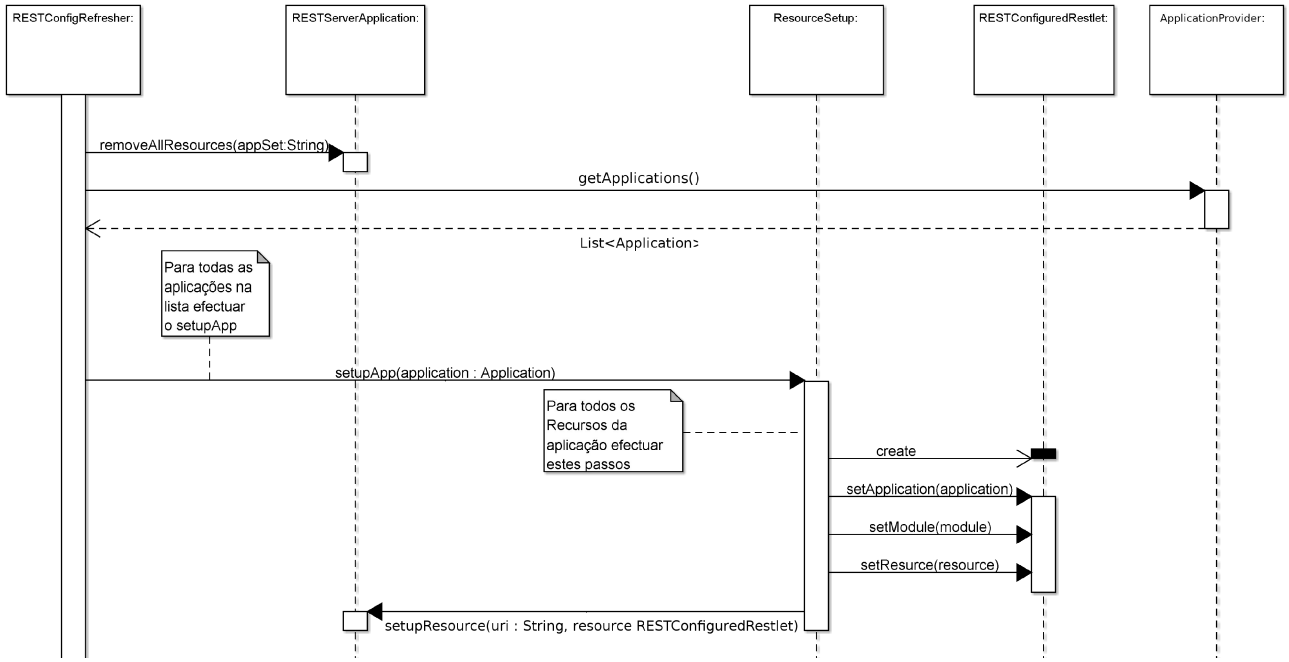

Figura 38: Actualização da configuração REST

O RESTConfigRefresher pode ser assim usado para forçar a instalação ou refrescamento da configuração de uma forma fácil.

Processamento de pedidos

Os objectos RESTConfigureRESTlet ao receberem um pedido precisam de o processar. O processamento depende das características do pedido e da configuração do recursos, nomeadamente, que métodos HTTP e tipos MIME são suportados. O problema de desenvolver a sequência de processamento reside em que o protocolo HTTP, para além do post, put, get e delete disponibiliza outros métodos, e é um protocolo extensível, pelo que mais métodos podem ser

definidos futuramente. Portanto a solução deve reflectir essa realidade.

Foi escolhido implementar a classe ResourceHandler que é instanciada sempre que um pedido é recebido(Figura 39). Essa classe acede a uma implementação da interface HttpMethodProcessor para processar os pedidos. A interface permite abstrair a implementação específica a utilizar. *O core rest server* já oferece uma solução predefinida, o DefaultHttpMethodProcessor.

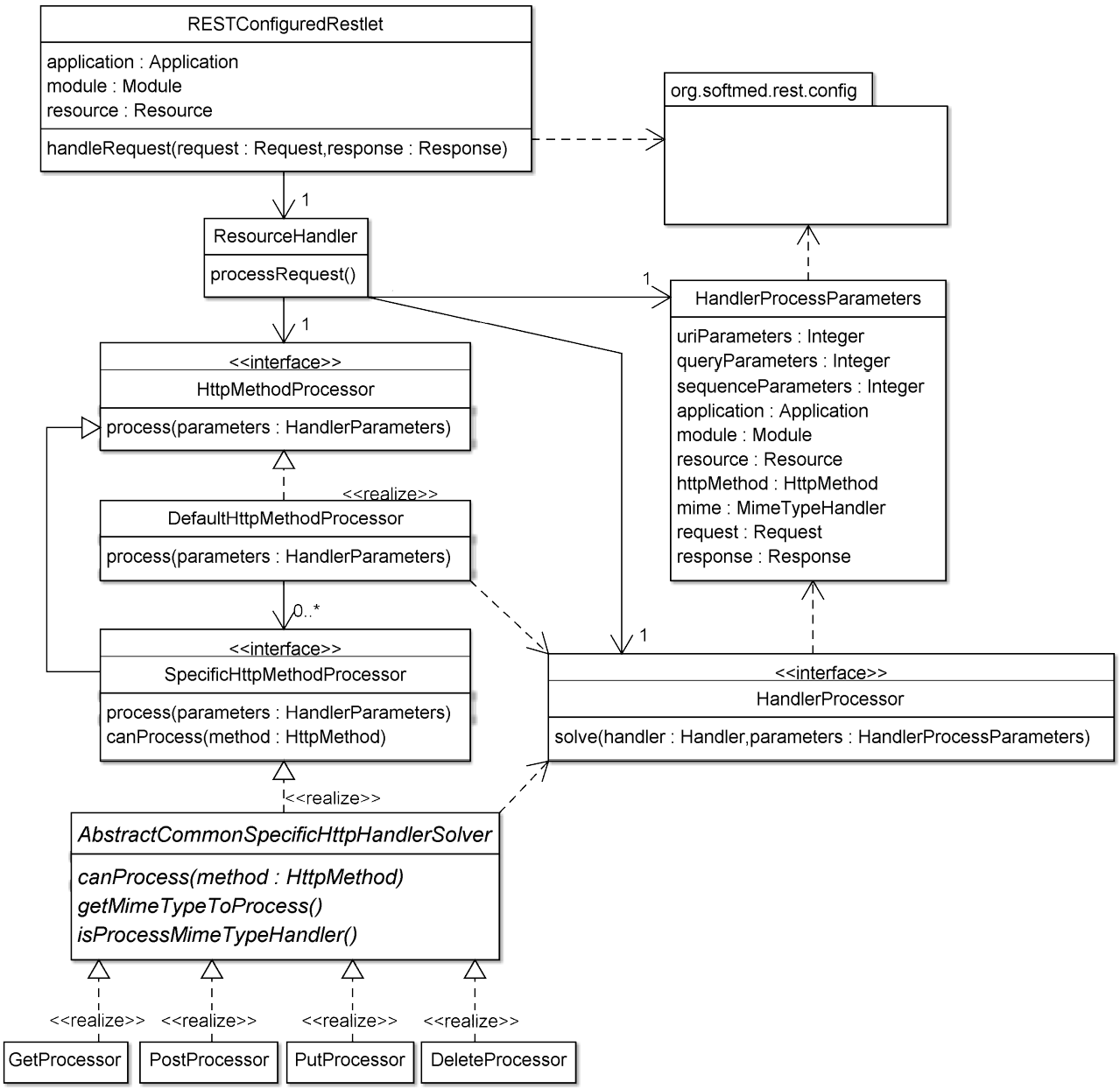

Figura 39: Arquitectura de processamento de pedidos HTML

Esta classe por sua vez, possui uma lista de objectos do tipo SpecificHttpMethodProcessor, que processam apenas um tipo específico de pedido. A lista de processadores específicos é injectada no DefaultHttpMethodProcessor através do GUICE na configuração inicial do servidor. Este projecto

já inclui 4 implementações da interface SpecificHttpMethodProcessor, uma para um dos métodos GET, POST, PUT e DELETE. Respectivamente foram definidas as classes GetProcessor,

PostProcessor, PutProcessor e DeleteProcessor. Como estas 4 classes detêm código em comum, nomeadamente a execução de handlers, é então utilizada uma classe intermédia abstracta: AbstractCommonSpecificHttpHandler.

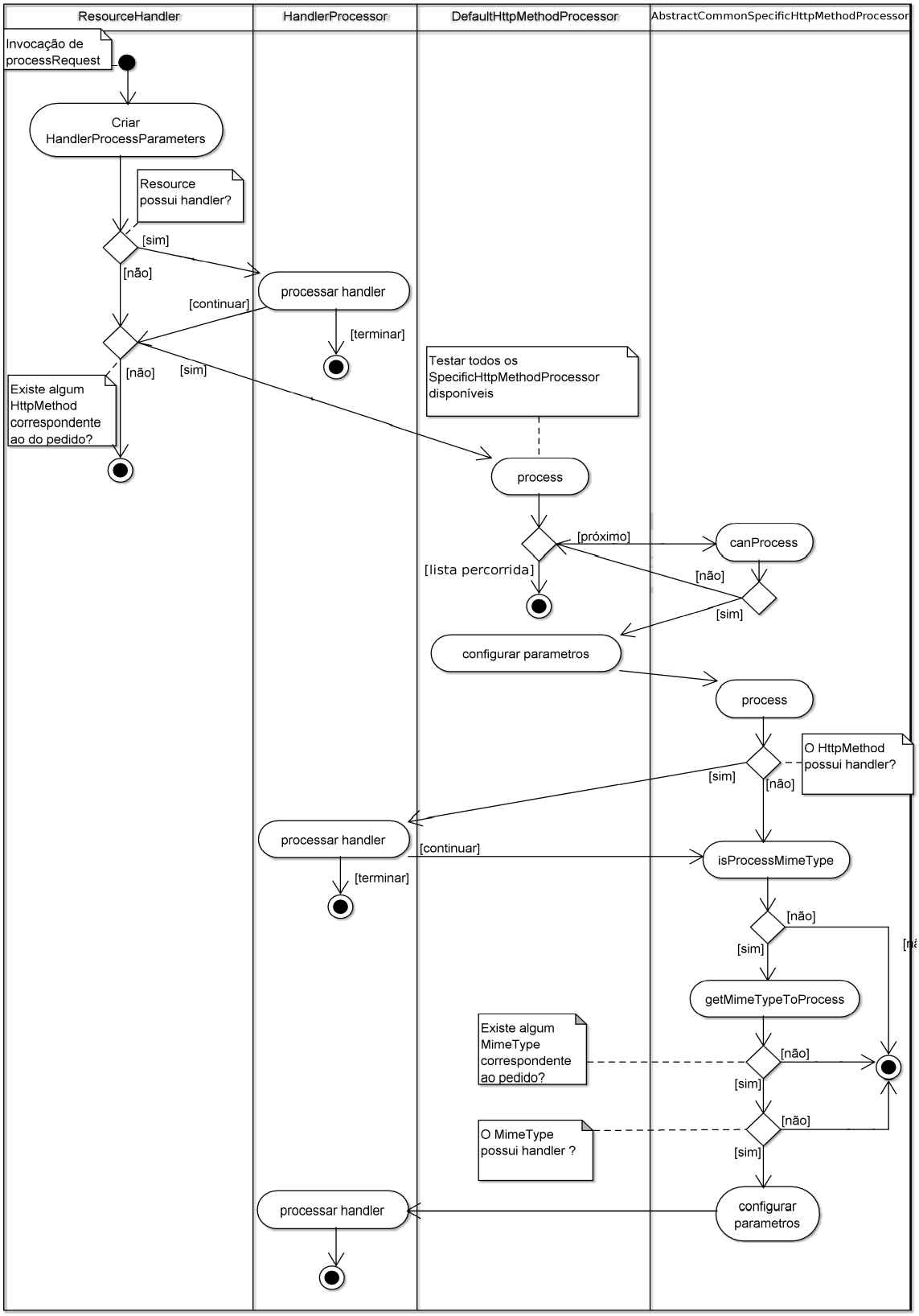

Figura 40: Processamento de pedido HTTP

É de referir também que o ResourceHandler cria um objecto do tipo HandlerParameters, que é preenchido com os dados relevantes ao pedido e ao recursos, sendo passado pelos vários componentes do sistema de processamento. Outro objecto também utilizado pelos vários componentes é o HandlerProcessor, uma interface que define um método para executar handlers, passando os parâmetros.

A sequência de processamento de um pedido está descrita no diagrama da Figura 40 demonstrado as funções de cada um destes componentes.

A interface HandlerProcessor abstrai a forma como os handlers serão tratados, nomeadamente como localizar o código a partir do FQCN. A implementação do HandlerProcessor é injectada no ResourceHandler via DI, sendo passado para o DefaultHttpMethodProcessor e deste para os SpecificHttpandlerProcessor. Assim, sempre que haja algum handler a ser executado, basta passar ao HandlerProcessor.

Em relação ao handler processor, como visto na especificação, é necessário permitir o suporte a vários abordagens de implementação dos handlers, tais como scripts Groovy, classesJava ou jtActions. Por forma a gerir essa solução foi criada a arquitectura representada na Figura 41:

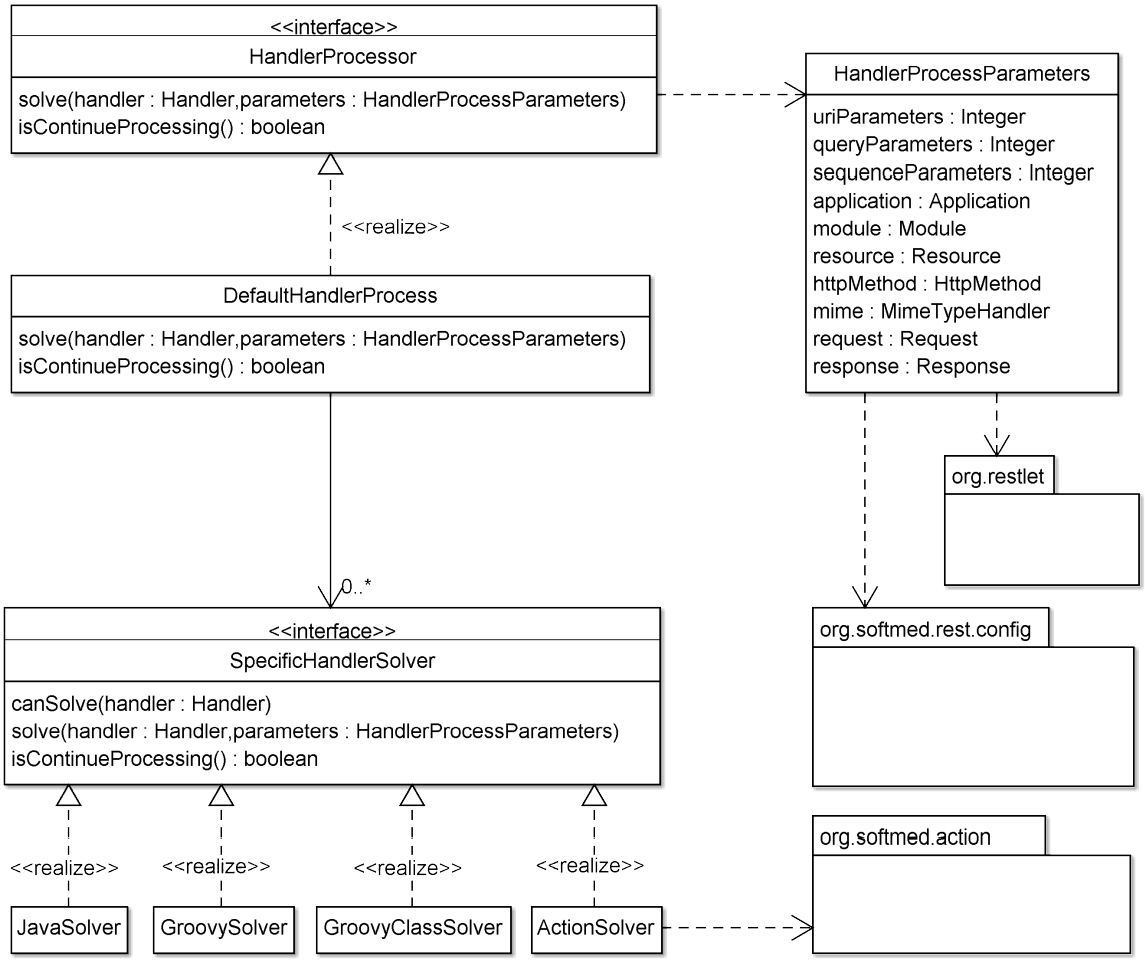

Figura 41: Arquitectura de processamento de handlers

A interface HandlerProcessor possui uma implementação por default, o DefaultHandlerProcessor,

disponibilizado neste projecto *core rest server*. O DefaultHandlerProcessor possui uma lista de implementações da interface SpecificHandlerSolver. Cada implementação suporta uma forma de lidar com handlers.

Este projecto tem já 4 implementações possíveis: JavaSolver, GroovyClassSolver, GroovyScript e o ActionSolver. Cada uma destas classes adiciona o suporte a handlers definidos como classes Java, classes groovy, script groovy e *acções*, respectivamente. Estas implementações são feitas no projecto *REST Server* e injectadas no DefaultHandlerProcessor via DI.

Ao receber um pedido para executar um handler, o DefaultHandlerProcessor percorre a lista de SpecificHandlerSolver, e invoca o método canSolve, com o handler como argumento.

Cada implementação de SpecificHandlerSolver determina se o atributo path do handler identifica algum tipo de recurso ou classe que consiga processar. Se assim for, retorna true e depois executa o handler(Figura 42).

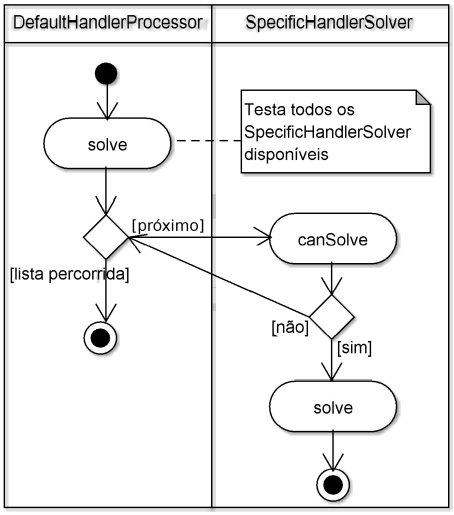

Figura 42: Processamento de um Handler

O projecto *core rest server* define assim:

- A arquitectura básica para obter a configuração de aplicações segundo a estrutura da especificação REST;
- Efectua o mapeamento da configuração para componentes da biblioteca RESTlet;
- A arquitecta de suporte à sequência de processamento conforme o método HTTP e tipo mime;
- Arquitectura de processamento de handlers, e;
- Mecanismo de refrescamento da configuração – RESTConfigRefresher.

Generation

O projecto de geração é o mais complexo de todos. Deve permitir gerar uma aplicação CRUD recorrendo a DDD, com acesso remoto via REST e solução de persistência. Analisando melhor este

objectivo, é necessário então:

- Uma forma fácil de definir aplicações e módulos;
- Algum modo de indicar que classes são parte do domínio do problema da aplicação;
- Providenciar um mecanismo de conversão de objectos para XML e vice-versa;
- Configuração do formato XML, tais como renomear atributos e classes ou não expôr certos atributos;
- Conjunto de handlers capaz de lidar com os pedidos de forma genérica, independentemente do tipos de classes da aplicação;
- Providenciar uma solução de persistência predefinia, que não requeira configuração do programador;
- Forma de indicar em que RouterSet colocar os recursos. Relembre que os serviços de administração são gerados por este projecto, e devem ser mantidos de cada vez que o servidor é actualizado. Como o RESTConfigurationRefresh remove todos os recursos que não os no RouterSet de nome "admin", é preciso haver forma de indicar em que RouterSet são colocados os recursos gerados para cada aplicação, e;
- Classe Facade para invocar o processo de geração automática - ScafoldFacade;
- Abstracção dos vários componentes por forma a permitir que o processo de geração seja alterável.;
- A extensibilidade do processo de geração, é preciso garantir que os programadores possam configurar as suas próprias soluções de uma forma simples;
- Cada aplicação deve poder definir os componentes a utilizar para o processo de geração, e;
- gerador de soluções que crie a configuração REST;

Gestão

Por forma a permitir que cada aplicação indique que componentes utilizar no processo de geração, e suportar várias aplicações no servidor então é necessário mecanismo para configurar e gerir os componentes, tais como as soluções de persistência. Para gerir os PersistenceProviders temos disponível o PersistenceManagerProvider. Resta gerir definir os três componentes essenciais para a geração automática de soluções CRUD : o próprio gerador da configuração, o conjunto de handlers genéricos que vão lidar com os pedidos e a solução de conversão em XML.

Para esse fim, definem-se três classes :

- **ModuleBuilderManager** – gere implementações de ModuleBuilder, uma interface que abstrai os detalhes de criação da configuração das aplicações.

- **HandlerSuiteManager**– gere objectos do tipo HandlerSuite, que identificam um conjunto de handlers genéricos para lidar com os pedidos.

- **XMLSuiteManager**– gere implementações de XMLConverterSuite, que definem a solução de conversão objecto-XML a utilizar.

Estas três classes usam uma estrutura de dados do tipo Map, associando a cada objecto gerido um nome para fácil identificação e acesso.

Estas três classes são referidas por uma classe ServiceGeneratorProcessor, responsável pela criação da configuração. Esta classe vai referir informações sobre cada aplicação e com bases nesses dados, escolhe os componentes correctos a partir das classes de gestão referidas anteriormente. Com base nesses componentes então inicia o processo de geração utilizando o ModuleBuilder escolhido.

Esta arquitectura pode ser visualizada na Figura 43:

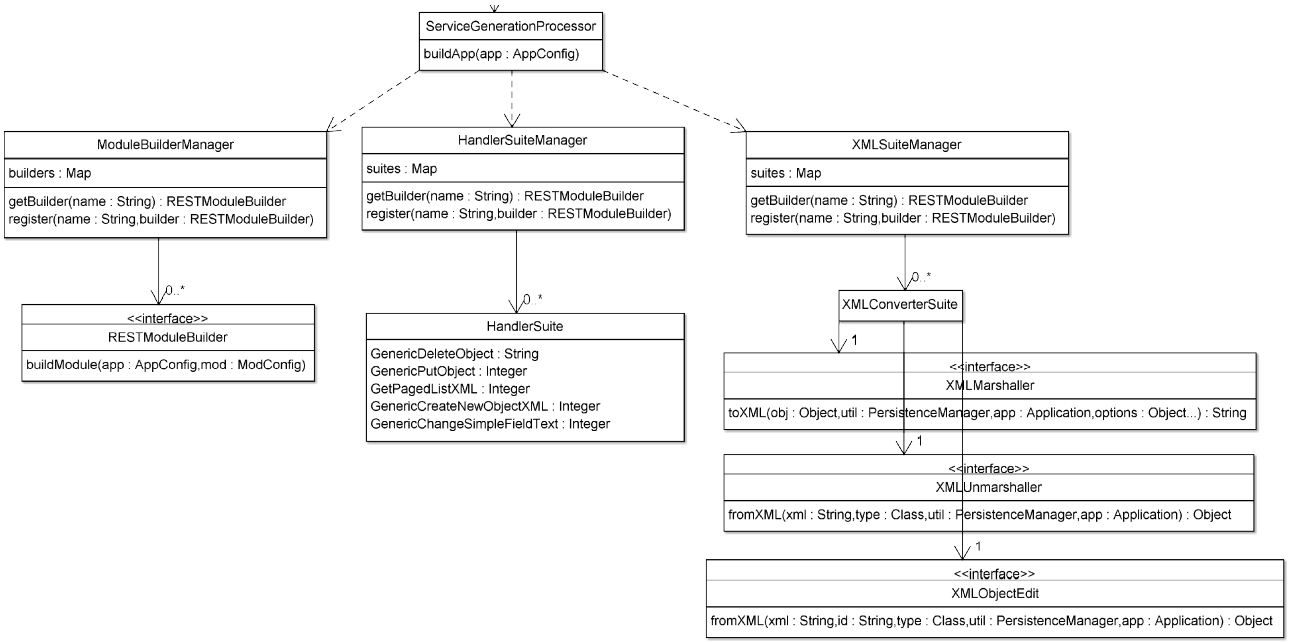

Figura 43: Gestão de componentes para geração de aplicações

O RESTModuleBuilder é o componente que realmente gera a configuração REST da aplicação. Sendo uma interface, deixa aberta a possibilidade de criar novas implementações. Este projecto já fornece uma implementação pré-definida.

O XMLConverterSuite indica um conjunto de ferramentas para conversão objecto-XML. Esta solução foi criada por é um formato popular para representar dados. É de referir estas ferramentas têm de representar os objectos segundo os princípios REST, ou seja, com uris para cada objecto e hyperlinks para referências a outros objectos. Portanto os algoritmos têm de lidar com esses aspectos.

Para esse fim o XMLConverterSuite é constituído por um XMLMarshaller , XMLUnmarshaller e XMLObjectEdit, interfaces que abstraem, respectivamente, os detalhes de como mapear um objecto para XML, mapear XML para um novo objecto ou editar um objecto existente conforme o

XML recebido, respectivamente. XMl e editam o objecto. O servidor já possui uma implementação default destas interfaces com recurso a XStream.

A classe HandlerSuite possui conjunto de atributos do tipo string que indicam os nome FQCN do código a executar como handler para cada tipo de pedido. Por exemplo para um HandlerSuite de nome "groovy " o handler para apagar objectos – campo genericDeleteObject - pode ter o valor de "scrips.groovy.delete", enquanto que numa outra HandlerSuite de nome "custom" esse campo pode ser "org.softmed.action.DeleteObject". Os HandlerSuite são utilizado para indicar o código genérico que lida com pedidos comuns, desde obter representação de objectos, a criar novos objectos, apagá-los, aceder atributos individuais, alterar esses atributos, ler listas, inserir objectos em listas, entre várias outras funções.

Meta-dados

É preciso obter um conjunto de informação adicional sobre cada aplicação de forma a gerar a sua funcionalidade. Essas informações devem ter valores comuns por default, para minimizar o esforço dos programadores. Antes de prosseguir com a exposição e necessário apresentar a estrutura de dados utilizada para conter as informações sobre as aplicações (Figura 44):

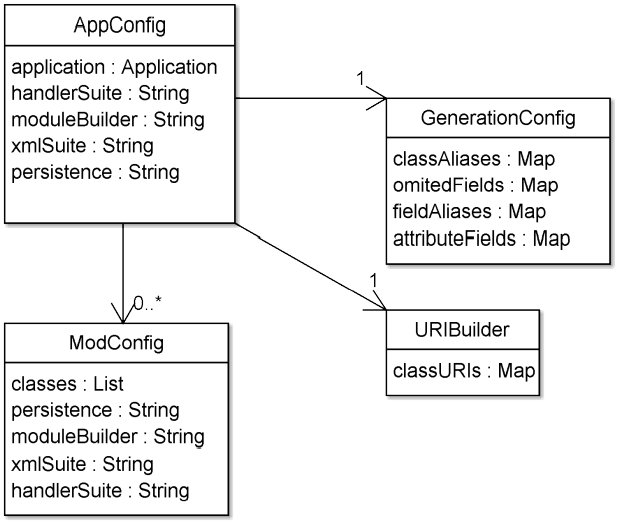

Figura 44: Diagrama de classes para dados sobre aplicações

O AppConfig possui campos string que identificam os componentes as usar na geração da configuração, nomeadamente, que ModuleBuilder, XMLSuite, HandlerSuite e PersistenceProvder utilizar. Estes mesmos campos estão disponíveis na classe ModConfig, que agrupa a informação sobre cada módulo da aplicação. Portanto, cada aplicação terá o seu Appconfig e cada módulo da aplicação terá um ModConfig.

É com base nestes campos que o ServiceGeneratorProcessor acede ao ModuleBuilderManager, HandlerSuiteManager e XMLSuiteManager para obter os componentes necessários à geração. Esta solução permite definir configuração para toda a aplicação ou para módulos individuais. Portanto

podem ser utilizadas estratégias diferente para módulos distintos na mesma aplicação caso seja necessário.

A classe URIBuilder é responsável pela criação dinâmica dos URIs dos objectos da aplicação. A classe GenerationConfig, por sua vez, detém um conjunto de dados de configuração do das classes e seus atributos para mapeamento XML e construção de URIs.

Na configuração gerada, os handlers devem estar configurados com parâmetros que indiquem que XMLConverterSuite e que PersistenceProvider utilizar no processamento de pedidos. Adicionalmente também são indicados como parâmetros a classe dos objectos e nomes de atributos por forma a que o código possa ser genérico e suporte qualquer tipo de classe em Java ou Groovy.

Facade de geração

Pode finalmente ser apresentada a arquitectura central para geração de aplicações no diagrama da Figura 45:

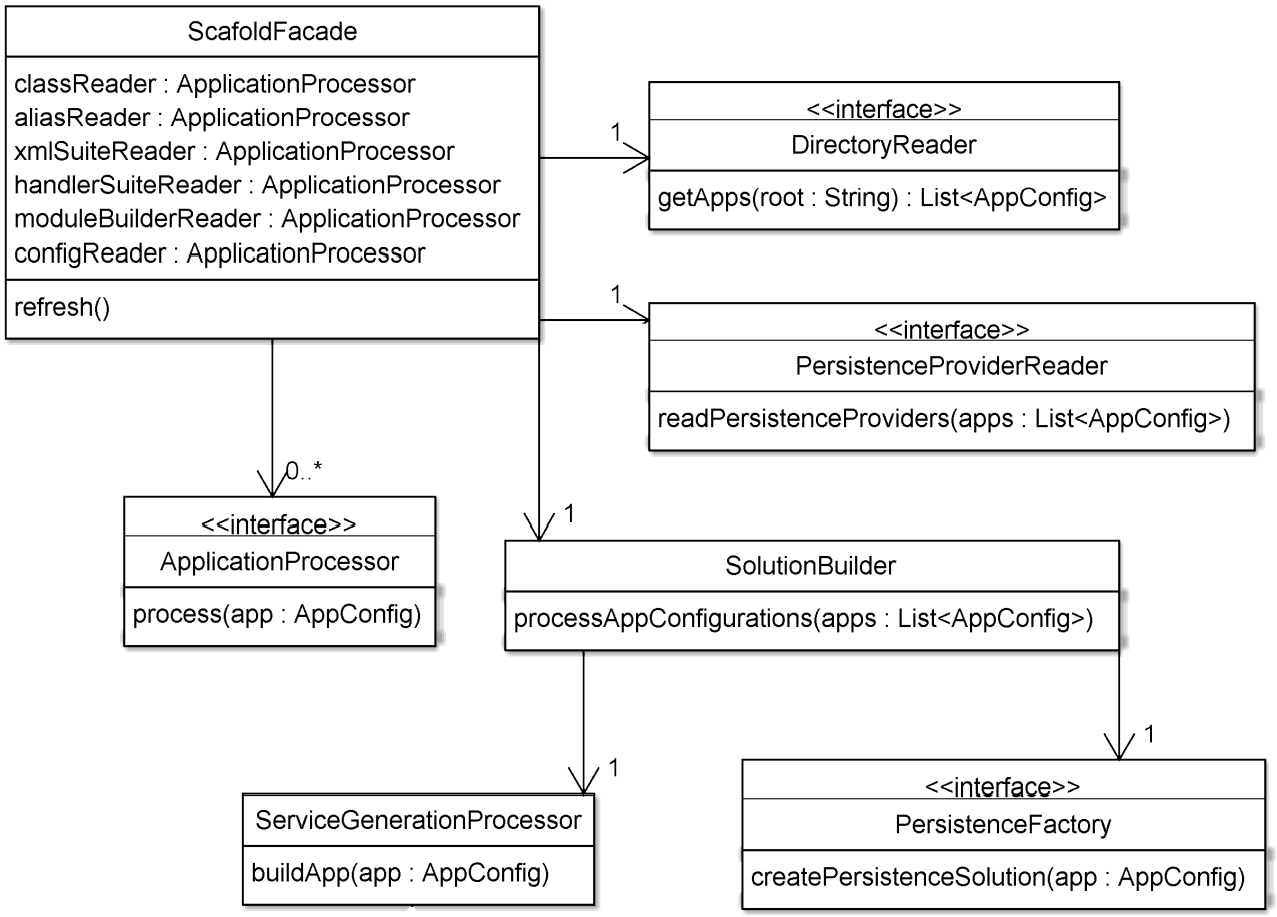

Figura 45: Arquitectura de geração de aplicações

A classe ScafoldFacade disponibiliza o método refresh por forma a despoletar todo o processo de geração. Nesta função são feitos os vários passos necessários à obtenção de informação e geração da aplicação. Para esse fim, o ScafoldFacade possui vários objectos do tipo ApplicationProcessor,

um DirectoryReader, PersistenceProviderReader e SolutionBuilder. Cada objecto tem uma função específica no processo de geração.

Em relação ao processo de obtenção de informação temos as seguintes etapas:

- Obter as aplicações e sua estrutura a partir da estrutura de directórios: DirectoryReader
- Obter as classes de cada aplicação: classReader
- Obter a configuração do output em XML: AliasReader
- Obter a solução de persistência a utilizar : PersistenceProviderReader
- Obter a solução de conversão objecto-XM:XmlSuiteReader
- Obter o conjunto de handlers que vão processar os pedidos - HandlerSuiteReader
- Obter o gerador da aplicação: moduleBuilderReader
- Obter o nome do RouterSet onde instalar os recursos da aplicação: configReader

A interface DirectoryReader é a responsável por determinar que aplicações existem, analisando a estrutura de directórios a partir do root do servidor e devolve uma lista de AppConfig. A classe AppConfig detêm a meta-informação necessária à geração de serviços. O DirectoryReader é uma interface por forma a abstrair os detalhes de acesso ao sistema de ficheiros, e a lógica de mapeamento da estrutura de directório para aplicações e módulos.

A lista de AppConfig retornada pelo DirectoryReader é passada por todos os outros objectos nesta sequência de leitura de dados. Cada objecto efectua a sua função e configura os AppConfig de acordo com o processado.

Finalmente após todos os dados terem sido recolhidos, pode-se gerara as soluções pelo SolutionBuilder. O SolutionBuilder testa se cada aplicação já possui uma solução de persistência configurada. Se não possuir, então utiliza o PersistenceFactory para criar uma solução pré-definida de persistência. A seguir o AppConfig é enviado para o ServiceGeneratorProcessor para criar a solução de serviços, como indicado anteriomente .

O PersistenceFactory é uma interface por forma a que a solução default da persistência possa ser alterada facilmente. Essa solução tem de ser compatível com a PersistenceAPI como definida no capítulo de especificação. Existe servidor já possui uma implementação de PersistenceFactory pré-definida que utiliza a base de dados NeodatisODB. Esta solução é muito fácil de utilizar e permite configurar as “chaves primárias” e a sua geração pela API da biblioteca NeodatisUtil.

A próxima secção é apresentada a arquitectura desenvolvida para a acções definidas como máquinas de estado finitas na biblioteca jtAction. Depois dessa apresentação introduz-se então a solução de AOP criada.

jtAction

É uma biblioteca genérica que define acções como máquinas de estado finitas. Portanto cada acção é constituída por vários estados onde cada estado pode conter uma série de passos, os ActionSteps. Como os próprios estados são ActionStep, isso possibilita inserir estados dentro de outros estados, criando máquinas mais complexas. (Figura 46 (A))

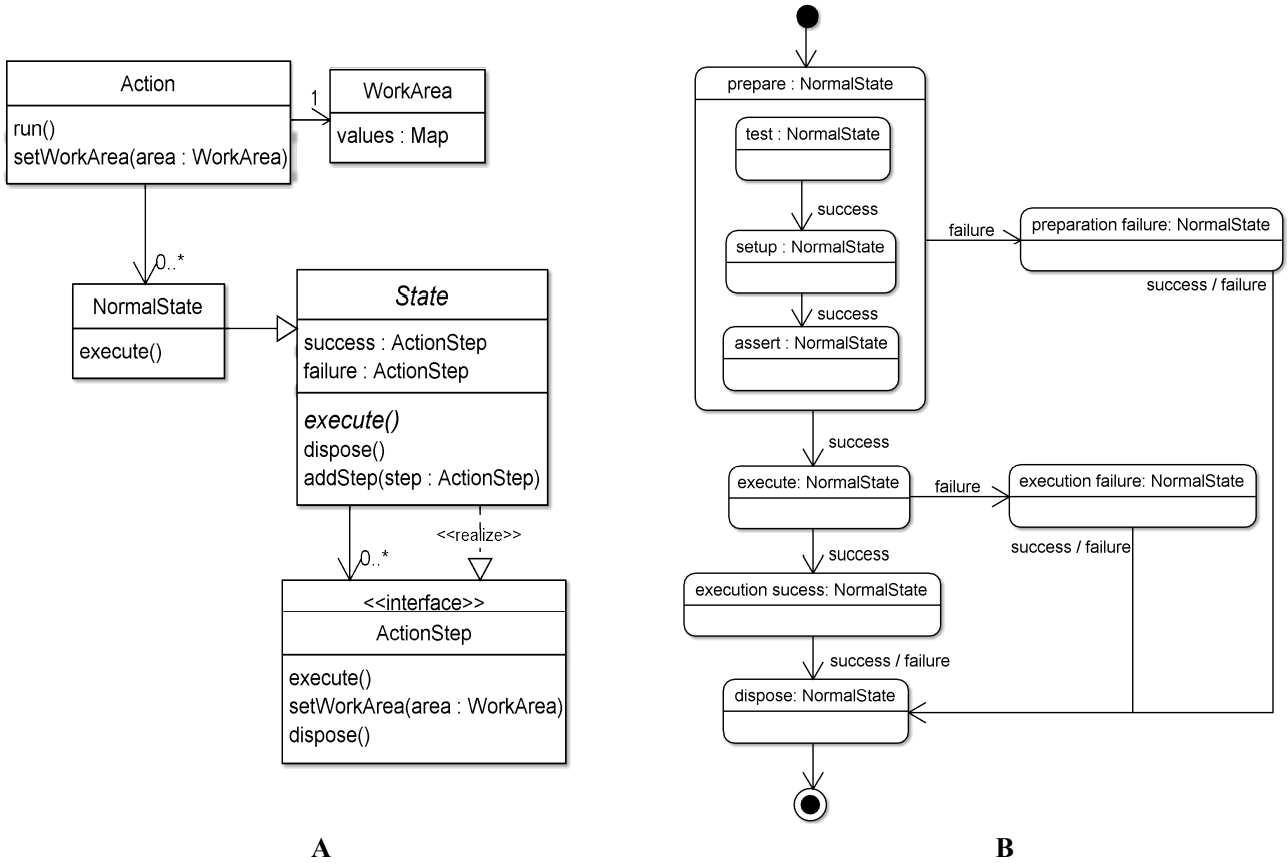

Figura 46: Classes de definição de Acções(A) e instância de Acção (B)

Cada estado tem duas transições pré-definidas, uma para o caso de sucesso e outra no caso de falha. Existe um mecanismo para criação de transições adicionais via adição de eventos e associação entre estados mas para o caso desta aplicação bastam estas duas transições base, pelo que esse mecanismo não é descrito no âmbito deste trabalho. As transições indicam para que estado se transita no caso de falha e no caso sucesso.

Com base nesta estrutura de dados é possível criar a acção propriamente dita, instanciando vários estados e definindo as suas transições. O resultado é a máquina de estados finita na Figura 46 (B):

A criação das *acções* depende da implementação de uma interface ActionCreator. A biblioteca jtAction inclui uma implementação por default, a classe DefaultActionCreator. (Figura 47) Esta classe utiliza a interface ActionDescriptioXMLConverter para abstracção dos detalhes de conversão acção-XML. A biblioteca já incluí uma implementação de ActionDescriptioXMLConverter conversor com recurso à biblioteca XStream denominado DefaultXStreamXMLConverter.

É de referir que o que se obtém do XML não é uma instância de acção em si, mas sim a descrição da acção, os seus estados e passos. Essa descrição consiste num objecto do tipo ActionDescription com vários StateManager, e cada StateManager com uma lista de ActionStepDescriptor. Estas classes de descrição são necessárias porque os ActionStep precisam de ser instanciados a partir do atributo *path* do ActionStepDescription. Este atributo identifica o código a ser executado como ActionStep recorrendo à forma FQCN .

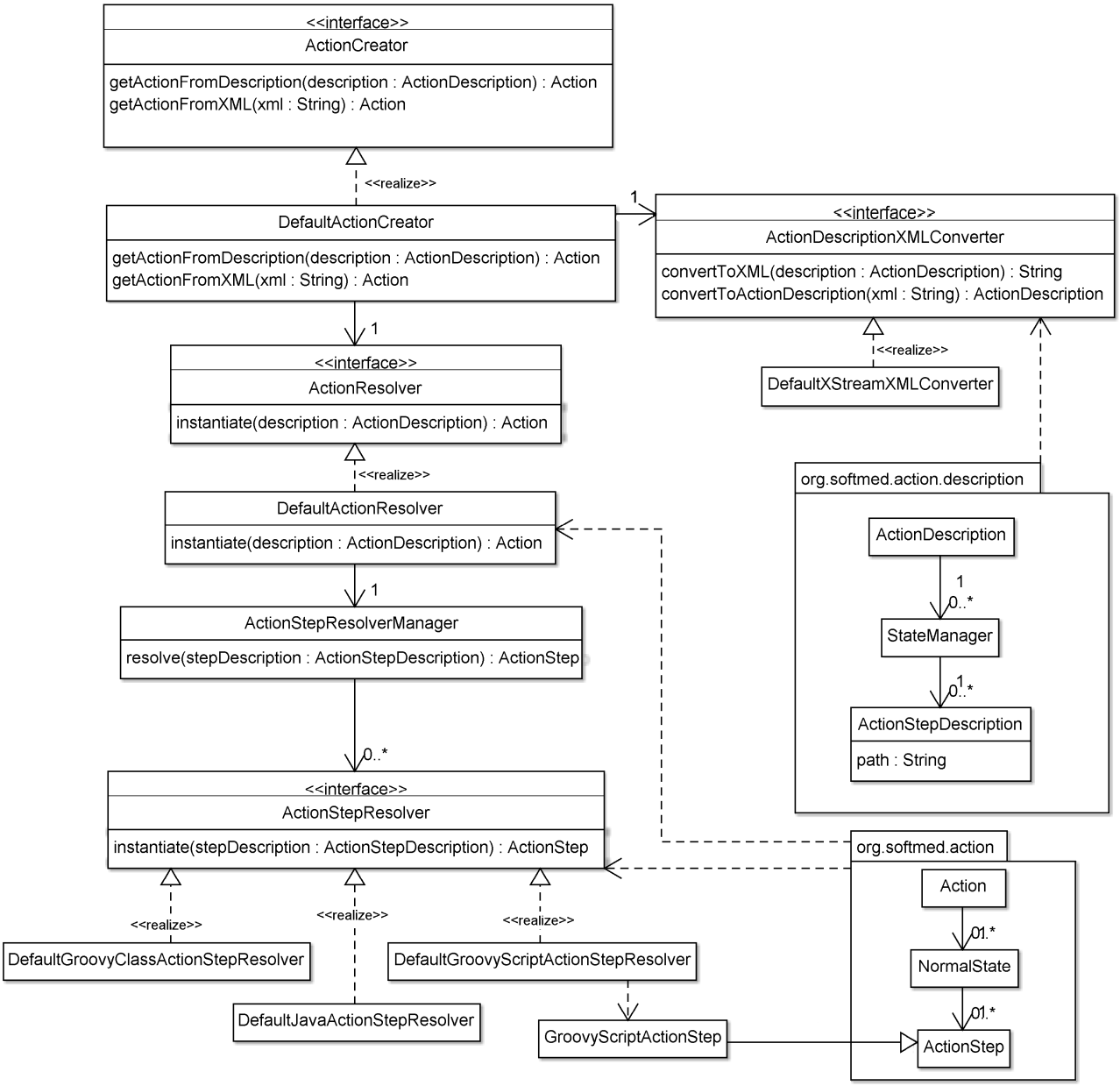

Figura 47: Arquitectura para criação de Acções

Após obtenção da descrição, passa-e à criação do Action propriamente dito. Esta função fica a cargo da implementação da interface ActionResolver. O servidor já possui uma implementação default, o DefaultActionResolver, que recebe o ActionDescriptioin e devolve uma Action.

Para essa função é necessário determinar cada ActionStep. O DefaultActionResolver possui assim um ActionStepResolverManager, que gere vários ActionStepResolver. Os ActionStepResolver é que

realmente recebem um ActionStepDescription, e verificam se o recurso ou classe identificado pelo atributo *path* na forma FQCN pode ser utilizado ou compilado de forma a retornar um ActionStep a ser usado na construção da Action.

O servidor já possui de base quatro ActionStepResolver distintos, cada um suportando um tipo diferente de implementação de ActionStep : classes Java, classes Groovy e scripts groovy. No caso do script groovy, como este não define nenhuma subclasse de ActionStep, então o script é compilado e colocado num GroovyScriptActionStep.

O diagrama de sequência Figura 48 demonstra o processo de criação de acções.

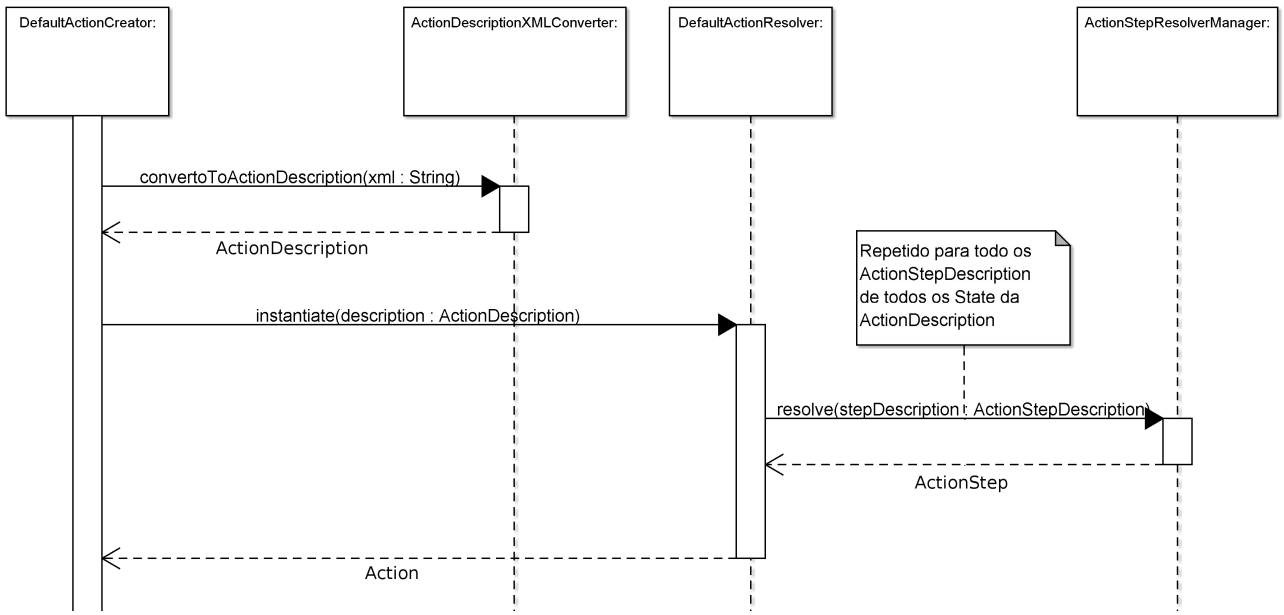

Figura 48: Criação de Acção

Um mesmo estado pode conter várias implementações diferentes de ActionStep, pelo que até é possível utilizar diferentes linguagens de programação na mesma acção. Utilizando a API da acção, esses detalhes ficam escondidos.

A questão de permitir passar informação de um ActionStep para o próximo é resolvidas com a WorkArea, uma classe que serve para partilha de dados entre os vários ActionSteps. A WorkArea possui um mapa de pares nome-objecto que representam variáveis. Qualquer ActionStep recebe essa estrutura e pode ler e colocar valores e para outros ActionStep.

No no contexto de acções para processar pedidos HTTP, o ActionSolver, a implementação que retorna handler a partir de acções, recebe o objecto HandlerParameters com os dados do pedido. O ActionSolver recorre ao ActionCreator para obter a *acção* e coloca todos os campos do HandlerParameters como variáveis na WorkArea. Assim, todo os passos da acções têm acesso aos objectos de request, response, application, entre vários outros.

Esta biblioteca oferece uma solução flexível para estruturação de acções a partir de passos pequenos e reutilizáveis. É uma biblioteca genérica, podendo ser utilizada noutras áreas, por exemplo como controlador numa aplicação MVC desktop.

AOP

A solução de AOP começa por definir as classes correspondes à especificação de aspects, advice e rules como indicado no diagrama da Figura 49:

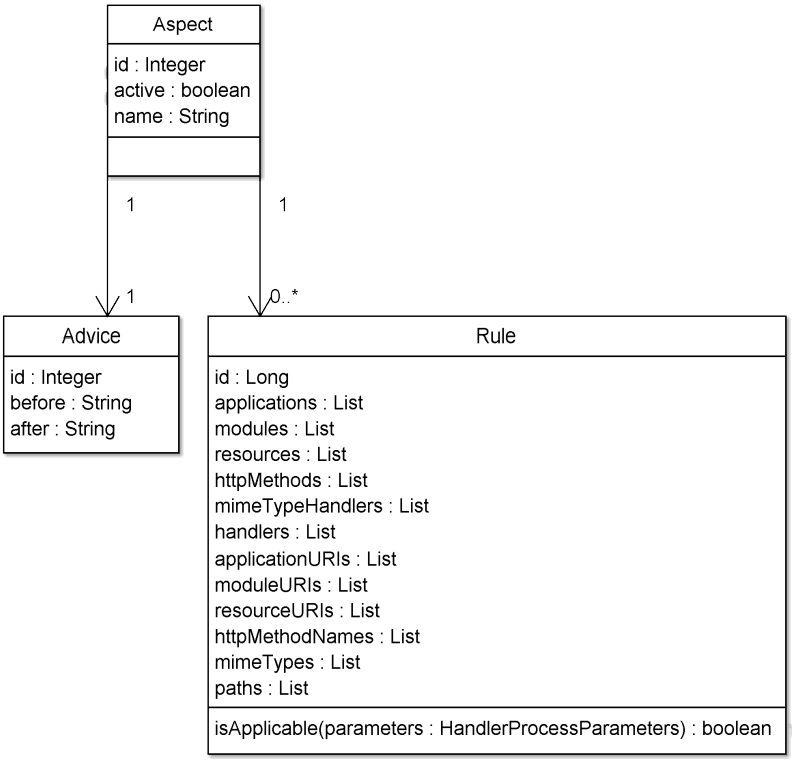

Figura 49: Classes para definição de Aspects

O serviço de administração depende das capacidades de geração anteriores descritas pelo projecto de G*eneration*, como explicado no capítulo de implementação.

O problema a resolver é como obter os aspecto existentes, e aplicá-los às acções. Como visto na Figura 41 do *core rest server*, uma das implementações disponíveis de SpecificHandlerSolver, o ActionSolver, retorna handlers a partir de *acções*.

Por questões de performance, não é eficiente estar sempre a ler a um ficheiro XML com a descrição de uma acção, instanciar a acção a partir da descrição, e só então processar o pedido. A solução é uma ActionCache, uma pool de instâncias da mesma acção, que podem ser pedidas para processar o pedido e depois libertadas, retornado à cache. Assim a acção só é lida uma vez a partir do ficheiro, e quando um pedido é recebido, só é necessário obter uma instância da acção a partir da cache.

Por forma a suportar alterações dinâmicas das acções, sempre que se tenta obter uma acção,

verifica-se o timestamp do ficheiro XML que a define. Se se verificar uma alteração, então obtém-se a acção actualizada e reconstrói-se a cache para aquela acção. É neste processo de obtenção de acções que se pose aplica os *aspects*. (Figura 50)

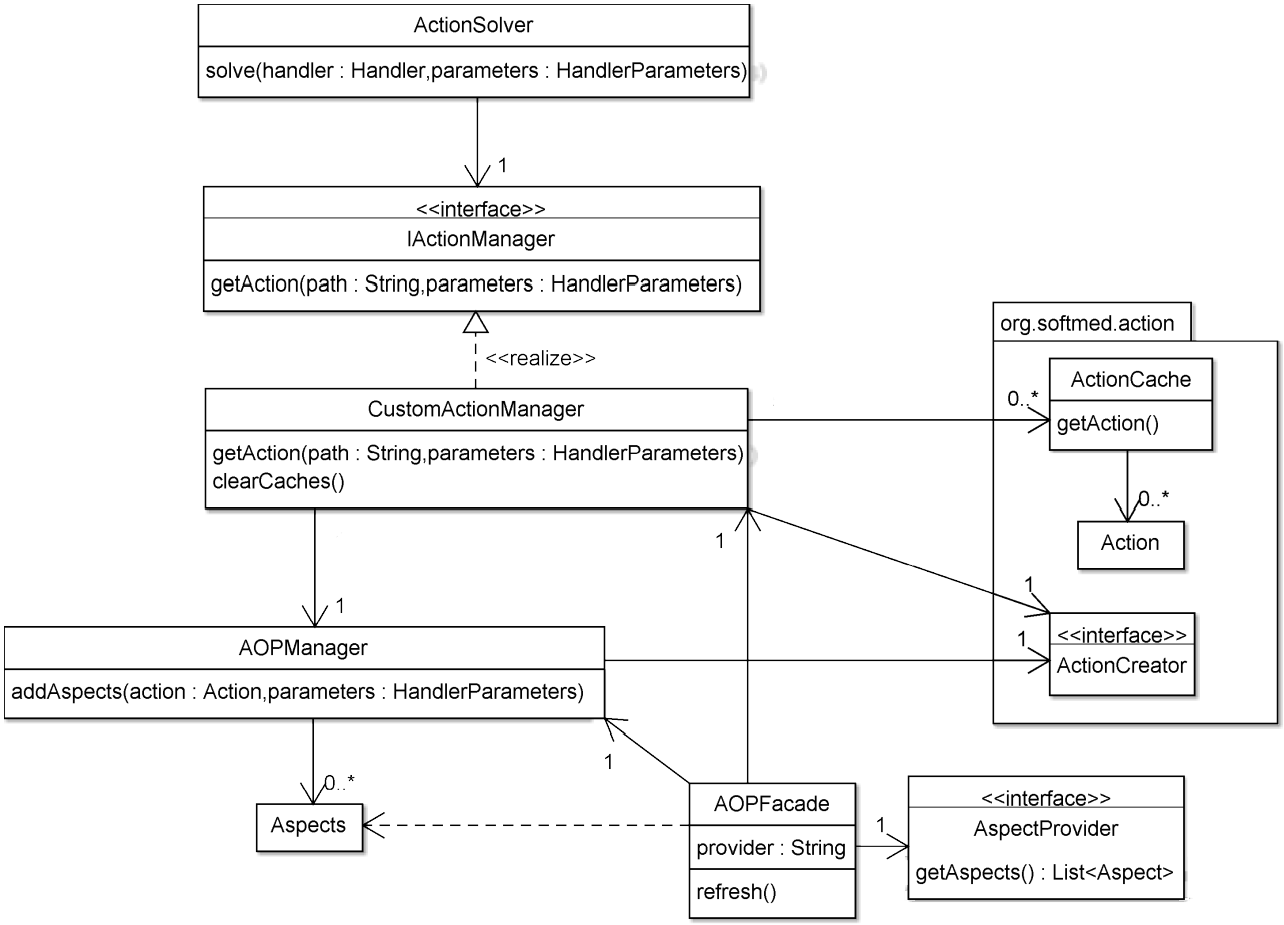

Figura 50: Arquitectura de AOP

O ActionSolver depende da interface IActionManager. A implementação por default que gere a questão das caches e da aplicação dos aspects é o CustomActionManager. Esta classe possui uma referência ao AOPManager, que gere uma lista de aspectos e possui um método para a sua aplicação em acções.

O diagrama da Figura 51 mostra a sequência para a obtenção de acções caso exista já uma cache com várias cópias.

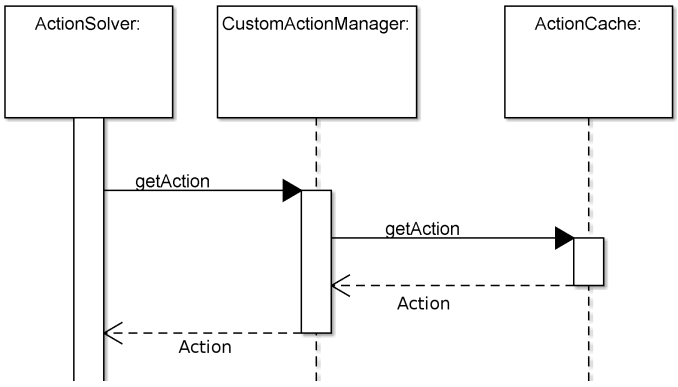

Figura 51: Obtenção de Acção com cache válida

Quando ainda não existe uma cache para a acção, então é efectuada a sequência da Figura 52, onde vemos o processo de construção e aplicação dos aspectos:

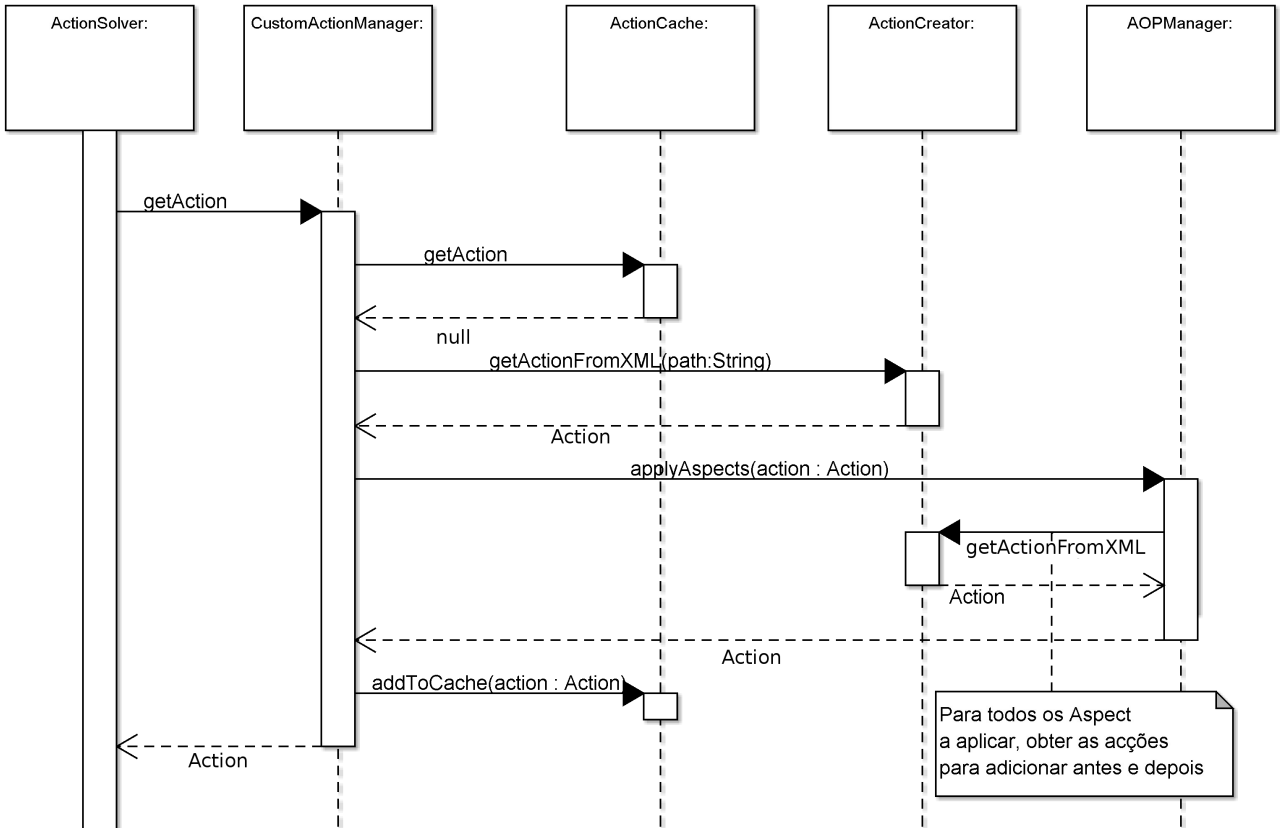

Figura 52: Obtenção de Acção sem cache válida

Vemos que neste caso que a cache retornou *null* pois não existe ainda nenhumas cópias da acção. Então é feito um pedido à interface ActionCreator para retornar uma Acção a partir do path do handler recebido pelo ActionSolver. A interface ActionCreator foi apresentada anteriormente na arquitectura da biblioteca jtAction e abstrai todos os detalhes de guardar a descrição de acções, em que formatos, que bibliotecas utiliza e a lógica de configuração dos estados.

Após a acção ser recebida, os aspectos são aplicados via o AOPManager Este testa os aspectos conforma as suas regras para determinar quais a aplicar. O AOPManager, segundo a nomenclatura própria de *Aspect Oriented Programming* é assim o *weaver* do serrvidor, integrados os advice com as acções.

Finalmente é criada a cache para a acção alterada. Ou seja, todas as instâncias na cache são cópias da acção alterada, portanto em todas essa cópias estão aplicados os aspectos.

Questão de como é que o AOPManager vai obter a lista dos aspectos disponívieis. Essa lista é colocada no AOPManager pelo AOPFacade. Quando o método refresh é invocado, o AOPFaccade recorre ao AspectProvider para obter uma lista de aspectos a aplicar. O AspectoProvider é uma interface, pelo que esconde todos os detalhes para obtenção dos aspectos. Estes podem ser obtidos a partir de um ficheiro XML ou base de dados, entre várias alternativas.

Actualização

Por forma a controlar a actualização dos aspects no servidor, foi criada a classe AOPFacade. A invocação do método refresh dessa classe resulta na sequência representada na Figura 53:

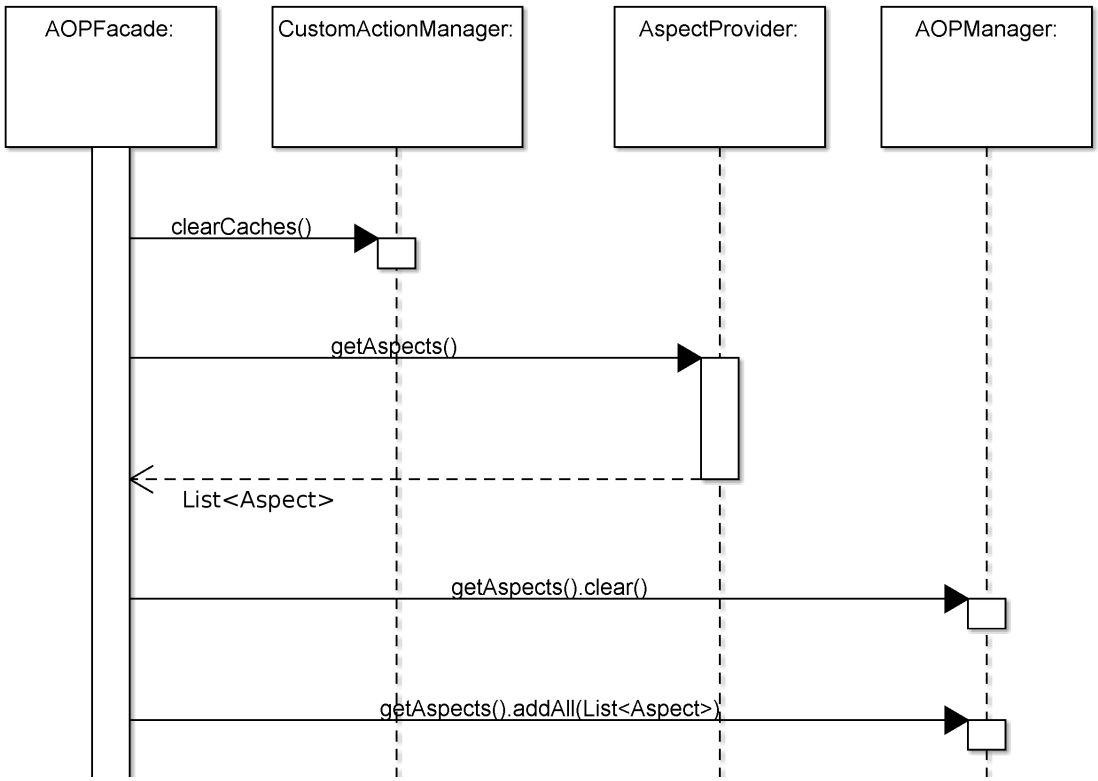

Figura 53: Sequência de actualização de AOP

O processo de actualização decorre da seguinte forma : o AOPfacade invoca o método clearCaches do CustomActionManager, invalidando toda as caches e libertando as cópias de acções. De seguida é invocado o método getAspects do AspectProvider para retornar os aspectos existentes no servidor. Finalmente o AOPFacade limpa os aspectos antigos que estavam no AOPManager e coloca a nova lista.

A partir deste momento, qualquer pedido que seja processado por um jtAction, como não existe caches válidas, vai levar à construção de nova cache pelo processo da figura X3, e assim à aplicação dos aspects do AOPManager.

REST server

Consiste no projecto de integração do servidor. Neste projecto é efectuada a implementação de interfaces essenciais a cada projecto anterior, e configuração das dependências a injectar via Guice em cada componente. Como tal não possui arquitectura própria, mas sim utiliza as arquitecturas definidas nos outros projectos.

Os componentes aqui implementados são :

- **DefaultBoorProcessor** – implementa o processo de iniciação do servidor. Por *default* recorre a scripts groovy executados em sequência, como explicado no capitulo de implementação.
- **SpecificHandlerSolver** - implementa suporte a quatro formas de handlers : classes Java, clases groovy, scripts groovy e jtActions.
- **DefaultScafoldfHelpers** -implementação dos componentes requeridos para a geração de aplicações, nomeadamente o DirectoryReader, classReader, AliasReader, DefaultPersistenceRreader, xmlSuiteReader, handlerSuiteReader, ModuleBuilderReader e ConfigReader. Estes componentes pré-definidos recorrem a ficheiros groovy como indicado no capítulo de implementação.
- **DefaultODBPersistence** – implementação da PersistenceAPI a partir da base de dados NeodatisODB e a biblioteca auxiliar NeodatisUtils.
- **DefaultPersistenceFactory** – implementação de PersistenceFactory, utilizada para construir a base de dados default em aplicações sem nenhuma solução inicial. Esta classe retorna uma base de dados em NeodatisODB utilizando a DefaultODBPersistence.
- **DefaultModuleBuilder** – implementação pré-definida para o componente que gera a configuração REST dos módulos das aplicações.
- **DefaultHandlerSuite** - define um conjunto de handlers genéricos baseados em acções definidas com a biblioteca jtAction. Estes handlers são utilizados para a funcionalidade das aplicações geradas automaticamente. Portanto estes handlers suportam: criar, editar e apagar objectos, aceder e alterar campos desses objectos, acesso a listas e a elementos únicos da lista entre várias outras opções.
- **DefaultApplicationProvider** – implementação de ApplicationProvider que retorna as configurações definidas manualmente no servidor. Estes dados são obtidos a partir da base de dados do módulo de *rest* da aplicação *admin*, como indicado no capítulo de implementação.
- **DefaultAspectProvider** implementação de AspectProvider que retorna os aspects existentes no servidor. Estes dados são obtidos a partir da base de dados do módulo de *aop* da aplicação *admin*, como indicado no capítulo de implementação.

O projecto REST Server é o que configura todos componentes por injecção de dependências recorrendo à biblioteca Guice. Assim, todos os projectos são integrados resultando num servidor funcional, possibilitando a criação de aplicações seguindo uma arquitectura *Model Controller Service*.

Anexo V - Benchmarks gerais

Os valores calculados permitem-nos finalmente iniciar a análise da performance e retirar várias conclusões. Iniciaremos com o teste de POST, ou seja, da criação de 500 clientes.

POST

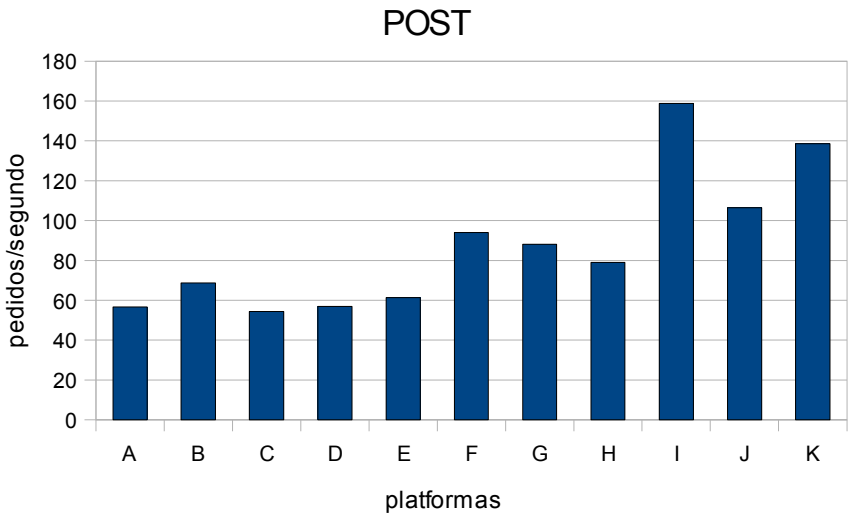

Figura 54: Resultados do teste de criação de Clientes (POST)

Como ilustrado na Figura 54, a plataforma I (Tomcat + Java + MySQL) destaca-se imediatamente, tal como previsto, seguida da plataforma K (Tomcat + Java + Neodatis ODB) o que mostra uma boa performance da base de dados Neodatis, sobretudo tendo em conta que é um projecto tão recente. Logo abaixo, vemos também uma dominância das plataformas com recurso a JPA/Hibernate/MySQL sobre todas as outras.

Em geral, as plataformas com base no Tomcat são mais rápidas que as outras, mostrando que ainda há caminho a percorrer para a biblioteca RESTlet. A plataforma H é 30% mais rápida que a E, sendo a única diferença entre as duas o uso do RESTlet na E e do Tomcat na H.

Podemos também verificar que o uso de ORM aumenta bastante a performance em relação a soluções sem pooling de ligações, como visto pela diferença entre as plataforma E (MySQL sem pooling) e as plataformas F e G, ambas com ORM. O mesmo pode ser confirmado entre as plataformas H e J.

GET single

A leitura de um cliente individual (Figura 55) reflecte uma diferença ainda maior em relação à plataforma I e todas as outras. Novamente a plataforma K está em segundo lugar, devido à alta performance da base de dados Neodatis.

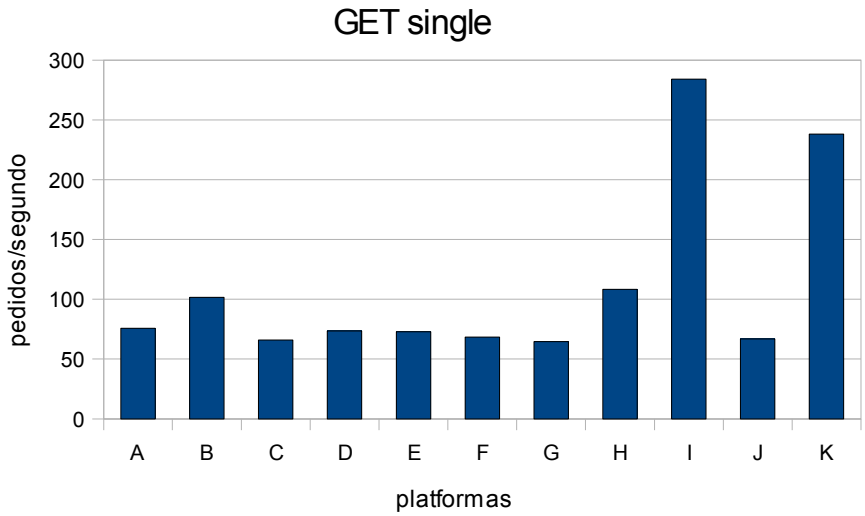

Figura 55: Resultados do teste de leitura de dados de clientes (GET single)

Também podemos ver o impacto das tecnologias ORM (JPA/Hibernate/MySQL) na diferença entre as solução I e J , pois a I com o SQL manual a ser 325% - 3.25 vezes - mais rápida que a J, com ORM.

Neste caso, as plataformas A e B ficam ambas à frente não só da J – 13% e 52%, respectivamente - como também de 5 outras plataformas : C a G.

Está comprovado que o projecto, pelo menos para este teste e mesmo na versão alfa actual, pode ser mais rápido que uma solução standard da indústria como a I.

GET list

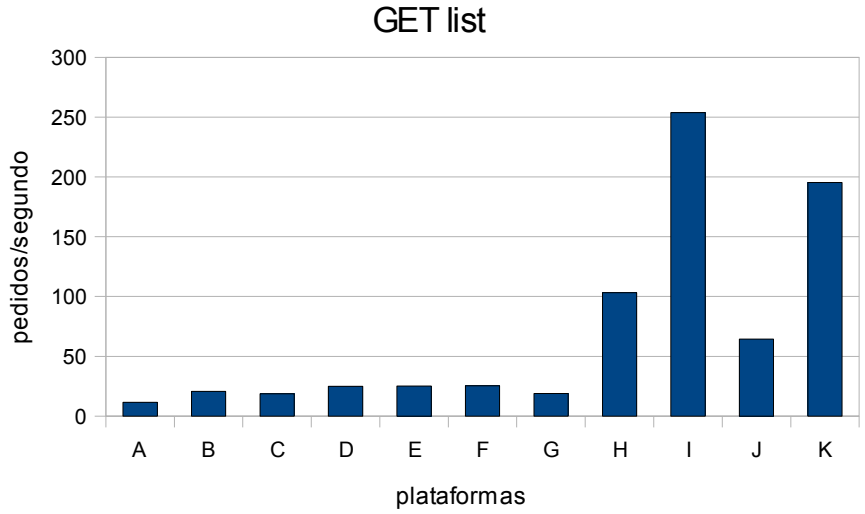

Figura 56: Resultados do teste de obtenção de lista de clientes (GET list)

Os resultados do teste de obtenção da uma lista de 20 clientes, como vistos na Figura 56 revelam um dado crucial: a biblioteca RESTlet é muito lenta a entregar respostas com um corpo já com alguma dimensão. Já foi visto que no geral o RESTlet é mais lento do que o Tomcat, mas nestas

condições a diferença é muito maior. Todas as plataformas que recorrem ao RESTlet como servidor ficam muito abaixo de qualquer uma que utilize o Tomcat.

Possivelmente deve-se a algum bug ou falta de optimização na realização de pedidos em que a resposta já é mais extensa do que algum número de bytes. Seriam necessários testes adicionais para verificar exactamente o que se passa, sobretudo se a performance descai gradualmente com o aumento do tamanho da mensagem, ou se há algum valor específico a partir do qual a performance cai abruptamente.

Sem dúvida que neste aspecto vemos a juventude de um projecto como a biblioteca RESTlet. De resto, o dado mais relevante é o impacto de performance na utilização do ORM, com a solução I a ser 3 vezes mais rápida que a J. (Figura 56)

PUT

O caso da edição de um cliente individual mostra um cenário em geral muito semelhante ao da sua criação via POST, com a excepção da plataforma I a se destacar mais em relação às outras, pelo que o SQL manual é muito eficiente neste tipo de operações.(Figura 57)

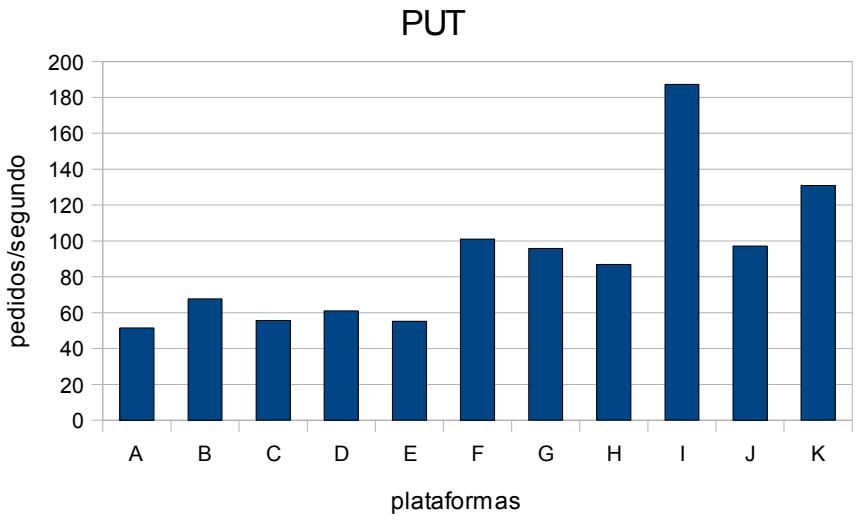

Figura 57: Resultados do teste de edição de clientes (PUT)

DELETE

Para o teste de remover clientes individuais o resultados confirmam as tendências anteriores(Figura 58): dominância do Tomcat, e de soluções com pooling de ligações à base de dados, mesmo que usadas com ORM.

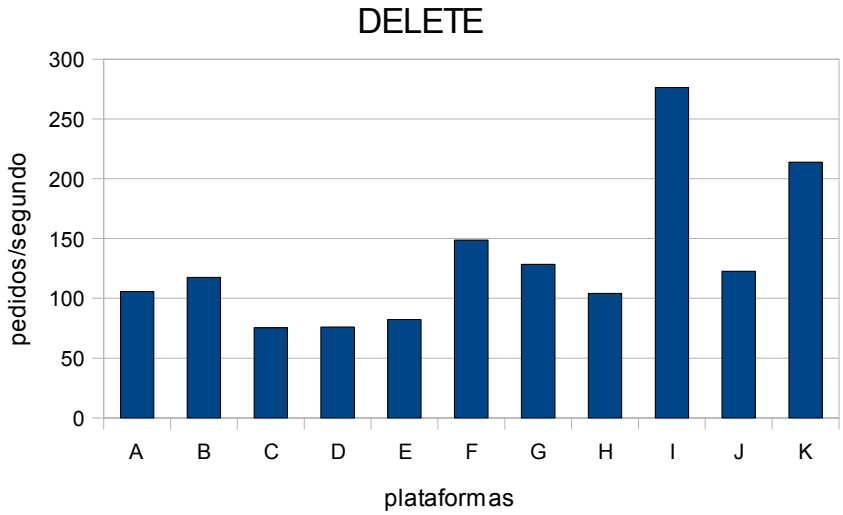

Figura 58: Resultados do teste de remoção de clientes (DELETE)

Adicionalmente, vemos novamente que as plataformas A e B baseadas no REST Server podem chegar perto da plataforma J com Tomcat e ORM, ficando a 14% e 4% da performance da J, respectivamente.

Anexo VI - Benchmarks específicos

Código

A comparação da plataforma A e B, onde a única diferença é a o uso de uma solução geral na A e código escrito especificamente para a aplicação em causa na B, permite calcular o impacto dessa solução genérica.

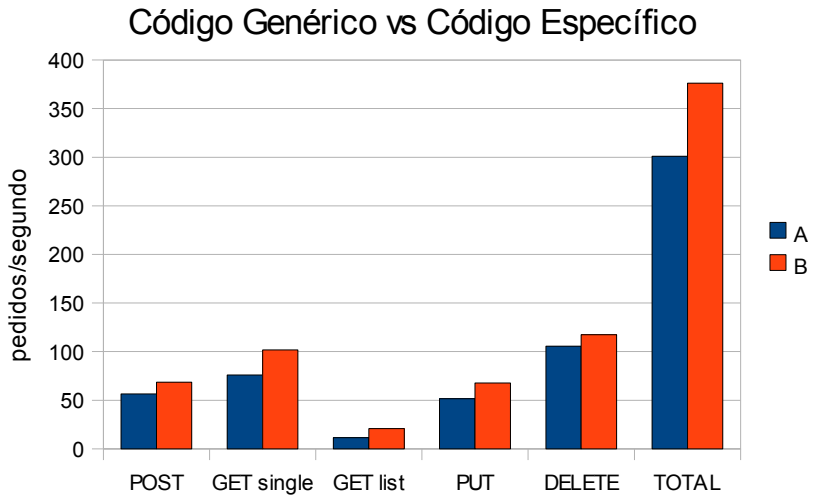

Figura 59: Comparação entre solução genérica e específica da framework Tábula

Podemos concluir que o impacto em geral é de uma baixa de 20% na performance, como ilustrado na Figura 59.

Outro aspecto a estudar é a baixa em performance provocada pelo uso de scripting em Groovy em vez de código Java compilado. Para esse fim, temos dois conjuntos de plataformas a analisar, a D e E bem como a F e G.

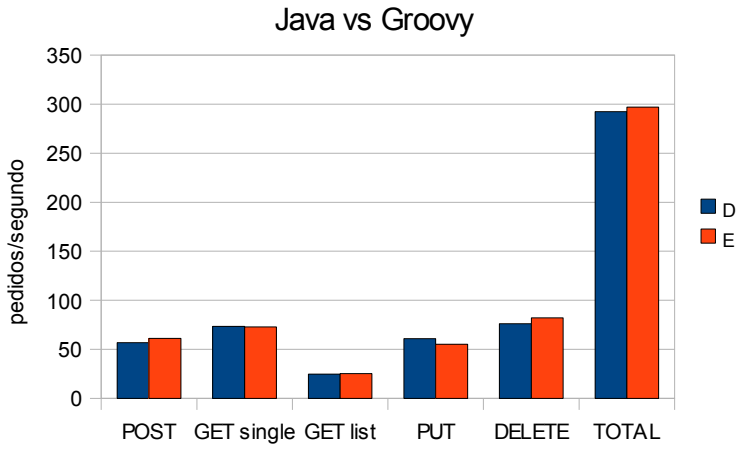

Figura 60: Comparação de resultados das plataformas D e E

Na Figura 60 temos uma diferença mínima de pouco mais de 1%, o que justifica largamente o uso

de Groovy, já que permite mudar a implementação sem ser necessário reiniciar o servidor.

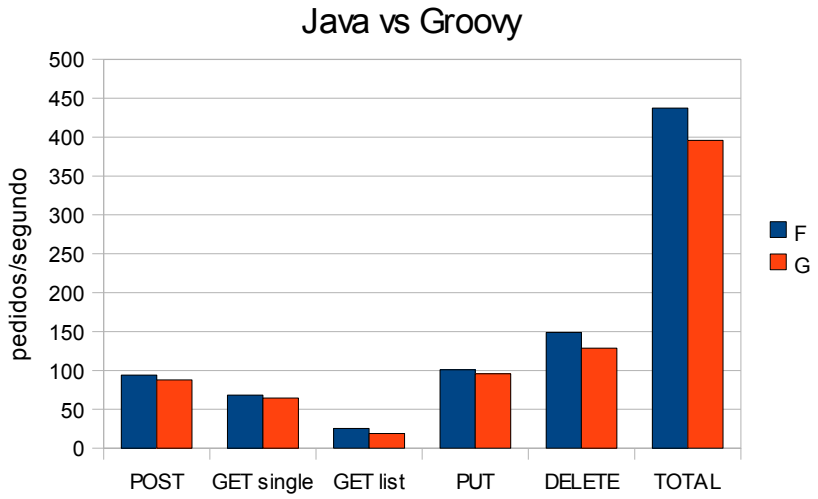

Figura 61: Comparação de resultados das plataformas F e G

Na situação da plataforma F e G (Figura 61), onde é usado o JPA/Hibernate/MySQL é verificada uma diferença maior, de cerca de 11%. Este aumento em relação à comparação anterior pode ser explicado devido às plataformas F e G utilizarem ORM com pooling de ligações. Embora o ORM tenha o seu custo em performance, ele é compensado pelo pooling. O resultado é que a plataforma F e G são mais rápidas que as E e D.

Ou seja, na situação anterior tínhamos a performance limitada sobretudo pela base de dados, daí que a diferença entre Groovy e Java não seja notória. Mas com pooling de ligações, a base de dados já é um factor menor na determinação da performance e notam-se melhor os efeitos da diferença entre o scripting e o código em Java.

Servidor

Uma das questões a avaliar é como o REST Server se compara em relação ao uso directo da biblioteca RESTlet. Embora o servidor use o RESTlet, ele adiciona também código extra que

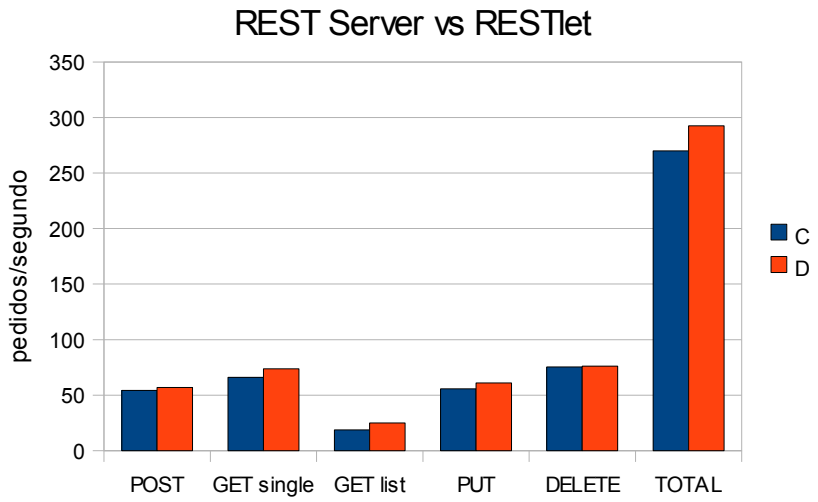

Figura 62: Comparação de resultados das plataformas C e D

permite a configuração em *runtime* da camada de serviços. Esse código terá impacto na performance, resta determinar o valor desse impacto.

Conforme ilustrado na Figura 62, as plataformas C e D são ideais para determinar o peso do código adicional, chegando ao valor de valor de 8% em termos de perda de performance.

De seguida é comparada a biblioteca RESTlet contra o bem estabelecido servidor Tomcat. Já vimos que o RESTlet em geral é mais lento que o Tomcat, possuindo um grande défice na situação particular de devolver listas de clientes.

Na Figura 63 são comparadas as plataformas E e H, onde é usado o MySQL sem pooling, e a diferença é o uso de RESTlet na E do Tomcat na H.

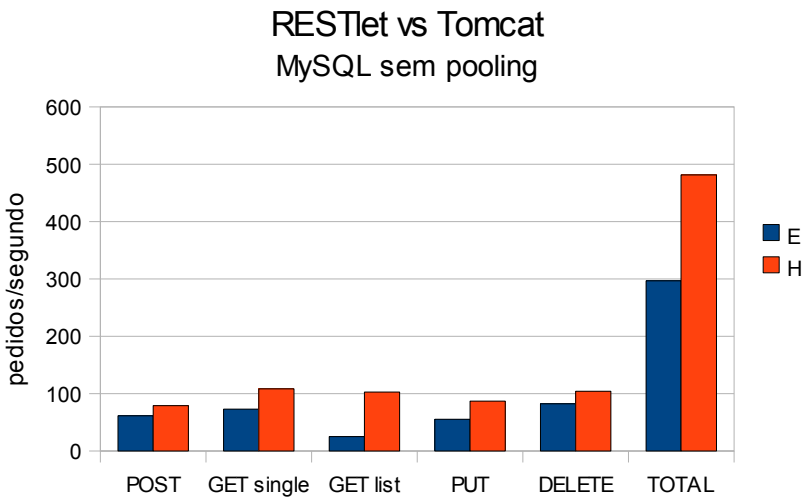

Figura 63: Comparação de resultados das plataformas E e H

Pode ser verificado que existe uma diferença considerável, sendo a solução em RESTlet 38% mais lenta em relação à solução em Tomcat.

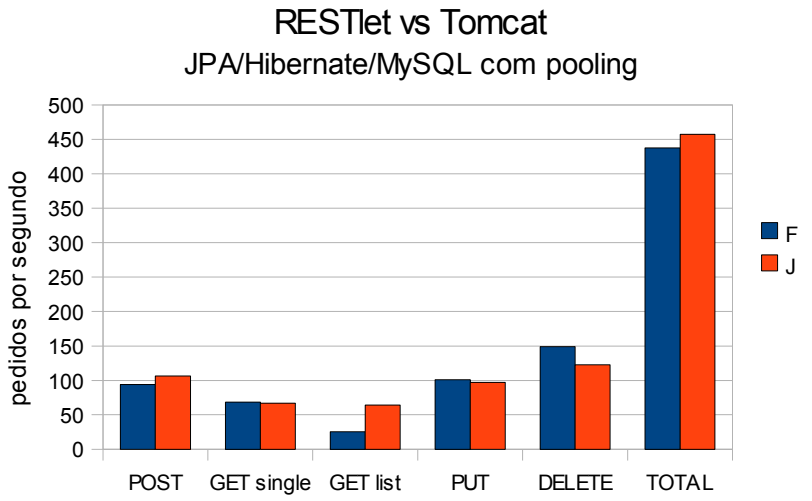

Figura 64: Comparação de resultados das plataformas F e J

Já na situação que é usado ORM (Figura 64) com pooling de ligações a diferença é menor, cerca de

4%, e existem vários testes onde a solução em RESTlet - F - ultrapassa a solução em Tomcat – J - , tais como GET single, PUT e DELETE. No entanto, esses ganhos são anulados pela má prestação do RESTlet no teste de GET list.

Bases de Dados

A base de dados Neodatis não possui pooling de ligações, e por isso mesmo, foi decidido que na maioria das plataformas seria usado o MySQL sem pooling, de modo a utilizar situações idênticas.

Uma questão é saber que diferença faz utilizar ou não o pooling de ligações no MySQL. Para esse fim são comparadas as plataformas H e I, no Tomcat, onde a diferença é apenas o uso de pooling na I.(Figura 65)

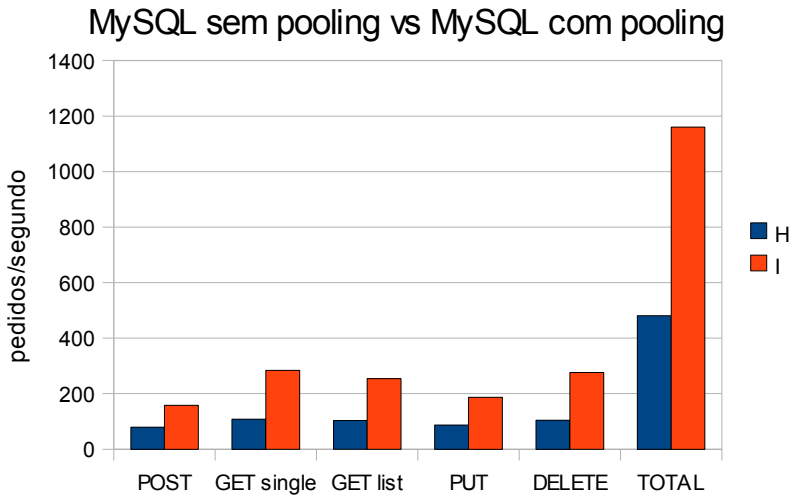

Figura 65: Comparação de resultados das plataformas H e I

A diferença de performance é notória, e no geral, o pooling garante um aumento de performance de 141%.

Para facilitar o processo de desenvolvimento de aplicações, o ORM tornou-se uma solução muito popular, abstraindo vários pormenores do trabalho com bases de dados relacionais. Qual o custo dessa abstracção ?

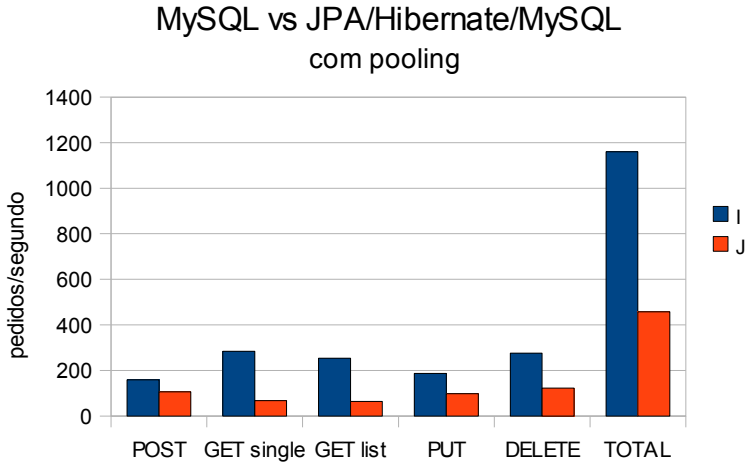

Figura 66: Comparação de resultados das plataformas I e J

Na Figura 66, a plataforma I corresponde ao Tomcat com MySQL e pooling de ligações, enquanto que na plataforma J, a situação e idêntica, com a adição de ORM via JPA+Hibernate.

O impacto do uso de ORM neste caso é grande, sendo de 154%. Ou seja, o ORM custou mais do que os ganhos de se ter usado o pooling. Sem dúvida que o ORM é uma ferramenta essencial actualmente, sobretudo para projectos com modelos de dados mais complexos, mas há que ter consciência do grande custo associado.

Faltando apenas comparar a base de dados Neodatis ODB contra a performance do MySQL. É de referir que em todas as situações, a Neodatis ODB não está a usar pooling de ligações, pois ainda não é suportado pela versão actual.

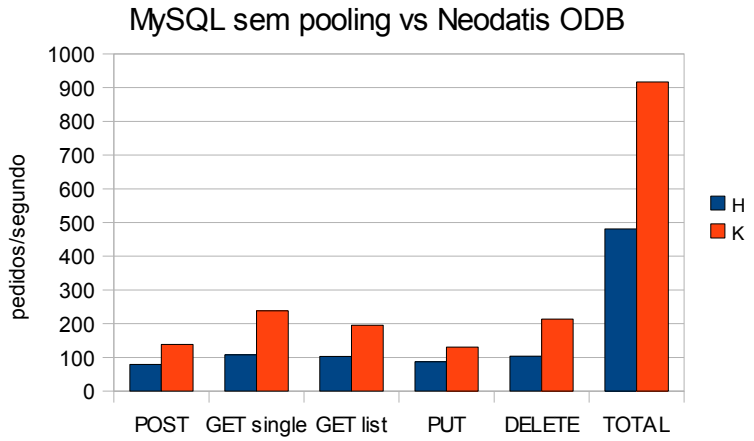

Figura 67: Comparação de resultados das plataformas H e K

Em primeiro lugar é realizada a comparação da Neodatis ODB contra o MySQL sem pooling de modo a obter a diferença de performance na mesma situação, conforme ilustrado na Figura 67. São utilizadas a plataforma a plataforma H, usando o MySQL, e a K usando a Neodatis ODB. Nas mesmas condições sem pooling de ligações, a Neodatis ODB é 90% mais rápida que o MySQL. Também podem ser questionados como serão os resultados no caso do MySQL com pooling? Usamos novamente a plataforma K pela Neodatis ODB contra a plataforma I, que usa o MySQL

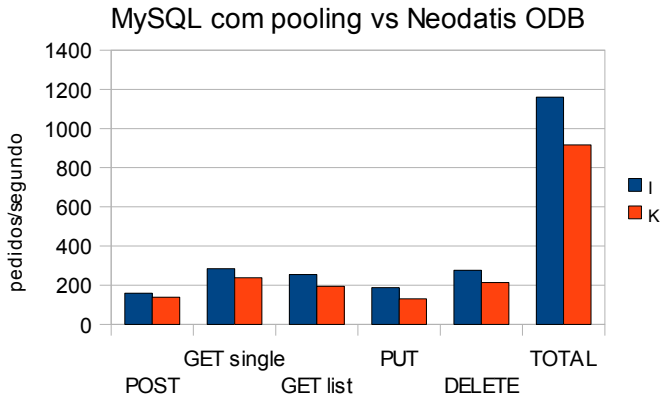

Figura 68: Comparação de resultados das plataformas I e K

com pooling.(Figura 68).

O pooling aumenta a performance de tal forma que elimina a diferença anterior e ainda coloca o MySQL a ser 26% mais rápido do que a Neodatis ODB.
Assim que a base de dados Neodatis ODB implemente o suporte ao pooling, será interessante efectuar testes adicionais para verificar se esta diferença consegue ser reduzida ou ultrapassada.

Finalmente temos um teste final entre o ORM via JPA+Hibernate contra a Neodatis ODB. É a comparação esperada entre uma solução inteiramente orientada a objectos – Neodatis ODB – contra uma que efectua o mapeamento entre objectos e bases de dados relacionais.

À partida, é de esperar que o ORM será mais lento, pois tem de lidar com bases de dados relacionais, uma tecnologia que nunca foi desenvolvida com o paradigma de orientação por objectos em mente.Para verificar esta hipótese, comparamos a plataforma K que usa a Neodatis ODB, contra a plataform J com ORM via JPA/HIbernate/MySQL com pooling de ligações. (Figura 69)

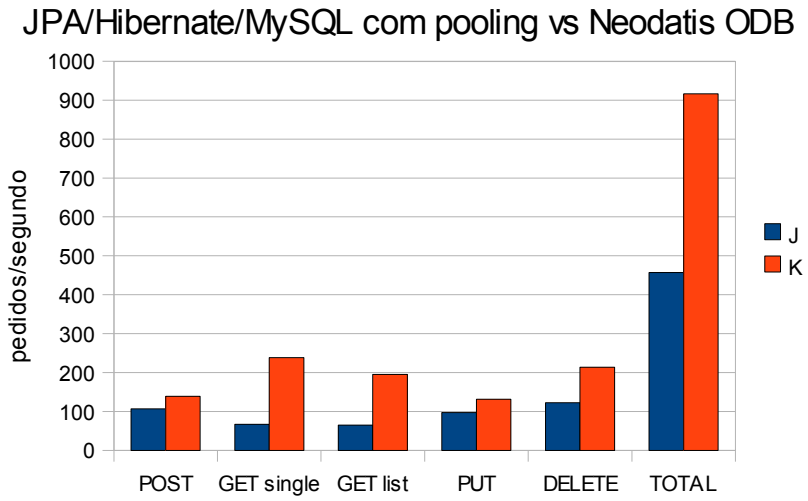

Figura 69: Comparação de resultados das plataformas J e K

De facto, confirma-se o impacto negativo que o ORM tem na performance. A base de dados Neodatis é 100% mais rápida que a solução com ORM.

Este três resultados permitem chegar à conclusão de que a Neodatis ODB não ganha face a SQL escrito manualmente e com pooling, mas ultrapassa em muito a performance do ORM.

A Neodatis ODB é um projecto recente, com muitos aspectos a melhorar, mas que mostra já um grande potencial, podendo tornar-se uma séria alternativa a soluções já existentes.