

SOA aplicado

Integrando com web services e além



Casa do
Código

ALEXANDRE SAUDATE

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos mecânicos, gravação ou quaisquer outros.

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil



Casa do Código
Livros para o programador

**Uma editora de livros técnicos
feita por desenvolvedores
para desenvolvedores.**



**Inscreva-se em nossa newsletter e
receba novidades e lançamentos**

www.casadocodigo.com.br/newsletter



Curta nossa fanpage no Facebook

www.facebook.com/casadocodigo



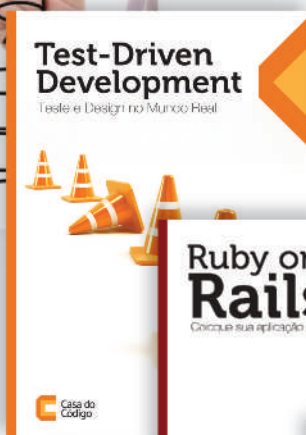
**Caelum:
Cursos de TI presenciais e online**

www.caelum.com.br



Dê seu feedback sobre o livro. Escreva para contato@casadocodigo.com.br

Já conhece os nossos títulos?



E muito mais em:
www.casadocodigo.com.br



RAFAEL STEIL

*“Agradeço à minha família, minha namorada e a todos os que acreditaram em mim
— mesmo quando eu mesmo não acreditei.”*
– Alexandre Saudate

Prefácio

SOA passa por integração, mas integração não é SOA

Por que me apaixonei pelo estilo de desenvolvimento baseado em SOA?

Há muitos anos, quando desenvolvia puramente para uma plataforma específica, fazer meu software se integrar a outro era praticamente um *Jihad*.

Essa necessidade começou a ficar mais forte quando precisava expor funcionalidades a clientes ou parceiros de negócio da companhia para qual prestava serviço, pois não poderia obrigar os seus desenvolvedores a utilizar nossa plataforma e linguagem.

Imaginem uma *startup* como PayPal criando seu leque de APIs para parceiros de negócio poderem reutilizar suas funções básicas de *gateway* de pagamento, detecção de padrões de fraude etc. Faz sentido que as mesmas só sejam disponíveis a desenvolvedores Delphi? Provavelmente essa *startup* teria falido.

Diversas técnicas de integração começaram então a serem desenvolvidas, desde troca de arquivos com um *layout* específico, base do EDI, *Electronic Data Interchange*, a “tabelinhas intermediárias”, utilizadas erroneamente por diversas equipes. Mas voltando ao exemplo anterior, como faria uma chamada *online* a uma API? No caso de um *e-commerce*, atualmente preciso fazer o pagamento no mesmo instante, os compradores não querem mais esperar dias para saberem se sua compra foi ou não processada.

Cada dia mais, todos os processos começaram a se tornar *online* e acredito que esse impulso veio da concorrência de mercado entre as empresas, ofertando cada vez mais comodidade aos seus clientes.

Comprar um celular numa loja de uma operadora e esperar 3 dias para ser liberada a utilização é uma experiência pouco feliz. Se a concorrência começar a antecipar para 1 hora ou imediatamente, quem estiver fora desse contexto perderá uma massa considerável de novos clientes.

Então, fazer seu *software* responder *online* a muitas plataformas era uma tarefa árdua mas necessária!

Tecnicamente precisávamos lidar com questões como interface comum de comunicação a todas as linguagens e plataformas, protocolo para *marshall* e *unmarshall* dos objetos serializados, transação entre plataformas, segurança etc. Eram tantos detalhes para pensar — então me deparei com uma especificação regida por um comitê que vinha evoluindo desde 1991, e clareou diversas questões: *Common Object Request Broker* — CORBA.

Expor uma API online em CORBA, seria a solução para que diversos outros programadores pudessem usar minha função básica de *gateway* de pagamento, naquele exato instante, levando satisfação ao cliente final. Assim ele já teria a solução de seu pedido processado, já com resposta sobre seu crédito de forma imediata.

Então vem a pergunta, o que é SOA afinal? SOA, como chamamos aqui na SOA|EXPERT, é um acrônimo para **Smart Oriented APIs**, que nada mais é que produzir seu *software* com APIs ricas, "**inteligentes**" e poderosamente **reutilizáveis**, sendo que qualquer plataforma e linguagem possa fazer uso da sua funcionalidade. Isso é lindo!

Como tudo em tecnologia, sobretudo em desenvolvimento de *software*, as técnicas tendem a ficar mais fáceis de se implementarem. Ao longo dos anos, vamos tendo entendimento sobre como se fazer melhor e mais fácil, cortando uma série de burocracias e abstraindo do desenvolvedor complexidades demasiadas.

Por isso SOA não tem a ver com uma tecnologia específica. SOA é o conceito de modelar sua API inteligente, rica e poderosa para reutilização.

A tecnologia que utilizamos atualmente vem evoluindo rapidamente. Começamos com CORBA lá trás, e hoje utilizamos *WebServices*, o estilo arquitetural Rest, alguns outros modelos como AtomPub, *WebSockets* entre várias combinações possíveis numa arquitetura. (CQRS, OData etc.)

Começamos a definir *patterns* para os problemas mais comuns e esses vão evoluindo à medida que nosso entendimento fica mais claro, fazendo com que esse processo de desenvolvimento de APIs inteligentes seja mais simples e intuitivo para o desenvolvedor.

Atualmente, o mundo está totalmente interligado, diversos aplicativos nas redes sociais estão entrelaçados, fazendo uso das suas funções básicas de comunicação, por exemplo: Twitter. Aliás, hoje seus equipamentos *twittam*, seu Nike plus exibe seus dados de corrida nas redes sociais.

Startups utilizam mapas geográficos de fornecedores como Google Maps e não

importam sua linguagem de programação ou plataforma. Todos podem se valer do esforço da equipe do Google e criar um FourSquare à partir dessa poderosa API de mapas.

Gosto de pensar que SOA é a socialização da sua API: torná-la social, democrática e com infinitas possibilidades de utilização, dando liberdade à criatividade dos desenvolvedores.

Contudo, o que aconteceria se o Google Maps saísse do ar? Quantas *startups* e até mesmo aplicativos internos do Google seriam impactados?

Desenvolver esse tipo de API exige bastante responsabilidade do desenvolvedor, tanto na qualidade da mesma, quanto na manutenção do seu estado.

As tecnologias associadas a esse modelo de desenvolvimento, como ESB, BPEL, SCA, CEP, BRMS, BPM, BAM, Governança, entre outras, são resultado do estudo das mais amplas necessidades e servem para gerenciar as problemáticas e o ciclo de vida de uma **Smart API**.

Esse livro começará com os aspectos mais importantes, definição e modelagem da sua API utilizando dois modelos de desenvolvimento: *Web Services* Clássico e Restful, passando por integração com o framework *Enterprise Service Bus*.

Há ainda um conjunto grande de tecnologias que serão abordadas em futuros volumes, pois apesar de fazerem parte do ecossistema, são bem específicas e amplas para serem abordadas em um único livro.

Caso estejam um pouco curiosos sobre as siglas citadas, farei uma rápida explicação para contextualizá-los e colocá-los num *roadmap* de estudos:

- Como pego uma função de um sistema legado, *mainframe* ou ERP e a deixo simples para outros desenvolvedores acessarem? Infelizmente não podemos reescrever todos os *softwares* existentes de uma companhia, então vamos necessitar de um **framework de integração**, que aplique as melhores práticas: *Enterprise Integration Patterns*. O **ESB** é o *framework* dentro desse universo, que irá fazer a implementação desses *patterns* e tornar simples para o desenvolvedor a utilização de funções legadas.

- Como controlo um contexto transacional e dou *rollback* entre tecnologias distintas? Se estivéssemos somente em Java, poderíamos acionar o JTA. Contudo, como propago o *rollback* para .NET ou outras plataformas? A DSL **BPEL** resolve essa e muitas outras questões.

- Como construo novas regras de negócio, fazendo junção (*Mashups*) entre APIs de fornecedores externos? Essa é uma das infinitas possibilidades com um motor de regras: **BRMS**.

- Como controlo um processo de *workflow* entre APIs mescladas à intervenção humana? O BPM tornará simples essa tarefa e você não precisará perder noites pensando em máquina de estados, compensação etc.

Outras tecnologias ou conceitos associados ao universo SOA nascem das amplas possibilidades de termos nosso ecossistema aberto, como o **BAM** (*Business Activity Monitoring*), com o qual fica fácil monitorarmos o negócio do cliente em tempo real, já que temos APIs expostas e podemos colocar sensores nas mesmas para coletar dados.

Há tantas possibilidades dentro de SOA, até mesmo como construir uma aplicação distribuída para melhoria de escalabilidade e *performance*, já que na prática você está montando um sistema distribuído, e fazer melhor uso de *cloud computing*, pois esse estilo está intimamente ligado ao mesmo. Seria lidar com sistemas em *real time* com *stream* de eventos etc.

Contudo, é importante lembrar que apesar de todas as tecnologias citadas ao redor, SOA é na essência a modelagem de APIs e isso precisa estar claro a vocês.

Uma alusão que sempre passo em sala de aula: “SOA é uma API que brilha no escuro”. A brincadeira didática serve para enaltecer que é mais que uma simples integração. Integração por si só é levar informação de um ponto ao outro. Criar uma **API que brilha** é modelar pensando em diversas reutilizações, e plataformas e devices como TV digital, *mobile* etc.

O livro do meu amigo Alexandre Saudate é um ótimo ponto de início, principalmente aos desenvolvedores Java, pois tirou toda a burocracia dos livros tradicionais e, de maneira pragmática, mostra como se implementar seu sistema orientado a APIs sem rodeios.

Espero de verdade que você também se apaixone por esse estilo de desenvolvimento de *software*, pois não faz sentido mais desenvolvermos presos a uma única plataforma como se fosse *software* de caixinha, sem comunicação com o mundo lá fora.

Aos novos desenvolvedores, também gostaria de convidá-los a participar da nossa comunidade SOACLOUD - <http://www.soacloud.com.br/>, onde paulatinamente você poderá testar seus conhecimentos, expor dúvidas e novos pensamentos adquiridos através desse livro.

Boa leitura,

Felipe Oliveira

Fundador da SOA|EXPERT.

Introdução

Comecei a programar de forma relativamente tardia, por volta dos quinze anos. Aprendi num curso técnico em informática a linguagem Visual Basic 6. Quatro anos depois, passei a trabalhar com programação profissionalmente, com a mesma linguagem que aprendi no meu curso técnico. Desde então, notei que os sistemas que desenvolvia eram monolíticos, e outros sistemas só podiam interagir com estes através de mecanismos arcaicos, como escrita de arquivos em pastas específicas ou diretamente pelo banco de dados.

Em fevereiro de 2008, eu comecei a trabalhar em uma pequena consultoria de São Paulo. Esta consultoria estava começando a encerrar o foco de suas atividades em JEE, e readaptando este foco para SOA — Arquitetura Orientada a Serviços.

Você pode se perguntar: qual a diferença?

Para leigos, não existe muita. Porém, à medida que você se envolve com este tipo de arquitetura, percebe que muitas decisões devem ser tomadas por esta simples mudança de foco. Ao expor serviços como foco de suas aplicações, muitas decisões devem ser tomadas: o que expor como serviços? Como expor? E, mais importante, por quê?

O foco deste livro é responder, principalmente, a **como** expor *web services*. A proposta é fornecer a você, leitor, os insumos para que possa prosseguir suas próprias investigações a respeito de SOA e descobrir a resposta para outras questões.

Do que se trata o livro?

Este livro está dividido em nove capítulos.

O capítulo um apresenta a motivação para uso de *web services*. Também mostra como realizar a criação de um *web service* simples em Java, assim como o consumo do mesmo, tanto em Java quanto em uma ferramenta de mercado chamada SoapUI.

O capítulo dois mostra detalhes dos mecanismos envolvidos na comunicação realizada no capítulo um. Ele detalha a estrutura do WSDL, SOAP e XML Schemas.

O capítulo três mostra como adaptar os documentos apresentados no capítulo dois para suas necessidades. Ele mostra com mais detalhes as APIs Java que regem o funcionamento destes mecanismos, ou seja, JAX-WS e JAXB.

O capítulo quatro mostra como tirar proveito de servidores de aplicação para realizar a implantação de seus *web services*. Além disso, mostra como aliar a tecnologia envolvida em *web services* com conceitos de JEE, como *Enterprise JavaBeans*.

O capítulo cinco mostra uma abordagem diferenciada para criação e consumo de *web services* — os serviços REST.

No capítulo seis, você irá aprender a instalar e configurar mecanismos de segurança nos serviços, tanto clássicos (WS-*) quanto REST. Você verá, também, o primeiro modelo de integração entre um serviço WS-* e REST.

No capítulo sete, você verá alguns dos *design patterns* mais importantes de SOA: modelo canônico, desenvolvimento *contract-first* e serviços assíncronos (com WS-Addressing).

No capítulo oito, você verá como instalar, configurar e utilizar o Oracle Service Bus, um dos principais *Enterprise Service Bus* da atualidade.

No capítulo nove, você verá como instalar, configurar e utilizar o Oracle SOA Suite, estando habilitado, então, a utilizar o Oracle BPEL.

Recursos do livro

Todo o código fonte deste livro está disponível no meu github: <https://github.com/alesaudate/soa>. Caso você tenha dúvidas adicionais/ sugestões/ reclamações sobre o livro, você pode postá-las num grupo de discussão específico, localizado em <https://groups.google.com/forum/?fromgroups=#!forum/soa-aplicado>.

Boa leitura!

Sumário

1	Começando um projeto SOA	1
1.1	Coisas inesperadas acontecem!	1
1.2	Formatos de arquivos: CSV	2
1.3	Discussão sobre formatos: linguagens de marcação	3
1.4	Qual camada de transporte utilizar?	4
1.5	E agora, como testar um serviço?	8
1.6	Crie o primeiro cliente Java	12
1.7	OK, e o que um web service tem a ver com SOA?	14
1.8	Sumário	16
2	Entendendo o fluxo de dados	17
2.1	Como funciona a estrutura de namespaces do XML	17
2.2	Conheça o funcionamento do SOAP	19
2.3	Entenda o enorme WSDL	19
2.4	A definição dos tipos e a seção types	21
2.5	A necessidade da seção messages	24
2.6	A seção portType	25
2.7	A diferença entre os WSDLs abstratos e concretos	26
2.8	A seção binding	27
2.9	Definição dos endereços com a seção service	29
2.10	Finalmente, o que aconteceu?	30
2.11	Customize a estrutura do XML com o JAXB	31
2.12	Resolva métodos de nomes iguais e adição de parâmetros	35
2.13	Envie a requisição pelo cliente	41
2.14	Sumário	42

3	Novos desafios e os ajustes finos para controles de exceções e adaptação de dados	45
3.1	Customize o sistema de parâmetros	46
3.2	Conheça o sistema de lançamento de exceções	49
3.3	Customize a sessão de detalhes	51
3.4	Customize ainda mais o lançamento de exceções	53
3.5	Embarcando mais a fundo no JAXB	55
3.6	Tire proveito de adaptadores	57
3.7	Trabalhe com JAXB usando herança	60
3.8	Trabalhe com enums	63
3.9	Modele suas classes com comportamentos de negócio e mantenha-as mapeadas com JAXB	66
3.10	Sumário	71
4	Embarcando no Enterprise - Application Servers	73
4.1	Como usar um Servlet Container - Jetty	74
4.2	Introdução a EJBs	77
4.3	Habilitando persistência e transacionalidade	80
4.4	Um novo sistema	83
4.5	Sumário	86
5	Desenvolva aplicações para a web com REST	89
5.1	O que é REST?	90
5.2	Entenda o HTTP	92
5.3	URLs para recursos	96
5.4	Métodos HTTP e uso de MIME types	98
5.5	Utilização efetiva de headers HTTP	100
5.6	Utilização de códigos de status	102
5.7	Utilização de hipermídia como motor de estado da aplicação	106
5.8	Como desenvolver serviços REST	107
5.9	Mapeamento avançado: tratando imagens	114
5.10	Incluindo links HATEOAS	119
5.11	Testando tudo	126
5.12	Programando clientes	130
5.13	Sumário	135

6	Segurança em web services	137
6.1	Ataques: Man-in-the-middle e eavesdropping	138
6.2	Proteção contra interceptação com HTTPS	140
6.3	Usando mecanismos de autenticação HTTP	143
6.4	Habilitando segurança em aplicações REST	146
6.5	Conhecendo WS-Security	151
6.6	Ajustes de infra-estrutura	161
6.7	O callback de verificação da senha	165
6.8	A atualização do serviço de usuários	166
6.9	A implementação do método de busca	171
6.10	Realize a comunicação entre os dois sistemas	173
6.11	Testes com SoapUI	180
6.12	Crie o cliente seguro	184
6.13	Sumário	190
7	Design Patterns e SOA	191
7.1	Integração versus SOA	191
7.2	O Modelo Canônico	192
7.3	Desenvolvimento contract-first	199
7.4	Serviços assíncronos com WS-Addressing	210
7.5	Sumário	220
8	Flexibilizando sua aplicação com um ESB	221
8.1	Como instalar o Oracle WebLogic e o OEPE	222
8.2	A instalação do OSB	228
8.3	Configuração do OSB	232
8.4	Conceitos do OSB	243
8.5	Crie uma rota no OSB	244
8.6	Sumário	248
9	Coordene serviços com BPEL	249
9.1	Conheça orquestração e coreografia	250
9.2	instale o Oracle SOA Suite	250
9.3	Instale o JDeveloper	260
9.4	Introdução a BPEL	264
9.5	Sumário	274

10 Conclusão	275
---------------------	------------

Bibliografia	277
---------------------	------------

Versão: 16.2.5

CAPÍTULO 1

Começando um projeto SOA

“Em todas as coisas, o sucesso depende de preparação prévia.”

– Confúcio

1.1 COISAS INESPERADAS ACONTECEM!

Você foi contratado como arquiteto de um novo projeto: desenvolver um *e-commerce*, chamado knight.com, para uma editora de livros. Como arquiteto, tudo o que você sabe inicialmente é que deve ser um sistema de médio porte. Você decide começar o desenvolvimento do projeto pela parte que lhe parece uma unanimidade: o controle de estoque. Para isso, cria um projeto baseado em Java EE chamado `knight-estoque.war`. Este projeto possui a seguinte estrutura:

```
* knight-estoque.war
  * vários pacotes distintos...
  * com.knight.estoque.modelos;
  * com.knight.estoque.daos;
  * com.knight.estoque.controllers;
```

Depois de vários meses de desenvolvimento, o módulo é lançado. Ele parece ser funcional e pronto para ser utilizado em outras aplicações, mas surge um problema: por questões de negócios, a gerência decide que, para que o sistema faça sucesso, deve ser possível interagir com o mesmo.

Tal requisito é absolutamente agnóstico em termos de linguagem de programação: deve ser possível interagir com o sistema através de celulares, aplicações escritas em Java, C#, PHP e diversas outras (até mesmo por APIs de redes sociais). Você não estava preparado para isso, afinal, você só estava acostumado com o desenvolvimento de sistemas de uma única linguagem, que não precisava interagir com nada externo.

1.2 FORMATOS DE ARQUIVOS: CSV

O seu primeiro pensamento é realizar a integração através de trocas de arquivos, por exemplo, os famosos arquivos CSV (*Comma-Separated Values*). Para avaliar o formato, você escolhe a principal classe do sistema, `Livro`:

```
package com.knight.estoque.modelos;

import java.util.List;

public class Livro {

    private String nome;
    private List<String> autores;
    private String editora;
    private Integer anoDePublicacao;
    private String resumo;

    // getters e setters...
}
```

Todos os atributos da entidade livro são perfeitamente elegíveis para serem incluídos em uma única linha do CSV, exceto a listagem de autores: por ser uma lista (ou seja, não ter uma definição exata do número de autores), você decide que a melhor estratégia para lidar com ela é colocar em uma segunda linha. Assim, o formato fica:

```
[nome];[editora];[ano de publicação];[resumo];
[autores (separados por ponto-e-vírgula)]
```

No entanto, o seguinte pensamento vem à sua mente: e se uma terceira linha (para comportar uma nova listagem) precisar ser adicionada? E se mais um campo for adicionado? E, pior: e se um campo precisar ser removido? Por exemplo, suponha que o seguinte arquivo CSV é gerado:

```
Guia do Programador;Casa do Código;2012;Vá do "nunca programei" ...;  
Paulo Silveira;Adriano Almeida;  
Ruby on Rails;Casa do Código;2012;Crie rapidamente aplicações web;  
Vinícius Baggio Fuentes;
```

No caso acima, se o campo `editora`, por exemplo, tiver que ser removido, todas as aplicações clientes devem ser alteradas para reconhecer o formato novo. Caso as alterações não sejam reprogramadas, duas coisas podem acontecer: ou as aplicações passam a identificar o ano como `editora`, o resumo como ano (!) e achar que não existe resumo algum, ou elas podem simplesmente detectar o problema e passar a lançar exceções.

Ambos os casos são indesejáveis, assim como notificar todas as aplicações clientes pode ser impraticável, dependendo do número de clientes que existirem.

1.3 DISCUSSÃO SOBRE FORMATOS: LINGUAGENS DE MARCAÇÃO

Você nota, então, que esse é um caso de uso ideal para uma linguagem de marcação qualquer, já que, em geral, as linguagens de marcação possuem boas definições do que cada campo representa, não originam problemas caso algum campo seja adicionado e minimizam impactos caso haja uma remoção de campos.

Você decide então avaliar o XML (*eXtensible Markup Language*), porque é uma das linguagens de marcação mais antigas e mais bem aceitas em termos de interoperabilidade entre linguagens, visto que pertence à mesma família do HTML (que é o formato utilizado por todas as páginas da Web).

Por exemplo, a definição do exemplo de livros, mostrada na seção 1.2, ficaria assim em XML:

```
<livros>  
  <livro>  
    <anoDePublicacao>2012</anoDePublicacao>  
    <autores>  
      <autor>Paulo Silveira</autor>
```

```
<autor>Adriano Almeida</autor>
</autores>
<editora>Casa do Código</editora>
<nome>Guia do Programador</nome>
<resumo>Vá do "nunca programei" ...</resumo>
</livro>
<livro>
  <anoDePublicacao>2012</anoDePublicacao>
  <autores>
    <autor>Vinícius Baggio Fuentes</autor>
  </autores>
  <editora>Casa do Código</editora>
  <nome>Ruby on Rails</nome>
  <resumo>Crie rapidamente aplicações web</resumo>
</livro>
</livros>
```

Além de tudo isso, você nota que, não importa a ordem em que os campos apareçam, eles continuam autoidentificados, tendo inclusive o bônus de ser facilmente legível por seres humanos. Desta forma, você decide utilizar XML. Agora, só precisa definir a maneira de enviar esse XML para outros sistemas e recebê-los também.

1.4 QUAL CAMADA DE TRANSPORTE UTILIZAR?

A opção mais óbvia que vem à sua mente para realizar essa transferência seria uma leitura direta de disco, talvez com utilização de NAS (*Network-Attached Storage*, que basicamente consiste na ideia de ter um servidor dedicado ao armazenamento de rede) ou mesmo SAN (*Storage Area Network*, que consiste de utilizar uma rede completa de armazenamento, com gerenciamento via software, segurança intrínseca etc.). Porém, diversos impedimentos são apontados: pelo fato de haver vários clientes, podem existir conflitos de segurança, sobrescrita de arquivos etc. Além disso, existem os problemas de que pessoas capacitadas em gerenciamento de recursos dessa natureza devem ser empregadas e também a (quase certa) burocracia que pode existir ao redor do gerenciamento desses recursos.

Com tantos impedimentos circulando a leitura direta, a opção que logo vem à cabeça, nesse aspecto, é o FTP. Além de ser um recurso muito mais barato por vários pontos de vista, também tem formas de segurança internas. No entanto, o gerenciamento de segurança não é muito abrangente, sem contar que pode não apresentar tanta interoperabilidade em relação a várias linguagens de programação. Além disso,

de qualquer maneira, você precisa de um sistema eficiente na questão de enviar e receber dados de forma *online*, não apenas escrita de dados (de qualquer maneira que ela seja).

Deste modo, a melhor solução que lhe vem à mente é enviar dados por meio de *sockets*. Você faz alguns testes e lhe parece bom. No entanto, com investigações mais aprofundadas, você começa a perceber algumas falhas. Primeiro, a falta um protocolo de transporte dificulta a passagem de quaisquer metadados, além de dificultar a encriptação de dados. Além disso, ainda é altamente inseguro fazer tráfego de dados através desse tipo de comunicação, pois *hackers* podem aproveitar essa brecha para atacar a aplicação — motivo pelo qual nenhum analista de segurança iria permitir esse tráfego.

O que fazer, então? A melhor solução para contornar todos estes problemas seria fazer uso de HTTP (*HyperText Transfer Protocol*). Este protocolo implementa passagem de metadados através de cabeçalhos, pode transportar os dados de maneira segura (através da sobreposição de SSL — *Secure Sockets Layer*), sem contar que é praticamente onipresente (todas as páginas de internet são trafegadas utilizando esse protocolo), fazendo-o ter grande aceitação nas diferentes linguagens de programação.

Você decide pesquisar mais a respeito da combinação XML/HTTP quando você acha um artigo sobre Web Services. A ideia é bem simples: trafegar XML utilizando HTTP como protocolo de transporte. Além disso, acrescenta um contrato de serviço, que mostra o que é esperado como entrada/saída do provedor de dados.

WEB SERVICES E O PROTOCOLO DE TRANSPORTE

Os Web Services tradicionais não estão restritos à utilização apenas de HTTP. No entanto, pela grande aceitação desse protocolo, a grande maioria dos serviços implementados em empresas são exclusivos para HTTP, e não existem muitos provedores de Web Services em outros tipos de protocolo.

A título de curiosidade, a linguagem Java provê mecanismos para transporte utilizando protocolos RMI, JMS, entre outros. Para mais informações, veja o projeto Apache WSIF em <http://ws.apache.org/wsif/>.

Você decide fazer um teste sobre uma das funcionalidades do módulo de esto-

ques, chamada “obter catálogo”. Ela nada mais faz do que obter a listagem de livros disponíveis para venda. O código inicial do serviço é o seguinte:

```
package com.knight.estoque.servicos;

// imports omitidos

public class ListagemLivros {

    public List<Livro> listarLivros() {
        LivroDAO livroDAO = obterDAO();
        return livroDAO.listarLivros();
    }

    // outros métodos
}
```

Para transformar esta classe em web service, basta anotá-la com `javax.jws.WebService`; e para expor este web service, ou seja, deixá-lo pronto para uso em outras aplicações, basta utilizar a classe `javax.xml.ws.Endpoint`. Não se preocupe, tudo isso está incluso na JDK, a partir do Java 6.

JAX-WS E A COMPATIBILIDADE DE BIBLIOTECAS

Nas versões anteriores ao Java 6, o JAX-WS e o JAXB precisavam ser baixados e empacotados junto da aplicação. Com vários problemas de incompatibilidade entre várias implementações de ambos, acabou-se decidindo que eles deveriam estar disponíveis diretamente na JRE do Java — o que causa alguns problemas em aplicações projetadas para rodar em Java 5 e que na verdade, executam em Java 6. Tenha sempre muito cuidado com esse cenário.

A classe anotada, e já com um método que inicializa o serviço, fica assim:

```
package com.knight.estoque.servicos;

// imports omitidos
```

```
@WebService
public class ListagemLivros {

    public List<Livro> listarLivros() {
        LivroDAO livroDAO = obterDAO();
        return livroDAO.listarLivros();
    }

    // Outros métodos...

    public static void main(String[] args) {
        Endpoint.publish("http://localhost:8080/livros",
                           new ListagemLivros());
        System.out.println("Serviço inicializado!");
    }
}
```

Além disso, você também precisa identificar as classes que irão trafegar pela rede perante o sistema de *parsing* de dados. Para isto, é necessário colocar um arquivo chamado `jaxb.index` que irá referenciar estas classes a partir do pacote onde ele está. Por exemplo, se tivermos uma classe Java com nome completo `com.knight.estoque.modelos.Livro`, e colocarmos o arquivo `jaxb.index` no pacote `com.knight.estoque.modelos`, o conteúdo do mesmo será somente a linha `Livro`.

Então, no arquivo `jaxb.index`, podemos ter:

`Livro`

O código completo ficaria assim:

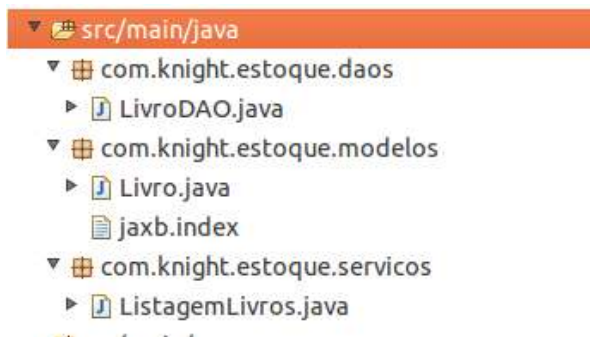


Figura 1.1: Primeiras classes do projeto

1.5 É AGORA, COMO TESTAR UM SERVIÇO?

Ao rodar este código, você percebe um detalhe: não faz a menor ideia de como interagir com o serviço! Você lê mais a respeito e descobre que um dos elementos de um web service é um contrato, chamado **WSDL**.

Seu serviço está no ar. Você deu a ele o endereço <http://localhost:8080/livros>, e, por conseguinte, você consegue o contrato dele em <http://localhost:8080/livros?wsdl>.

COMO ENCONTRAR O WSDL

Em geral, o WSDL de serviços feitos em Java são encontrados adicionando `?wsdl` ao final do endereço do serviço. Essa não é uma regra formal; algumas linguagens disponibilizam esse contrato com `.wsdl` ao final do endereço do serviço e outras podem nem sequer disponibilizá-lo pela rede.

Acessando o contrato, você encontra um documento que parece extremamente complexo, e praticamente impossível de entender, ao dar uma rápida olhada. Você procura um pouco mais na internet e acha uma ferramenta chamada SoapUI, disponível em <http://soapui.org/>, que é usada para realizar testes de web services. Após o download e instalação da ferramenta, você vê uma tela como a seguinte:

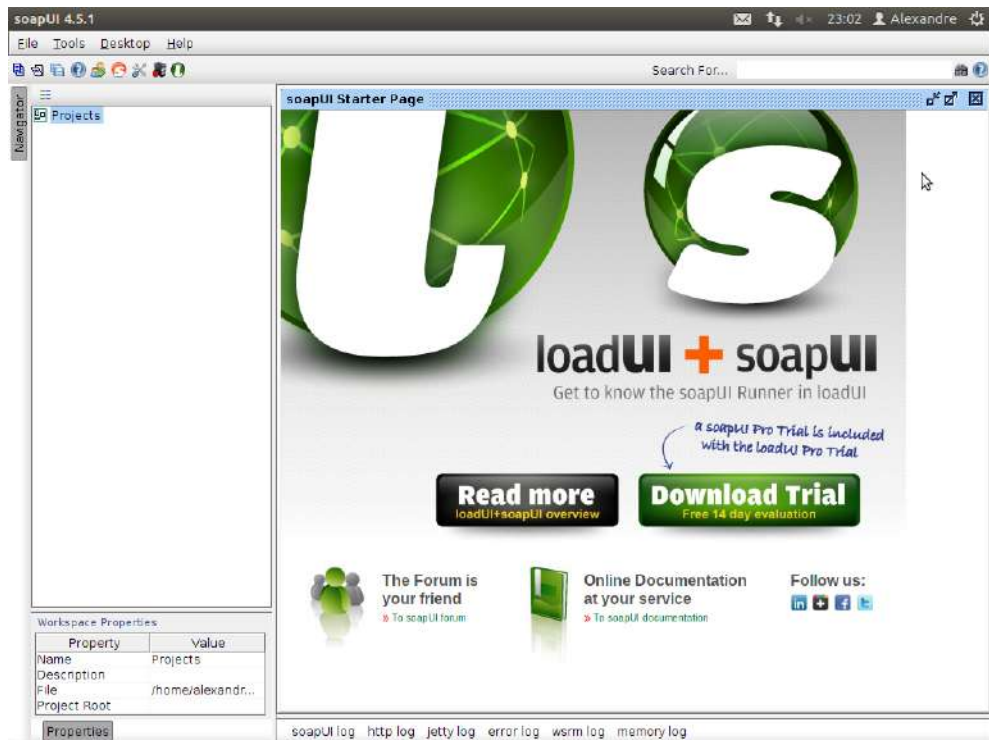


Figura 1.2: Tela inicial do SoapUI

No SoapUI, é possível colocar o WSDL como entrada e realizar testes sobre ele, assim como mostram as figuras 1.3 e 1.4:

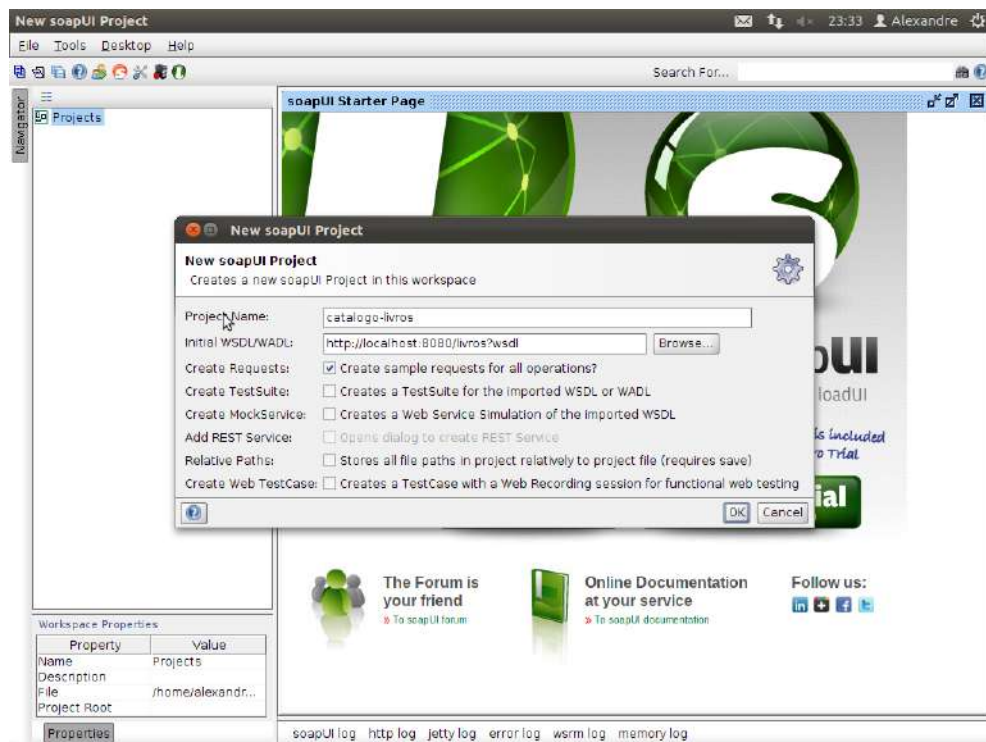


Figura 1.3: Criando um projeto com o SoapUI

Fazendo, então, a inclusão do projeto, você repara que o SoapUI criou a seguinte estrutura:

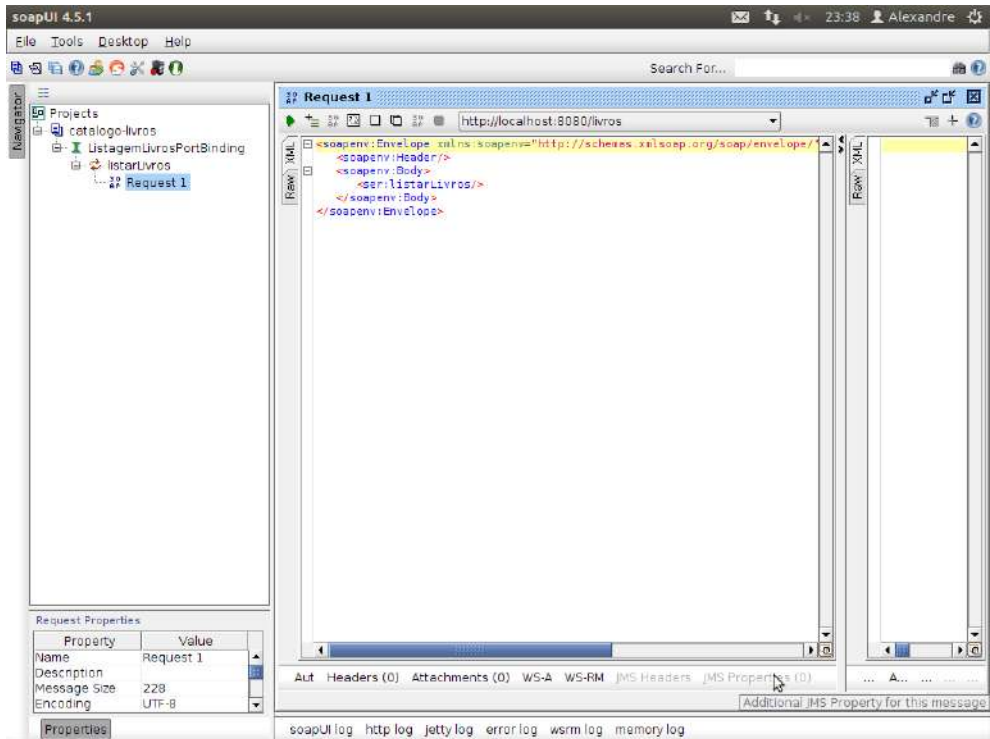


Figura 1.4: Começando os testes com o SoapUI

Ao clicar duas vezes sobre `Request 1`, você se depara com um XML formatado: o chamado **envelope SOAP**.

SOAP

SOAP é uma sigla para *Simple Object Access Protocol*, ou Protocolo Simples de Acesso a Objetos, numa tradução livre. Mais à frente, você verá em detalhes este protocolo e como manipulá-lo corretamente.

O envelope SOAP contém todos os dados necessários para que nosso serviço interprete corretamente os dados que estamos fornecendo. Respeitando o propósito da interoperabilidade, ele é inteiramente baseado em XML, que deve ser “traduzido” em termos da linguagem que o recebe na ponta. No nosso caso, ele não possui quaisquer parâmetros ainda, e para nós, basta submeter a requisição do jeito como está.

Para isso, basta apertar um botão verde, em formato de “play”, na parte superior do SoapUI e ao lado esquerdo do endereço do serviço.

Quando você o faz, obtém o seguinte retorno:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:listarLivrosResponse
      xmlns:ns2="http://servicos.estoque.knight.com/">
      <return>
        <anoDePublicacao>2012</anoDePublicacao>
        <autores>Paulo Silveira</autores>
        <autores>Adriano Almeida</autores>
        <editora>Casa do Código</editora>
        <nome>Guia do Programador</nome>
        <resumo>Vá do "nunca programei" ...</resumo>
      </return>
      <return>
        <anoDePublicacao>2012</anoDePublicacao>
        <autores>Vinícius Baggio Fuentes</autores>
        <editora>Casa do Código</editora>
        <nome>Ruby on Rails</nome>
        <resumo>Crie rapidamente aplicações web</resumo>
      </return>
    </ns2:listarLivrosResponse>
  </S:Body>
</S:Envelope>
```

1.6 CRIE O PRIMEIRO CLIENTE JAVA

Até aqui, o seu serviço está plenamente operacional: é capaz de enviar um XML contendo um nome de uma operação (ou seja, a descrição do que você quer que seja feito), e quando envia esse XML, um código é executado no servidor, trazendo como resposta um XML contendo dados. Mas você não quer parar por aí: **você quer ver código Java fazendo isso**, da melhor maneira.

Existem vários meios de criar clientes de web services — quase todos são “tunados” por meio de IDEs — mas o jeito mais fácil continua sendo, sem dúvidas, pelo programa *wsimport*, localizado na pasta `bin` da JDK. Com o seu serviço sendo executado, você pode colocar o código do cliente do serviço na pasta `generated` (relativa a partir da pasta atual) com o seguinte comando:

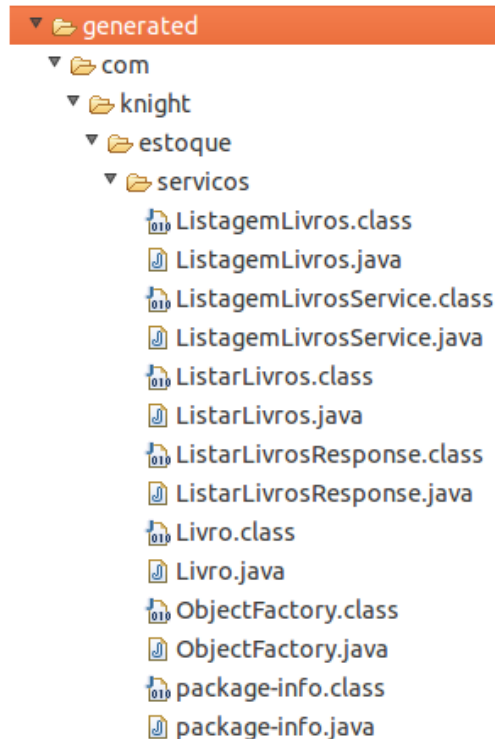
Windows:

```
%JAVA_HOME%/bin/wsimport.exe -s generated  
-keep http://localhost:8080/livros?wsdl
```

Linux:

```
$JAVA_HOME/bin/wsimport.sh -s generated  
-keep http://localhost:8080/livros?wsdl
```

Com este comando, você pode observar que a seguinte estrutura foi criada:



Você pode copiar apenas os arquivos de extensão `.java` para seu projeto. Estes arquivos compõem um cliente JAX-WS completo, sendo que a principal classe, aqui, é `ListagemLivrosService` que implementa a interface `ListagemLivros`. Podemos usá-los para criar um cliente com o seguinte código:

```
package com.knight.estoque.servicos;
```

```
import java.util.List;

public class Client {

    public static void main(String[] args) {

        //Inicia a fábrica dos proxies
        ListagemLivrosService listagemLivrosFactory =
        new ListagemLivrosService();

        //Obtém um proxy
        ListagemLivros listagemLivros =
        listagemLivrosFactory.getListagemLivrosPort();

        //Executa o método remoto
        List<Livro> livros = listagemLivros.listarLivros();
        for (Livro livro : livros) {
            System.out.println("Nome: " + livro.getNome());
        }
    }
}
```

A execução desse código vai produzir a seguinte saída:

```
Nome: Guia do Programador
Nome: Ruby on Rails
```

1.7 OK, E O QUE UM WEB SERVICE TEM A VER COM SOA?

Até aqui, um web service do sistema foi exposto. Isso faz com que você tenha implementado SOA no seu projeto? Não, não faz. E se você implementar mais dez? Vinte? Trinta? Talvez não.

A definição do que faz um sistema ser orientado a serviços não é o número de web services que ele possui (aliás, você poderia construir um sistema orientado a serviços sem nenhum web service — ver [2] para detalhes). No entanto, você deve enxergar além da simples tecnologia para entender o que é SOA. Não se trata de web services, mas sim, de **exposição de lógica de negócios através de meios agnósticos** — ou seja, SOA não é baseada em web services, mas sim em quanto do sistema pode ser acessado por mecanismos externos ao próprio sistema, de maneira independente de linguagem.

Isso implica, automaticamente, que seu web service não precisa necessariamente ser feito com SOAP e um WSDL. Existem outros mecanismos para se oferecer o que se chama de serviço (dentre esses mecanismos, o mais proeminente é, sem sombra de dúvidas, REST [6]). Note que SOA, em si, é totalmente agnóstico — representa mais um conceito do que simplesmente um guia de como ser aplicado.

Aliás, o que é um serviço? A resposta para essa pergunta foi, em grande parte, responsável pela grande confusão em que SOA esteve envolvida nos últimos anos. Segundo (<http://www.soaglossary.com/service.php>) , um serviço é:

“(...) uma unidade de lógica da solução em que a orientação a serviços foi aplicada de maneira significativa.”

—

Ou seja, uma arquitetura orientada a serviços é definida em termos de serviços, que por sua vez, são definidos em termos de orientação a serviços. A orientação a serviços, conforme visto em http://www.soaglossary.com/service_orientation.php (e também no livro de Thomas Erl *SOA Principles of Service Design* [3]), é

“(...)um paradigma de design pretendido para a criação de unidades de lógica de solução que são individualmente moldadas para que possam ser coletivamente e repetidamente utilizadas para realizar um conjunto específico de objetivos (...)”

—

Desta vez o glossário fornece quais são os princípios que compõem esse paradigma, que são:

- Contrato de serviços padronizado;
- Baixo acoplamento;
- Abstração;
- Reutilização;
- Autonomia;
- Não manter estado;
- Habilidade de poder ser descoberto;

- Habilidade de poder ser composto.

Ou seja, você deve moldar sua aplicação para que a parte relevante da lógica de negócios seja exposta em forma de serviços, que devem respeitar estes princípios. Ao ter a arquitetura do seu sistema projetada com isso em mente, faz com que a mesma seja orientada a serviços.

Ao longo deste livro, você verá cada um desses princípios ser aplicado em exemplos práticos, sempre com a devida motivação para cada um. Você verá que SOA não é um bicho de sete cabeças, e pode ser utilizado hoje, na sua própria aplicação, e passar a colher os resultados disso o quanto antes. Divirta-se!

1.8 SUMÁRIO

Você viu, neste capítulo, quais as necessidades para se criar um web service e o que faz deles uma ótima solução para problemas de comunicação entre linguagens. Você também viu como criar um web service simples em Java, como realizar testes (através da ferramenta SoapUI) e como criar um cliente para este web service.

Você também viu que, de acordo com definições, em geral não se pode chamar uma aplicação de SOA se ela possuir apenas um *web service*. Você também viu alguns dos princípios SOA de criação de aplicações.

Claro, deixamos de ver muita coisa em detalhes, ainda estamos só no começo da jornada. Vamos ainda ver a fundo como configurar o JAXB, as opções de customização disponíveis do JAX-WS e quando devemos usá-las. Enfim, há muito por vir, então, vamos continuar?

CAPÍTULO 2

Entendendo o fluxo de dados

“A ignorância afirma ou nega veementemente; a ciência duvida.”

– Voltaire

Já fizemos bastante até o momento. Criamos um serviço, conseguimos consumi-lo através de outro código, testamos com o SoapUI, enfim, tudo parece ótimo. Mas muito ainda não foi explicado: como foi feita a criação do WSDL? Como foi feita a tradução de/para XML? E quanto ao cliente, como ele sabia como enviar as informações?

Para entender isso, precisamos primeiro revisar três conceitos-chave: *namespaces* XML, SOAP e WSDL.

2.1 COMO FUNCIONA A ESTRUTURA DE NAMESPACES DO XML

Basicamente, o XML possui uma estrutura de identificação bastante semelhante à estrutura de nomenclatura em Java (na verdade, de praticamente qualquer lingua-

gem de programação moderna). Para evitar confusão de conceitos, é utilizada uma estrutura de *namespaces*, que são utilizados para diferenciar uma *tag* de outra. Por exemplo, considere um XML de usuários:

```
<usuarios>
...
</usuarios>
```

Estaria tudo bem se essa mesma *tag* não fosse utilizada em diversos sistemas diferentes! Para evitar conflitos entre estruturas de nomes iguais, os *namespaces* podem ser utilizados da mesma maneira, passando a ficar assim:

```
<usuarios xmlns="http://meusistema.com">
...
</usuarios>
```

Essa é uma grande ideia para evitar conflitos entre sistemas que se comunicam por XML. No entanto, às vezes não queremos diferenciar as coisas, ao contrário: no caso de serviços desenvolvidos dentro de uma mesma empresa, é desejável que as *tags* usadas sejam sempre justamente iguais.

Como você pôde ver no capítulo 1, as ferramentas de geração de *web services* criam algumas estruturas de *namespaces* automaticamente. Vejamos novamente o XML de retorno do nosso primeiro serviço:

```
<!-- algumas marcações... -->
<ns2:listarLivrosResponse
  xmlns:ns2="http://servicos.estoque.knight.com/">
  <return>
    <!-- Corpo da resposta.. -->
  </return>
</ns2:listarLivrosResponse>
```

Neste XML, o namespace é <http://servicos.estoque.knight.com/>. Ele pode ser referenciado em qualquer ponto do XML em questão, através do prefixo `ns2`. Desta maneira, quando o elemento XML é `ns2:listarLivrosResponse`, sabemos automaticamente que é o elemento `listarLivrosResponse` definido no namespace <http://servicos.estoque.knight.com/>.

2.2 CONHEÇA O FUNCIONAMENTO DO SOAP

SOAP é uma sigla que significa *Simple Object Access Protocol* (<http://www.w3.org/TR/soap/>). Ele foi criado e é mantido pelo *World Wide Web Consortium*, ou simplesmente W3C. Ele também é conhecido como Envelope SOAP, já que seu elemento raiz é o Envelope. Ele obedece ao seguinte formato geral:

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ser="http://servicos.estoque.knight.com/">
  <!-- Aqui pode ter, ou não, um elemento soapenv:Header -->
  <soapenv:Body>
    <ser:listarLivros />
  </soapenv:Body>
</soapenv:Envelope>
```

O elemento `Envelope` é puramente um *container* para os elementos `Header` e `Body`. O elemento `Body` contém o corpo da requisição, propriamente dito: o nome da operação, parâmetros etc. Já o elemento `Header` contém metadados pertinentes à requisição, como informações de autenticação, endereço de retorno da mensagem etc. Veremos sobre isso mais adiante no livro e como esse elemento serve, em grande parte, como elemento de extensibilidade para a aplicação de especificações de recursos adicionais relativos a web services, como WS-Addressing, WS-Transaction, WS-Security e vários outros recursos.

2.3 ENTENDA O ENORME WSDL

No capítulo 1, ao acessarmos o endereço <http://localhost:8080/livros?wsdl> para visualizar o WSDL, nos deparamos com um enorme XML, que para o desenvolvedor que nunca o viu, pode ser um código grande e assustador:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://servicos.estoque.knight.com/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace="http://servicos.estoque.knight.com/"
  name="ListagemLivrosService">
  <types>
    <xsd:schema>
```

```

    <xsd:import
      namespace="http://servicos.estoque.knight.com/"
      schemaLocation="http://localhost:8080/livros?xsd=1">
    </xsd:import>
  </xsd:schema>
</types>
<message name="listarLivros">
  <part name="parameters" element="tns:listarLivros">
  </part>
</message>
<message name="listarLivrosResponse">
  <part name="parameters" element="tns:listarLivrosResponse">
  </part>
</message>
<portType name="ListagemLivros">
  <operation name="listarLivros">
    <input message="tns:listarLivros"></input>
    <output message="tns:listarLivrosResponse"></output>
  </operation>
</portType>
<binding name="ListagemLivrosPortBinding"
  type="tns:ListagemLivros">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document">
  </soap:binding>
  <operation name="listarLivros">
    <soap:operation soapAction=""></soap:operation>
    <input>
      <soap:body use="literal"></soap:body>
    </input>
    <output>
      <soap:body use="literal"></soap:body>
    </output>
  </operation>
</binding>
<service name="ListagemLivrosService">
  <port name="ListagemLivrosPort"
    binding="tns:ListagemLivrosPortBinding">
    <soap:address location="http://localhost:8080/livros">
    </soap:address>
  </port>

```

```
</service>
</definitions>
```

Agora que já passou o trauma e o susto com esse código, vamos entendê-lo com calma.

Nota-se que o WSDL possui cinco seções bem definidas: `types`, `message` (onde pode existir mais de um elemento), `portType`, `binding` e `service`.

2.4 A DEFINIÇÃO DOS TIPOS E A SEÇÃO TYPES

O JAXB (**J**ava **A**rchitecture for **X**ML **B**inding) é o responsável, em Java, por gerar tipos em **XML Schemas** a partir de classes Java e vice-versa. Por exemplo, tomando como parâmetro a seguinte classe:

```
package com.knight.estoque.modelos;

// imports omitidos

@XmlAccessorType(XmlAccessType.FIELD)
public class Livro {

    private Integer anoDePublicacao;
    private List<String> autores;
    private String editora;
    private String nome;
    private String resumo;

    //Getters e setters...
}
```

O JAXB vai mapear esta classe para o seguinte conteúdo num **XML Schema**:

```
<xs:complexType name="livro">
  <xs:sequence>
    <xs:element name="anoDePublicacao" type="xs:int" minOccurs="0">
    </xs:element>
    <xs:element name="autores" type="xs:string" nillable="true"
      minOccurs="0" maxOccurs="unbounded">
    </xs:element>
    <xs:element name="editora" type="xs:string" minOccurs="0">
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

```

<xs:element name="nome" type="xs:string" minOccurs="0">
</xs:element>
<xs:element name="resumo" type="xs:string" minOccurs="0">
</xs:element>
</xs:sequence>
</xs:complexType>

```

Isto mostra que as classes são, preferencialmente, mapeadas para `complexType`s ao invés de elementos. Você pode notar que os campos da classe Java foram todos mapeados com seus equivalentes em uma linguagem chamada *XML Schema* (na seção 2.11 você verá com mais detalhes como customizar o mapeamento de campos). O atributo `type`, assim como numa classe Java, determina a tipagem do elemento em questão. Esses tipos podem ser simples, como `string`, `int`, `boolean` e outros (a listagem completa está disponível em <http://www.w3.org/TR/xmlschema-2/>) . Também podem ser complexos, como a própria entidade `livro`.

O fato de a classe ser definida como `complexType` quer dizer que ela não pode ser utilizada diretamente no XML. Traçando um paralelo entre a programação Java e o *XML Schema*, é como se o `complexType` fosse a definição de uma classe. A maneira de instanciar esta classe é usando um `element`, que contém as definições corretas de número de ocorrências, tipagem e outros atributos específicos. O atributo `minOccurs` e `maxOccurs` determinam, respectivamente, o número mínimo de ocorrências e o máximo. No caso de listagens, por exemplo, o valor de `maxOccurs` pode ser igual a `unbounded`, o que quer dizer que não há limite superior para o número de elementos com este nome.

A seção `types`, do WSDL, contém essas definições de formatos de dados de acordo com o que é esperado pelo código Java. Os *XML Schemas* (também chamados de XSDs), são o formato ideal para essas definições, já que possuem um formato devidamente padronizado pela W3C, órgão que regulamenta formatos de dados, normas e regras de tráfego e outros assuntos relacionados à passagem de dados pela web. No nosso caso, o esquema contido na seção `types` faz uma importação de um arquivo externo, referenciado pela URL <http://localhost:8080/livros?xsd=1>. O conteúdo deste arquivo é o seguinte:

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:tns="http://servicos.estoque.knight.com/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  version="1.0"

```

```
targetNamespace="http://servicos.estoque.knight.com/">

<xs:element name="listarLivros"
  type="tns:listarLivros">
</xs:element>

<xs:element name="listarLivrosResponse"
  type="tns:listarLivrosResponse">
</xs:element>

<xs:complexType name="listarLivros">
  <xs:sequence></xs:sequence>
</xs:complexType>

<xs:complexType name="listarLivrosResponse">
  <xs:sequence>
    <xs:element name="return"
      type="tns:livro"
      minOccurs="0"
      maxOccurs="unbounded">
    </xs:element>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="livro">
  <xs:sequence>
    <xs:element name="anoDePublicacao"
      type="xs:int"
      minOccurs="0">
    </xs:element>
    <xs:element name="autores"
      type="xs:string"
      nillable="true"
      minOccurs="0"
      maxOccurs="unbounded">
    </xs:element>
    <xs:element name="editora"
      type="xs:string"
      minOccurs="0">
    </xs:element>
    <xs:element name="nome"
```

```
        type="xs:string"
        minOccurs="0">
    </xs:element>
    <xs:element name="resumo"
        type="xs:string"
        minOccurs="0">
    </xs:element>
</xs:sequence>
</xs:complexType>
</xs:schema>
```

Repare que temos a definição do `complexType` `livro`, assim como a definição de tipos `listarLivros` e `listarLivrosResponse`, que contém respectivamente os dados de entrada e de saída do método `listarLivros`. Estes tipos são, por sua vez, referenciados por elementos de mesmo nome. Esta referência faz-se necessária para que estes elementos, e somente estes, sejam acessíveis pelos usuários do esquema, ou seja, os clientes do *web service*.

O PODER DOS XML SCHEMAS

Um XML Schema pode ser usado para definir muitas coisas dentro de um XML, em termos de valores: valores predefinidos, números, sequência de bytes, e até mesmo expressões regulares a que os valores devem estar de acordo. No entanto, este não é o foco deste livro; caso deseje saber mais sobre definições de schemas, consulte a bibliografia deste livro.

2.5 A NECESSIDADE DA SEÇÃO MESSAGES

O objetivo da seção `messages` é “amarrar” diversos elementos, presentes ou importados na seção `types`, para compor os dados que tráfegarão efetivamente de/para o serviço. Lembre-se de que, até aqui, não existe definição alguma quanto ao formato dos dados que tráfegarão para o serviço — sabemos que é XML, mas não temos nenhuma definição de protocolo de camada de aplicação. Neste caso, a seção `messages` é útil para fazer a amarração do corpo da mensagem com cabeçalhos, por exemplo.

O conteúdo desta seção é composto por um ou mais elementos `part`. Cada um deles só precisa conter um `name` (de livre escolha) e um atributo `type` ou `element`,

referenciando, respectivamente, um *XML type* (`simpleType` ou `complexType`) ou um `element`.

Repare que, no caso do nosso WSDL gerado automaticamente, duas mensagens foram definidas:

```
<message name="listarLivros">
  <part name="parameters" element="tns:listarLivros">
  </part>
</message>
<message name="listarLivrosResponse">
  <part name="parameters" element="tns:listarLivrosResponse">
  </part>
</message>
```

Os nomes das mensagens são gerados de acordo com o nome da operação correspondente, e duas mensagens (uma para a requisição e outra para a resposta) são criadas. No entanto, pode haver quantos elementos `message` forem necessários, já que várias operações podem ser definidas num único WSDL.

2.6 A SEÇÃO PORTTYPE

A seção *portType* contém a definição das operações do serviço em questão. Ele pode conter uma ou mais definições de operações, com diversos formatos, a saber:

- Request - response;
- One-way;
- Solicit - response;
- Notification.

Cada um desses formatos possui um mecanismo diferente de mensagem. No modelo *request-response*, uma mensagem de entrada e uma de saída são definidas e a resposta é obtida de maneira síncrona (tornando-se, portanto, o meio mais comum de troca de mensagens).

No modelo *one-way*, apenas uma mensagem de entrada é especificada, de forma que a operação, como diz o nome, é de uma única via, enquanto que no modelo *solicit-response*, uma mensagem de entrada e uma de saída são definidas, porém, a comunicação é feita de modo assíncrono.

No modelo *notification*, apenas uma mensagem de saída é especificada, sendo uma notificação direta do servidor para o cliente.

OS MODELOS *REQUEST-RESPONSE* E *SOLICIT-RESPONSE*

Do ponto de vista do WSDL, ambos os modelos são desenhados da mesma maneira. O mecanismo de transporte deve decidir qual a melhor forma de enviar uma resposta para o cliente. Em geral, esse problema é solucionado através da aplicação do mecanismo WS-Addressing, a ser apresentado, que veremos em detalhes no capítulo 7

2.7 A DIFERENÇA ENTRE OS WSDLs ABSTRATOS E CONCRETOS

Todos os elementos definidos até agora são obrigatórios. Os restantes, *binding* e *service*, são as exceções. Isto porque os elementos definidos até agora são usados para **definir o modelo de mensagens**. Em outras palavras, eles não têm qualquer relação com o meio de transporte em si, e são, portanto, agnósticos em termos de modelo de transporte.

Se o WSDL fosse definido apenas com os elementos que foram explicados até agora, ele seria chamado de **WSDL abstrato**; ou seja, ele apenas define os modelos de mensagem, não os detalhes. Deste modo, ele poderia ficar assim:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://servicos.estoque.knight.com/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace="http://servicos.estoque.knight.com/"
  name="ListagemLivrosService">
  <types>
    <xsd:schema>
      <xsd:import
        namespace="http://servicos.estoque.knight.com/"
        schemaLocation="http://localhost:8080/livros?xsd=1">
      </xsd:import>
    </xsd:schema>
  </types>
```

```
<message name="listarLivros">
  <part name="parameters" element="tns:listarLivros">
  </part>
</message>
<message name="listarLivrosResponse">
  <part name="parameters" element="tns:listarLivrosResponse">
  </part>
</message>
<portType name="ListagemLivros">
  <operation name="listarLivros">
    <input message="tns:listarLivros"></input>
    <output message="tns:listarLivrosResponse"></output>
  </operation>
</portType>
</definitions>
```

Desta maneira, ele poderia ser reutilizado em vários outros WSDLs diferentes, para definir vários formatos diferentes de protocolo de transporte e o método de transporte em si. Isso confere grande flexibilidade ao WSDL que, como mencionado anteriormente, pode ter uma grande gama de protocolos de transporte aplicados. O padrão JAX-WS é o protocolo SOAP (*Simple Object Access Protocol*) sobre HTTP, e este será o foco deste livro.

O WSDL E OS MODOS DE TRANSPORTE

O WSDL, na realidade, não é limitado a nenhum modelo de transporte em particular, e pode ser aplicado virtualmente a qualquer um. No entanto, com o recente crescimento dos REST services, ele tem sido expandido para comportar o modelo REST também. Este modelo expandido é o WSDL 2.0. Como a especificação JAX-WS (e a maioria dos serviços produzidos hoje) ainda não trabalha com este modelo, concentraremos o estudo no WSDL 1.1.

2.8 A SEÇÃO BINDING

A seção `binding` do nosso WSDL que abrimos no capítulo anterior apresenta o seguinte conteúdo:

```

<binding name="ListagemLivrosPortBinding"
  type="tns:ListagemLivros">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document">
  </soap:binding>
  <operation name="listarLivros">
    <soap:operation soapAction=""></soap:operation>
    <input>
      <soap:body use="literal"></soap:body>
    </input>
    <output>
      <soap:body use="literal"></soap:body>
    </output>
  </operation>
</binding>

```

O atributo `type` refere-se ao nome do `portType` a que este `binding` está sendo vinculado. Note que todos os elementos apresentam um *namespace* XML, referenciado em todo o documento pelo prefixo `tns`. Desta maneira, uma referência ao `portType` deve vir acompanhada do prefixo `tns`, tornando-se `tns:ListagemLivros`.

Na sequência, o elemento **soap:binding** cria um elemento específico para o protocolo SOAP 1.1 (o namespace está definido no elemento raiz do WSDL, *definitions*). É este elemento quem diz que o protocolo de aplicação utilizado será o SOAP, versão 1.1. Além disso, ele também define que o padrão SOAP utilizado será o **document style**.

Já no elemento `operation`, são definidos os modelos de tráfego, entre `literal` ou `encoded` (conforme visto no box anterior). Também dentro dele, há a definição de um elemento `soap:operation`, que se encarrega de definir um cabeçalho HTTP, muito comum em tráfego SOAP, chamado `SoapAction`. No entanto, ele é opcional, e é utilizado apenas para facilitar o endereçamento da operação para o método Java a ser executado em definitivo.

SOAP e os padrões de codificação

Os seguintes sub-modelos de transporte, no SOAP, são aceitos:

- Document/literal style;
- Document/literal wrapped style;

- Document/encoded style;
- RPC/literal style;
- RPC/encoded style.

A diferença entre o *document style* e o *rpc style* é bem sutil: trata-se apenas de colocar ou não a operação que está sendo invocada no servidor no corpo da mensagem SOAP. Enquanto o `document` trafega apenas os parâmetros das operações, o `RPC` enriquece estes dados utilizando o nome do método que está definido no WSDL. No entanto, o `document style` pode ser difícil de ser endereçado no servidor, por não conter o nome da operação (lembre-se de que métodos Java podem ser sobrecarregados) e o `rpc` utiliza os nomes das operações que estão definidos no WSDL e que, por estarem separados do resto do conteúdo, também podem ser difíceis de validar.

A solução para este dilema é o uso do *document wrapped style*: trata-se de *document style*, porém com os nomes das operações definidos nos próprios *schemas*. Com este mecanismo, é possível reduzir as fraquezas de ambos os estilos.

Quanto ao `literal/encoded`, a diferença reside quanto a passar informações de tipos na própria mensagem ou não. No entanto, normalmente o uso de `encoded` apenas traz *overhead* de processamento, e deve ser evitado. Para saber mais sobre essas tipagens, consulte <http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/>.

2.9 DEFINIÇÃO DOS ENDEREÇOS COM A SEÇÃO SERVICE

Finalmente, na seção `service`, o WSDL tem o seguinte trecho:

```
<service name="ListagemLivrosService">
  <port name="ListagemLivrosPort"
        binding="tns:ListagemLivrosPortBinding">
    <soap:address location="http://localhost:8080/livros">
    </soap:address>
  </port>
</service>
```

O elemento `service` pode ter um ou mais subelementos `port`, que por sua vez é utilizado para relacionar o `binding` a endereços específicos. No nosso caso,

ele define um subelemento `soap:address`, que é responsável por dizer ao consumidor do serviço para qual endereço enviar o envelope SOAP em questão. Assim, o servidor já está ciente de que naquele endereço é esperado receber várias requisições através de envelopes SOAP, assim como também tem noções de como responder a estas mensagens.

2.10 FINALMENTE, O QUE ACONTECEU?

Enviamos o seguinte conteúdo para o endereço <http://localhost:8080/livros> :

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ser="http://servicos.estoque.knight.com/">
  <!-- Aqui pode ter, ou não, um elemento soapenv:Header -->
  <soapenv:Body>
    <ser:listarLivros />
  </soapenv:Body>
</soapenv:Envelope>
```

Ao fazermos isso, o servidor HTTP que estava habilitado na porta 8080 da nossa máquina identificou que o contexto era `/livros`. Este contexto, por sua vez, estava diretamente relacionado à nossa classe `ListagemLivros`. Ao mandarmos a tag `ser:listarLivros` dentro do elemento `Body`, mostramos ao servidor que essa era uma requisição para a operação `listarLivros`, sem parâmetros. O servidor, ao receber esse conteúdo, realizou a extração do conteúdo do elemento `Body` e fez *parsing* utilizando (no caso da linguagem Java) o JAXB para a conversão deste conteúdo em objetos.

JAXB E A CONVERSÃO DE XML PARA OBJETOS E VICE-VERSA

Você pode ter reparado que não existe, em lugar algum do nosso código, uma classe chamada `ListarLivros` ou algo assim. Então, como o JAXB soube como realizar *parsing* deste conteúdo?

A resposta para esta pergunta reside no próprio mecanismo de criação de classes da JVM. A classe `ListarLivros` é gerada em tempo de execução (você pode conferir este detalhe no próprio log da aplicação, quando executar o exemplo do 1), assim como a classe `ListarLivrosResponse`, que será utilizada para armazenar o resultado da execução. Caso necessário, é possível alterar estas classes em tempo de compilação — mostrarei como no capítulo 3.

2.11 CUSTOMIZE A ESTRUTURA DO XML COM O JAXB

Uma vez que você conhece essas informações, você pode usá-las a seu favor. Por exemplo, o nosso XML de retorno não está exatamente da maneira que queremos. Inicialmente, deparamo-nos com o seguinte XML de resposta do nosso serviço:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:listarLivrosResponse
      xmlns:ns2="http://servicos.estoque.knight.com/">
      <return>
        <anoDePublicacao>2012</anoDePublicacao>
        <autores>Paulo Silveira</autores>
        <autores>Adriano Almeida</autores>
        <editora>Casa do Código</editora>
        <nome>Guia do Programador</nome>
        <resumo>Vá do "nunca programei" ...</resumo>
      </return>
    </return>
    <anoDePublicacao>2012</anoDePublicacao>
    <autores>Vinícius Baggio Fuentes</autores>
    <editora>Casa do Código</editora>
    <nome>Ruby on Rails</nome>
    <resumo>Crie rapidamente aplicações web</resumo>
  </return>
</S:Body>
</S:Envelope>
```

```
</ns2:listarLivrosResponse>
</S:Body>
</S:Envelope>
```

No entanto, a formatação desse XML está um tanto esquisita. Qual o motivo de os elementos de resposta serem `return`? Por que cada autor está numa *tag* `autores`, ao invés de ter cada autor dentro de uma *tag*-pai, que seria `autores`?

Para customizar o elemento `autores`, podemos usar a anotação `javax.xml.bind.annotation.XmlElementWrapper`. Esta anotação pode ser colocada tanto no atributo quanto em seu *getter*. No entanto, sugiro que você coloque no atributo, para que sua classe não fique poluída com *getters* e *setters*, que muitas das vezes são desnecessários.

Em qualquer que seja o lugar em que você coloque as anotações, é preciso orientar o JAXB quanto a onde ele deve retirar essas informações. Para isso, você deve utilizar a anotação `javax.xml.bind.annotation.XmlAccessorType`, com um parâmetro do tipo `javax.xml.bind.annotation.XmlAccessType`. A classe `XmlAccessType` na verdade é um `enum`, que possui os seguintes valores:

- `PROPERTY`: Esta configuração faz com que o JAXB mapeie os *getters* e *setters* para XML;
- `FIELD`: Esta configuração faz com que os campos das classes sejam mapeados para XML (sejam estes públicos ou não);
- `PUBLIC_MEMBER`: Esta configuração faz com que os membros públicos (campos ou *getters/setters*) sejam mapeados. É a configuração padrão, caso nada seja informado.
- `NONE`: Esta configuração faz com que nenhum membro seja mapeado automaticamente, a não ser que o JAXB seja explicitamente informado do contrário.

Desta maneira, a melhor configuração é `FIELD`, já que mapeia automaticamente os campos (mesmo que sejam privados). Para os campos onde queremos modificar informações, utilizamos a anotação `javax.xml.bind.annotation.XmlElement`, que pode ser utilizada para informar ao JAXB o nome da *tag*, se é obrigatória ou não, dentre outras características.

Podemos configurar completamente a classe `Livro` da seguinte forma:


```
package com.knight.estoque.modelos;

// imports omitidos

@XmlAccessorType(XmlAccessType.FIELD)
public class Livro {

    private Integer anoDePublicacao;

    @XmlElementWrapper(name="autores")
    @XmlElement(name="autor")
    private List<String> autores;
    private String editora;
    private String nome;
    private String resumo;

    // getters e setters...

}
```

Desta maneira, nosso XML de retorno fica melhor estruturado e, consequentemente, mais compreensível:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:listarLivrosResponse
      xmlns:ns2="http://servicos.estoque.knight.com/">
      <return>
        <anoDePublicacao>2012</anoDePublicacao>
        <autores>
          <autor>Paulo Silveira</autor>
          <autor>Adriano Almeida</autor>
        </autores>
        <editora>Casa do Código</editora>
        <nome>Guia do Programador</nome>
        <resumo>Vá do "nunca programei" ...</resumo>
      </return>
    </return>
    <anoDePublicacao>2012</anoDePublicacao>
    <autores>
      <autor>Vinícius Baggio Fuentes</autor>
    </autores>
  </S:Body>
</S:Envelope>
```

```

        <editora>Casa do Código</editora>
        <nome>Ruby on Rails</nome>
        <resumo>Crie rapidamente aplicações web</resumo>
    </return>
</ns2:listarLivrosResponse>
</S:Body>
</S:Envelope>

```

A próxima *tag* a ser ajustada é `return`, através de uma configuração do JAX-WS que é feita pela anotação `javax.jws.WebResult`, colocada diretamente no método que é invocado como serviço. Desta maneira, a classe do nosso *web service* fica assim:

```

package com.knight.estoque.servicos;

// imports omitidos

@WebService
public class ListagemLivros {

    @WebResult(name="livro")
    public List<Livro> listarLivros() {
        LivroDAO livroDAO = obterDAO();
        return livroDAO.listarLivros();
    }

    // outros métodos...

}

```

Desta forma, o retorno do nosso serviço fica conforme o desejado:

```

<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:listarLivrosResponse
      xmlns:ns2="http://servicos.estoque.knight.com/">
      <livro>
        <anoDePublicacao>2012</anoDePublicacao>
        <autores>
          <autor>Paulo Silveira</autor>
          <autor>Adriano Almeida</autor>
        </autores>
      </livro>
    </ns2:listarLivrosResponse>
  </S:Body>
</S:Envelope>

```

```
<editora>Casa do Código</editora>
<nome>Guia do Programador</nome>
<resumo>Vá do "nunca programei" ...</resumo>
</livro>

<!-- Restante da resposta -->

</ns2:listarLivrosResponse>
</S:Body>
</S:Envelope>
```

2.12 RESOLVA MÉTODOS DE NOMES IGUAIS E ADIÇÃO DE PARÂMETROS

Como você deve ter notado, nosso serviço não possui até o momento quaisquer parâmetros, porém, a partir de agora, você decide listar os dados de forma paginada. Confiando na técnica de sobrecarga de métodos, você cria um método de nome igual, porém passando como parâmetros o número da página e o tamanho:

```
package com.knight.estoque.servicos;

// Imports omitidos

@WebService
public class ListagemLivros {

    @WebResult(name="livro")
    public List<Livro> listarLivros() {
        LivroDAO livroDAO = obterDAO();
        return livroDAO.listarLivros();
    }

    public List<Livro> listarLivros(Integer numeroDaPagina,
                                     Integer tamanhoDaPagina) {
        LivroDAO livroDAO = obterDAO();
        return livroDAO.listarLivros(numeroDaPagina, tamanhoDaPagina);
    }

    private LivroDAO obterDAO() {
        return new LivroDAO();
    }
}
```

```

    }

    public static void main(String[] args) {
        Endpoint.publish("http://localhost:8080/livros",
                        new ListagemLivros());
        System.out.println("Serviço inicializado!");
    }
}

```

Porém, ao rodar esta classe, você tem uma surpresa ao ver que uma exceção é lançada com a seguinte mensagem:

```

class com.knight.estoque.servicos.jaxws.ListarLivros do not have a
property of the name arg0

```

A mensagem diz que a classe `com.knight.estoque.servicos.jaxws.ListarLivros` não tem uma propriedade de nome `arg0`. Você procura no seu projeto e não acha classe alguma com esse nome. O que aconteceu, então?

Esta classe é gerada em tempo de execução, *runtime*. A engine do JAX-WS é quem a gera, com base no nome do método que está sendo exposto, obedecendo por padrão à seguinte regra: utiliza o nome do pacote onde o serviço está, acrescenta `jaxws` e, então, o nome da operação. No entanto, a operação já existia, pois havia dois métodos com esse nome, então a *engine* do JAX-WS fez a criação da classe para o primeiro método e não achou necessário recriar para o segundo (já que eles têm o mesmo nome). Como esta classe foi criada para o primeiro método, ela não contém os parâmetros necessários para invocar o segundo. Daí a exceção lançada.

É possível customizar esta classe gerada utilizando a anotação `javax.xml.ws.RequestWrapper`, e o código fica assim:

```

@WebResult(name="livro")
public List<Livro> listarLivros() {
    LivroDAO livroDAO = obterDAO();
    return livroDAO.listarLivros();
}

@RequestWrapper(className=
    "com.knight.estoque.servicos.jaxws.ListarLivrosPaginacao")
public List<Livro> listarLivros(Integer numeroDaPagina,

```

```

        Integer tamanhoDaPagina) {
    LivroDAO livroDAO = obterDAO();
    return livroDAO.listarLivros(numeroDaPagina, tamanhoDaPagina);
}

```

No entanto, um novo problema acontece quando você tenta executar este código:

```

Two classes have the same XML type name
"http://servicos.estoque.knight.com/"}listarLivros".
Use @XmlType.name and @XmlType.namespace to assign different
names to them.

```

```

this problem is related to the following location:
    at com.knight.estoque.servicos.jaxws.ListarLivrosPaginacao
this problem is related to the following location:
    at com.knight.estoque.servicos.jaxws.ListarLivros

```

Esta nova exceção acontece porque as classes geradas foram mapeadas para o mesmo elemento do *XSD Schema* gerado. Com isso, é necessário utilizar um de dois atributos da exceção `@RequestWrapper`: ou `localName` ou `targetNamespace`. Caso utilizemos `localName`, devemos atribuir um nome que ainda não exista no XSD, e se for o `targetNamespace`, devemos tomar cuidado para que o elemento `listarLivros` ainda não exista neste *namespace*.

Escolhendo utilizar `localName`, colocamos o valor `listarLivrosPaginacao`. No entanto, os mesmos problemas ocorrem e, seguindo a mesma linha de raciocínio, temos que mapear a resposta também, utilizando a anotação `javax.xml.ws.ResponseWrapper`. O código fica assim:

```

@RequestWrapper(
    className="com.knight.estoque.servicos.jaxws.ListarLivrosPaginacao",
    localName="listarLivrosPaginacao")
@ResponseWrapper(
    className=
        "com.knight.estoque.servicos.jaxws.ListarLivrosPaginacaoResponse",
    localName="livrosPaginados")

public List<Livro> listarLivros(Integer numeroDaPagina,
                                Integer tamanhoDaPagina) {
    LivroDAO livroDAO = obterDAO();
    return livroDAO.listarLivros(numeroDaPagina, tamanhoDaPagina);
}

```

Assim, finalmente o código inicializa; porém, não sem um aviso do JAX-WS:

```
WARNING: Non unique body parts! In a port, as per BP 1.1 R2710
operations must have unique operation signature on the wire for
successful dispatch. Methods [ listarLivros, listarLivros] have
the same request body block
{http://servicos.estoque.knight.com/}listarLivrosPaginacao. Method
dispatching may fail, runtime will try to dispatch using SOAPAction.
Another option is to enable AddressingFeature to enabled runtime to
uniquely identify WSDL operation using wsa:Action header.
```

Esta mensagem está dizendo que, apesar do JAX-WS ter conseguido gerar os documentos, problemas de roteamento (ou seja, fazer o XML chegar até o código Java) podem acontecer caso você insista em utilizar este código, pois as operações na seção `portType` continuam definidas com o mesmo nome. Ele diz que ainda assim pode tentar fazer o roteamento (utilizando o *header* `SOAPAction`); no entanto, este mecanismo pode apresentar falhas. Você tenta importar o projeto no SoapUI e ele dá uma mensagem de erro:

```
ERROR:An error occured [Duplicate operation with name=listarLivros,
inputName=:none, outputName=:none, found in portType
' {http://servicos.estoque.knight.com/}ListagemLivros' .], see error log
for details
```

Para alterar o nome do método, você pode utilizar a anotação `javax.jws.Webmethod`. Com esta anotação, você consegue configurar o nome que a operação terá no WSDL. O código fica assim:

```
@RequestWrapper(
    className="com.knight.estoque.servicos.jaxws.ListarLivrosPaginacao",
    localName="listarLivrosPaginacao")
@ResponseWrapper(
    className=
        "com.knight.estoque.servicos.jaxws.ListarLivrosPaginacaoResponse",
    localName="listarLivrosPaginacaoResponse")
@WebResult(name="livro")
@WebMethod(operationName="listarLivrosPaginacao")
public List<Livro> listarLivros(int numeroDaPagina,
                                int tamanhoDaPagina) {
    LivroDAO livroDAO = obterDAO();
    return livroDAO.listarLivros(numeroDaPagina, tamanhoDaPagina);
}
```

Finalmente, você consegue inicializar o serviço, sem quaisquer alertas. Se você checar as seções `portType` e `binding` do WSDL, verá que elas ficaram assim (algumas informações foram omitidas para melhor legibilidade):

```
<portType name="ListagemLivros">
  <operation name="listarLivros">
    <input message="tns:listarLivros"></input>
    <output message="tns:listarLivrosResponse"></output>
  </operation>
  <operation name="listarLivrosPaginacao">
    <input message="tns:listarLivrosPaginacao"></input>
    <output message="tns:listarLivrosPaginacaoResponse"></output>
  </operation>
</portType>
<binding name="ListagemLivrosPortBinding" type="tns:ListagemLivros">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document" />
  <operation name="listarLivros">
    <soap:operation soapAction=""></soap:operation>
    <input>
      <soap:body use="literal"></soap:body>
    </input>
    <output>
      <soap:body use="literal"></soap:body>
    </output>
  </operation>
  <operation name="listarLivrosPaginacao">
    <soap:operation soapAction=""></soap:operation>
    <input>
      <soap:body use="literal"></soap:body>
    </input>
    <output>
      <soap:body use="literal"></soap:body>
    </output>
  </operation>
</binding>
```

No final das contas, fica uma lição aprendida: usar sobrecarga de métodos em *web services* é muito difícil. Existem técnicas que podem possibilitar, mas no final das contas, acaba sendo trabalhoso demais para resolver um problema que nós poderíamos ter solucionado simplesmente renomeando o método para

listarLivrosPaginacao, por exemplo.

JAX-WS E O MAPEAMENTO DE MÉTODOS

Assim como acontecem diversos problemas com métodos sobrecarregados em uma mesma classe, algumas *engines* JAX-WS também têm problemas com métodos de nomes iguais, ainda que eles estejam em classes diferentes. Em geral, não é recomendado ter métodos de nomes iguais em ponto algum do projeto — o que, aliás, é condizente com alguns princípios de SOA que veremos ao longo deste livro.

Quando renomeamos o método, tudo é gerado com sucesso. Ao regerar o projeto no SoapUI, observamos a seguinte requisição pré gerada:

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ser="http://servicos.estoque.knight.com/">
  <soapenv:Header/>
  <soapenv:Body>
    <ser:listarLivrosPaginacao>
      <!--Optional:-->
      <arg0?></arg0>
      <!--Optional:-->
      <arg1?></arg1>
    </ser:listarLivrosPaginacao>
  </soapenv:Body>
</soapenv:Envelope>
```

Se fizermos a requisição assim, obteremos o seguinte retorno:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
      <faultcode>S:Server</faultcode>
      <faultstring>java.lang.NullPointerException</faultstring>
    </S:Fault>
  </S:Body>
</S:Envelope>
```


Isso quer dizer que a *engine* JAX-WS capturou uma `NullPointerException` internamente. Se observamos o *stack trace*, na saída padrão da nossa aplicação, vamos chegar à conclusão de que isso aconteceu porque os parâmetros eram, na verdade, nulos. Isso porque o JAX-WS não fez a atribuição da `?` aos parâmetros (que eram do tipo `Integer`). Ou seja, ele preferiu simplesmente ignorar os valores. Isso porque, como você deve ter observado, colocamos os parâmetros como objetos *wrapper*, ou seja, objetos para armazenar internamente os valores dos números.

É possível colocar tipos primitivos como parâmetros; no entanto, eles sempre serão populados com valores *default* se assim o fizermos, e não teremos condições de diferenciar se o requisitante passou ou não parâmetros para o serviço. Portanto, geralmente é uma boa prática colocar objetos *wrapper* como parâmetros.

2.13 ENVIE A REQUISIÇÃO PELO CLIENTE

Você já cumpriu quase todas as etapas: reescreveu o serviço, checkou a existência de problemas na inicialização e o testou com SoapUI. Uma vez funcionando da maneira esperada, você executa o cliente que havia escrito anteriormente quando você tem uma surpresa: nada acontece. O cliente aparentemente é executado; no entanto, ele não imprime nada na saída. O que aconteceu?

Lembre-se de que você fez alterações na estrutura de retorno. Isso faz com que o *client* não apresente problemas de execução ou compilação, mas também não consiga atribuir os resultados da execução a objetos Java. Ou seja, **lembre-se de que uma alteração nos *web services* nunca originarão problemas de compilação, apenas em tempo de execução**. Portanto, tenha em mente que uma boa política de testes deve ser aplicada nos seus ambientes sempre.

Você também poderia utilizar *scripts* num servidor de integração contínua para sempre reger os clientes dos seus *web services* quando você fizer alterações neles, assim como uma ferramenta de *build* para utilizar sempre as versões mais recentes destes *clients*.

Caso você faça uma alteração no seu serviço, como inclusão de método (sem nenhuma alteração nos métodos já existentes), nada acontecerá. No entanto, se você retirar um método (ainda que você não esteja utilizando esse método em lugar algum), o seguinte problema acontecerá:

```
Method listarLivrosPaginacao is exposed as WebMethod, but there is no
corresponding wsdl operation with name listarLivrosPaginacao in the
wsdl:portType{http://servicos.estoque.knight.com/}ListagemLivros
```

Isso porque a classe que fará a comunicação propriamente dita com o servidor também é gerada em tempo de execução, do lado do cliente. Mas essa classe será gerada implementando uma interface Java. Se um método não existir mais do lado do servidor (checagem que é feita pelo WSDL), a *engine* do JAX-WS não terá condições de gerar essa classe, originando a exceção.

Existe a possibilidade de contornar este problema (e diversos outros, como *overhead* de trazer uma cópia do WSDL a cada criação de *client*) mantendo um *cache* local — ou seja, no lado do cliente — do WSDL. Essa abordagem pode ter problemas com versionamento do contrato; mas, do ponto de vista de produção é mais vantajoso manter esse *cache*, já que as aplicações já estarão testadas e não devem, ou não deveriam, pelo menos, ter esse tipo de problema.

O código do cliente fica como:

```
public static void main(String[] args) {

    ListagemLivrosService listagemLivrosService =
        new ListagemLivrosService(Client.class.getResource("/livros.wsdl"));

    ListagemLivros listagemLivros =
        listagemLivrosService.getListagemLivrosPort();

    List<Livro> livros = listagemLivros.listarLivrosPaginacao(0, 2);

    for (Livro livro : livros) {
        System.out.println("Nome do livro: " + livro.getNome());
    }

}
```

2.14 SUMÁRIO

Neste capítulo, você viu alguns dos mecanismos mais elementares de funcionamento de XML. Notavelmente, um dos mecanismos mais utilizados é o sistema de *namespaces*, que são utilizados para fazer referências a elementos XML. Você também viu quais são as seções de um envelope SOAP e as seções de um WSDL. Finalmente, você viu como cada uma dessas seções trabalha em conjunto para tirar o maior proveito em termos de extensibilidade e baixo acoplamento.

Você também viu como o *web service* Java trabalha para realizar a interpretação dessas informações, de maneira que execute corretamente o método no *web service* e

retorne os dados para o cliente. Você também viu alguns dos possíveis problemas da geração do *web service*, como a nomenclatura dos métodos, sistema de parâmetros e a criação/destruição de métodos.

No próximo capítulo, você verá algumas técnicas para manter seu projeto sustentável, de maneira a diminuir alguns dos possíveis problemas de comunicação.

CAPÍTULO 3

Novos desafios e os ajustes finos para controles de exceções e adaptação de dados

“Nós somos o que fazemos repetidas vezes. Portanto, a excelência não é um ato, mas um hábito.”

– Aristóteles

Seu *web service* está crescendo. Uma nova solicitação da diretoria pede que você crie um novo método no seu serviço, para que seus usuários sejam também capazes de criar novos livros em exposição. No entanto, esta criação requer um sistema de autorização de usuários — não basta apenas expor o serviço. Você decide então criar a seguinte assinatura para o método:

```
public void criarLivro (Livro livro, Usuario usuario)
    throws UsuarioNaoAutorizadoException
```

Todavia, você ainda não sabe como o sistema vai se comportar com a passagem de parâmetros e o tratamento de exceções. O que fazer?

3.1 CUSTOMIZE O SISTEMA DE PARÂMETROS

Quando você cria esse código, a seguinte entrada para o seu serviço é gerado pelo SoapUI:

```
<!-- declarações de namespaces omitidas -->
<soapenv:Envelope>
  <soapenv:Header>
  </soapenv:Header>
  <soapenv:Body>
    <ser:criarLivro>
      <!--Optional:-->
      <arg0>
        <!--Optional:-->
        <anoDePublicacao?</anoDePublicacao>
        <!--Optional:-->
        <autores>
          <!--Zero or more repetitions:-->
          <autor?</autor>
        </autores>
        <!--Optional:-->
        <editora?</editora>
        <!--Optional:-->
        <nome?</nome>
        <!--Optional:-->
        <resumo?</resumo>
        <!--Optional:-->
        <dataDeCriacao?</dataDeCriacao>
      </arg0>
      <arg1>
        <!--Optional:-->
        <nome?</nome>
        <!--Optional:-->
        <login?</login>
        <!--Optional:-->
        <senha?</senha>
      </arg1>
    </ser:criarLivro>
```

```
</soapenv:Body>
</soapenv:Envelope>
```

O primeiro problema é detectado assim que você vê a requisição: o que é `arg0` e `arg1`? Os nomes desses parâmetros deveriam ser, respectivamente, `livro` e `usuario`, certo? O que aconteceu?

O grande problema, aqui, é que o Java não possui nenhum sistema capaz de “avisá-lo” de quais são os parâmetros. Em outras palavras, não é possível armazenar o nome dos parâmetros já que o Java não retém essa informação. Também não seria sábio simplesmente usar os nomes das classes, já que a assinatura dos métodos poderia conter vários parâmetros diferentes utilizando a mesma classe (por exemplo, se os parâmetros dos métodos fossem `String`, não seria elegante colocar os nomes dos parâmetros, no *web service*, como `string`).

Para “contornar” este problema, pode-se anotar os próprios parâmetros com a anotação `@WebParam`, ficando assim:

```
public void criarLivro (@WebParam(name="livro") Livro livro,
                        @WebParam(name="usuario") Usuario usuario)
    throws UsuarioNaoAutorizadoException
```

Ao visualizar a nova requisição no SoapUI, você vê o seguinte XML:

```
<soapenv:Envelope>
  <soapenv:Header/>
  <soapenv:Body>
    <ser:criarLivro>
      <livro>
        <!--Elementos de livro-->
      </livro>
      <usuario>
        <!-- Elementos de usuário -->
      </usuario>
    </ser:criarLivro>
  </soapenv:Body>
</soapenv:Envelope>
```

Nada mal! Ao colocar a anotação `@WebParam` nos parâmetros, você tem uma forma de dizer ao JAX-WS o deve ser feito com cada um. Quando você utiliza uma IDE e usa o *auto-complete* nos atributos da anotação, você percebe alguns interessantes:

- name
- partName
- targetNamespace
- mode
- header

Destes, um em particular lhe chama a atenção: `header`. Este atributo é definido como um `boolean`. O que isso quer dizer? Você altera a anotação, então, para ficar assim:

```
public void criarLivro (@WebParam(name="livro")Livro livro,  
                        @WebParam(name="usuario", header=true) Usuario usuario) ...
```

Ao refazer a requisição no SoapUI, você vê o seguinte XML:

```
<soapenv:Envelope>  
  <soapenv:Header>  
    <ser:usuario>  
      <!-- elementos de usuario -->  
    </ser:usuario>  
  </soapenv:Header>  
  <soapenv:Body>  
    <ser:criarLivro>  
      <livro>  
        <!-- elementos de livro -->  
      </livro>  
    </ser:criarLivro>  
  </soapenv:Body>  
</soapenv:Envelope>
```

Você notou que, agora, em vez de `usuario` ser colocado dentro do elemento `Body`, ele está em `Header`. Qual a diferença?

Em geral, o elemento `Header`, num envelope SOAP, é utilizado para passagem de metadados e itens que não fazem parte da requisição propriamente dita. No nosso caso, estamos invocando a operação de criar um livro. O usuário é apenas um elemento de segurança, que não faz parte da criação propriamente dita do livro. Na verdade, o usuário é apenas um elemento de segurança — ele pode, ou não, ser utilizado pela requisição.

SOAP E SEGURANÇA

Pelo simples fato de SOAP ser “íntimo” do WSDL, pode-se utilizar diversos esquemas de autenticação/autorização que são definidos no próprio WSDL. Quando o cliente lê esses esquemas, já está ciente de que deve passar certas informações a mais, relativas à segurança. O conjunto de práticas de segurança é conhecido como *WS-Security*, que veremos no capítulo 6.

3.2 CONHEÇA O SISTEMA DE LANÇAMENTO DE EXCEÇÕES

Você decide colocar no seu método um código para tratamento de autenticação do usuário. Inicialmente, você simplesmente projeta a aplicação para lançar uma exceção, caso o usuário não seja autenticado. Seu código fica assim:

```
public void criarLivro (@WebParam(name="livro") Livro livro,
    @WebParam(name="usuario", header=true) Usuario usuario)
    throws UsuarioNaoAutorizadoException, SOAPException {

    if (usuario.getLogin().equals("soa") &&
        usuario.getSenha().equals("soa")) {
        obterDAO().criarLivro(livro);
    } else {
        throw new UsuarioNaoAutorizadoException("Não autorizado");
    }
}
```

Para testar esse código, você faz uma requisição com informações inválidas de autorização e obtém a seguinte resposta:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
      <faultcode>S:Server</faultcode>
      <faultstring>Usuário não autorizado</faultstring>
      <detail>
        <ns2:UsuarioNaoAutorizadoException
          xmlns:ns2="http://servicos.estoque.knight.com/">
          <message>Não autorizado</message>
        </ns2:UsuarioNaoAutorizadoException>
      </detail>
    </S:Fault>
  </S:Body>
</S:Envelope>
```

```
</ns2:UsuarioNaoAutorizadoException>
</detail>
</S:Fault>
</S:Body>
</S:Envelope>
```

Analisando o código, você nota que o conteúdo da *tag* `Body` não é mais algo como `criarLivroResponse`, mas sim, uma *tag* `Fault` (que aliás, possui o mesmo *namespace* que as *tags* `Body` e `Envelope`).

Esta *tag* representa, no SOAP, que uma falha foi lançada. No conteúdo do elemento, temos três outros elementos: `faultcode`, `faultstring` e `detail`. O campo `faultcode` aceita quatro valores: `VersionMismatch`, `MustUnderstand`, `Client` e `Server` (veja http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383510 para mais informações a respeito). Nesse aspecto, os campos que a nossa aplicação deve aplicar são, de fato, `Client` (que indica um problema na requisição enviada) e `Server` (que indica um problema no processamento realizado pelo servidor). Veremos na seção 3.4 como manipular esse campo.

Além disso, temos os campos `faultstring`, que contém uma descrição do problema, e o campo `detail`. Este, por sua vez, contém uma descrição detalhada da exceção levantada, no caso, `UsuarioNaoAutorizadoException`.

SOAP E AS EXCEÇÕES NO CORPO DA MENSAGEM

A especificação SOAP proíbe que as exceções trafegadas no corpo da mensagem sejam relativas a problemas de dados passados no elemento `Header`. No entanto, tenha em mente que estes problemas de autenticação são tratados pela especificação `WS-Security`.

O código da classe `UsuarioNaoAutorizadoException` é o seguinte:

```
package com.knight.estoque.servicos;

public class UsuarioNaoAutorizadoException extends Exception {

    public UsuarioNaoAutorizadoException() {}

    public UsuarioNaoAutorizadoException(String message) {
```

```
        super(message);
    }

    public UsuarioNaoAutorizadoException(Throwable cause) {
        super(cause);
    }

    public UsuarioNaoAutorizadoException(String message,
                                           Throwable cause) {
        super(message, cause);
    }

    //Construtor JDK 1.7
    public UsuarioNaoAutorizadoException(String message, Throwable cause,
                                           boolean enableSuppression, boolean writableStackTrace) {
        super(message, cause, enableSuppression, writableStackTrace);
    }
}
```

Como você pode notar, é uma exceção comum. O campo `message`, presente no XML de retorno da mensagem, é proveniente do método `getMessage`, presente em toda a hierarquia de exceções Java.

3.3 CUSTOMIZE A SESSÃO DE DETALHES

Examinemos a seção de detalhes da exceção lançada:

```
<ns2:UsuarioNaoAutorizadoException
  xmlns:ns2="http://servicos.estoque.knight.com/">
  <message>Usuário não autorizado</message>
</ns2:UsuarioNaoAutorizadoException>
```

O que acontece se o cliente não desejar, no entanto, ter um campo `message` mapeado? O que acontece se, por acaso, você desejar produzir um mapeamento mais detalhado da exceção (trocar o *namespace*, trocar o nome do elemento etc)?

Esse mapeamento customizado é provido, em partes, através da anotação `@WebFault`. Por exemplo, suponha que queiramos trocar o *namespace* desta exceção para `http://servicos.estoque.knight.com/excecoes/` e o nome do elemento para `UsuarioNaoAutorizado`. Basta que anotemos a classe assim:

```
@WebFault(
    targetNamespace="http://servicos.estoque.knight.com/excecoes/",
    name="UsuarioNaoAutorizado")
public class UsuarioNaoAutorizadoException extends Exception {
    //Construtores
}
```

Quando fazemos esta alteração e enviamos a requisição novamente, o resultado fica assim:

```
<ns3:UsuarioNaoAutorizado
  xmlns:ns3="http://servicos.estoque.knight.com/excecoes/"
  xmlns:ns2="http://servicos.estoque.knight.com/">
  <message>Usuário não autorizado</message>
</ns3:UsuarioNaoAutorizado>
```

No entanto, o campo `message` continua lá. É impossível tirar esse mapeamento (já que o JAXB, por padrão, mapeia os campos que começam com `get` e o método `getMessage` pertence à classe `Exception`). O que fazer, então?

Pensando neste problema, o JAX-WS fornece um método chamado `getFaultInfo`. Este método pode retornar um objeto (compatível com JAXB) para ser usado como o mapeamento da própria exceção. Por exemplo, supondo que queiramos que o elemento `UsuarioNaoAutorizado` contenha um atributo mensagem, tendo a mensagem que foi usada como descrição da exceção, podemos ter o seguinte código:

```
@WebFault(
    targetNamespace="http://servicos.estoque.knight.com/excecoes/",
    name="UsuarioNaoAutorizado")
public class UsuarioNaoAutorizadoException extends Exception {

    // construtores

    public UsuarioFaultInfo getFaultInfo() {
        return new UsuarioFaultInfo(getMessage());
    }

    @XmlAccessorType(XmlAccessType.FIELD)
    public static class UsuarioFaultInfo {

        @XmlAttribute
```

```

        private String mensagem;

        public UsuarioFaultInfo(String mensagem) {
            this.mensagem = mensagem;
        }

        public UsuarioFaultInfo() {
        }
    }
}

```

O código de lançamento da exceção pode permanecer inalterado (algo como `throw new UsuarioNaoAutorizadoException("Usuário não autorizado");`). Ainda assim, quando testamos este código, obtemos o seguinte retorno:

```

<ns3:UsuarioNaoAutorizado mensagem="Usuário não autorizado"
  xmlns:ns3="http://servicos.estoque.knight.com/excecoes/"
  xmlns:ns2="http://servicos.estoque.knight.com/">

```

3.4 CUSTOMIZE AINDA MAIS O LANÇAMENTO DE EXCEÇÕES

Até aqui, toda a manipulação de exceções foi feita através do lançamento delas. Ainda assim, o código não é flexível o bastante: lembre-se de que ainda existe o campo `faultcode` a ser preenchido de maneira mais adequada; além disso, existe um campo chamado `faultactor` que sequer foi populado. Como resolver este problema?

Através da API do JAX-WS, é possível realizar manipulação das exceções por meio de manipulação direta de XML, usando um mecanismo híbrido da API JAX-WS e DOM (*Document Object Model*). Isso pode ser inicializado a partir da classe `SOAPFactory`, que contém métodos utilitários para criação de *templates* SOAP. Por exemplo, para manipularmos livremente estes campos, podemos utilizar o seguinte código:

```

SOAPFault soapFault = SOAPFactory.newInstance().createFault(
    "Usuário não autorizado",
    new QName(SOAPConstants.URI_NS_SOAP_1_1_ENVELOPE,
        "Client.autorizacao"));
soapFault
    .setFaultActor("http://servicos.estoque.knight.com/LivrosService");

```

```
throw new SOAPFaultException(soapFault);
```

Este código irá produzir a seguinte saída:

```
<S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
  <faultcode>S:Client.autorizacao</faultcode>
  <faultstring>Usuário não autorizado</faultstring>
  <faultactor>
    http://servicos.estoque.knight.com/LivrosService
  </faultactor>
</S:Fault>
```

O campo `faultcode`, de acordo com a especificação SOAP, deve ser preenchido com o prefixo XML que referencia o *namespace* SOAP (ou seja, <http://schemas.xmlsoap.org/soap/envelope/>), seguido de um dos seguintes valores:

- `VersionMismatch`
- `MustUnderstand`
- `Client`
- `Server`

De uma maneira prática, os únicos campos que nos interessam, no dia a dia, são os campos `Client` e `Server`, que indicam, respectivamente, erros na requisição passada pelo cliente ou um problema no processamento interno. A partir daí, informações a mais são incluídas com `.` (como no exemplo, onde o retorno é `Client.autorizacao`, indicando um problema de autenticação na requisição enviada pelo cliente).

Quando o campo `faultcode` é populado, é necessário utilizar um prefixo XML, que em nosso caso `S` referencia o *namespace* <http://schemas.xmlsoap.org/soap/envelope/>, que o código Java enxerga a partir da constante `URI_NS_SOAP_1_1_ENVELOPE`, presente na interface `SOAPConstants`.

O campo `faultactor`, por outro lado, deve ser populado com uma URI que referencia o componente que originou o problema (no nosso caso, o serviço `LivrosService`).

Por último, uma vez que o objeto `SOAPFault` é populado, basta lançar uma exceção do tipo `SOAPFaultException`, passando o `SOAPFault` criado no construtor. Esta exceção faz parte da API para manipulação de SOAP, e quando ela é lançada, você está automaticamente dizendo ao JAX-WS o que fazer.

Particularmente, a abordagem que me parece mais atraente entre as duas é o lançamento de uma exceção de negócios (como no caso da `UsuarioNaoAutorizadoException`). No entanto, em cenários críticos — onde o volume de dados trafegado deve ser o menor possível — pode ser necessário diminuí-lo ao máximo. Neste caso, a segunda opção pode ser mais interessante.

3.5 EMBARCANDO MAIS A FUNDO NO JAXB

Pode ser interessante (para fins de *debugging*, por exemplo) que as exceções carreguem informações de data/hora. O JAXB é compatível com os tipos que existem nativamente na API Java, como `Calendar` e `Date`. Você decide, então, modificar a classe `UsuarioFaultInfo`, que carrega as informações da exceção de autorização, para que contenha a data de criação da exceção. O código poderia ficar assim:

```
@XmlAccessorType(XmlAccessType.FIELD)
public static class UsuarioFaultInfo {

    @XmlAttribute
    private String mensagem;
    private Calendar data = Calendar.getInstance();

    public UsuarioFaultInfo(String mensagem) {
        this.mensagem = mensagem;
    }

    public UsuarioFaultInfo() {}
}
```

Desta forma, a exceção lançada ficaria assim:

```
<ns3:UsuarioNaoAutorizado mensagem="Usuário não autorizado">
  <data>2012-09-30T23:58:35.648-03:00</data>
</ns3:UsuarioNaoAutorizado>
```

No entanto, esta informação parece complexa demais. Ela está no seguinte formato:

```
{ano}-{mês}-{dia}T{hora}:{minutos}:{segundos}.{milissegundos}-{timezone}
```

Este é o formato para datas completas utilizado em XML. No entanto, ele pode sofrer reduções de tamanho baseado em regras arbitrárias.

Para realizar estas alterações, no entanto, é necessário utilizar uma classe especial, chamada `XMLGregorianCalendar`. Ao utilizá-la, é possível modificar estes valores, bem como removê-los completamente. No entanto, esta classe adiciona sua própria camada de complexidade, como o fato de não liberar a criação da classe diretamente, sendo necessário utilizar uma classe de apoio chamada `DatatypeFactory`:

```
import javax.xml.datatype.DatatypeConfigurationException;
import javax.xml.datatype.DatatypeConstants;
import javax.xml.datatype.DatatypeFactory;
import javax.xml.datatype.XMLGregorianCalendar;

// demais imports

@XmlAccessorType(XmlAccessType.FIELD)
public static class UsuarioFaultInfo {

    @XmlAttribute
    private String mensagem;
    private XMLGregorianCalendar data;

    public UsuarioFaultInfo(String mensagem) {
        this.mensagem = mensagem;
        try {
            this.data = DatatypeFactory.newInstance()
                .newXMLGregorianCalendar(new GregorianCalendar());
        } catch (DatatypeConfigurationException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Por si só, este código produz a mesma saída que utilizar um `GregorianCalendar` comum. No entanto, é possível usar métodos acessórios, em conjunto com a classe `DatatypeConstants`, para desconfigurar valores.

Por exemplo, suponha que você precise remover as informações de milissegundos e *timezone*. É possível fazer isso através do seguinte código:

```
this.data.setMillisecond(DatatypeConstants.FIELD_UNDEFINED);  
this.data.setTimezone(DatatypeConstants.FIELD_UNDEFINED);
```

Desta maneira, a saída do serviço ficará assim:

```
<data>2012-10-01T00:32:02</data>
```

3.6 TIRE PROVEITO DE ADAPTADORES

Sejamos francos: é rudimentar adaptar o nosso sistema em relação ao sistema de formatação para XML. A coisa certa a fazer seria **adaptar o sistema de formatação**, e não adaptar a classe. Pensando nesse princípio, o JAXB possui o conceito de **adaptadores**.

Um adaptador é uma classe Java especializada em fazer tradução de formatos, ou seja, transformar uma classe Java em outra. Esta tradução é útil quando temos uma classe (ou mesmo uma interface) que não conseguimos mapear porque, de alguma forma, ela é fechada para nós. Alguns exemplos disso são a classe `java.util.Date`, a classe `java.util.Calendar`, a interface `java.util.Map` etc.

Esta tradução é realmente muito simples. Trata-se apenas de criar a classe adaptadora e dizer ao JAXB qual classe utilizar para fazer a tradução de tipos. Por exemplo, suponha que você queira incluir um campo `dataDeCriacao` na classe `Livro`. Esta classe faz parte do seu modelo e pode ser mapeada para um banco de dados utilizando JPA e, portanto, faz mais sentido utilizar nesse mapeamento uma classe compatível com JPA, como `Date` ou `Calendar`.

MAPEANDO CLASSES DE WEB SERVICES PARA O BANCO DE DADOS

É perfeitamente possível mapear as classes que contêm anotações JAXB para o banco de dados. No entanto, é considerada melhor prática separar um mapeamento do outro. Por exemplo, se você configurar o JAXB para acessar os dados da sua classe diretamente pelos campos, você deve configurar o JPA para acessar estes dados por *getters* e *setters* (e vice-versa). Além disso, note que é perfeitamente possível colocar comportamentos de negócio nas classes anotadas com JAXB e JPA — você verá mais à frente técnicas mais elaboradas para fazer isso.

Vejamos a classe:

```
@XmlAccessorType(XmlAccessType.FIELD)
public class Livro {

    private Integer anoDePublicacao;

    @XmlElementWrapper(name = "autores")
    @XmlElement(name = "autor")
    private List<String> autores;
    private String editora;
    private String nome;
    private String resumo;
    private Date dataDeCriacao = new Date();

    // getters e setters
}
```

Se você fizer uma requisição para seu serviço dessa maneira, obterá, entre outras coisas, resultados como o seguinte:

```
<dataDeCriacao>2012-10-02T01:13:14.960-03:00</dataDeCriacao>
```

Como já vimos antes, esses dados contêm informações minuciosas sobre a data, como os milissegundos da data e a informação de *timezone*. No entanto, se quisermos mapear estes dados de outra forma, podemos utilizar a classe

XMLGregorianCalendar - que, no entanto, não é compatível com nosso negócio; é mais uma classe de adaptação para o formato de XML.

Aqui, podemos utilizar um adaptador, que irá transformar a instância de `Date` em uma instância de `XMLGregorianCalendar`. Para isso, o segredo é criar uma classe que estenda `javax.xml.bind.annotation.adapters.XmlAdapter`. Esta classe possui dois métodos abstratos, `marshal` e `unmarshal`, que vão, respectivamente, fazer a tradução para o tipo que deve ser considerado válido para tráfego pelo XML e fazer a tradução a partir deste.

Por exemplo, podemos criar o seguinte adapter para realizar a tradução entre `Date` e `XMLGregorianCalendar`:

```
package com.knight.estoque.adaptadores;

// imports omitidos

public class AdaptadorDate extends
    XmlAdapter<XMLGregorianCalendar, Date> {

    public XMLGregorianCalendar marshal(Date date) throws Exception {

        GregorianCalendar calendar = new GregorianCalendar();
        calendar.setTime(date);
        XMLGregorianCalendar xmlGregorianCalendar = DatatypeFactory
            .newInstance().newXMLGregorianCalendar(calendar);
        xmlGregorianCalendar
            .setMillisecond(DatatypeConstants.FIELD_UNDEFINED);
        xmlGregorianCalendar
            .setTimezone(DatatypeConstants.FIELD_UNDEFINED);

        return xmlGregorianCalendar;
    }

    public Date unmarshal(XMLGregorianCalendar v) throws Exception {
        Date date = v.toGregorianCalendar().getTime();
        return date;
    }
}
```

Na sequência, anotamos o campo que queremos adaptar com `javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter`:

```
@XmlJavaTypeAdapter(AdaptadorDate.class)
private Date dataDeCriacao = new Date();
```

Agora, quando fizermos uma nova requisição para o serviço, obteremos o seguinte resultado:

```
<dataDeCriacao>2012-10-02T01:13:14</dataDeCriacao>
```

Também é possível utilizar adaptadores através do pacote inteiro, utilizando uma combinação de descrição do pacote e o `XMLAdapter`. Basta que se crie um arquivo `package-info.java`, no diretório do pacote, contendo a declaração `package` e a anotação:

```
@javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter(
    type = java.util.Date.class,
    value = com.knight.estoque.adaptadores.AdaptadorDate.class)
package com.knight.estoque.modelos;
```

Assim, mesmo que retiremos a anotação `XmlJavaTypeAdapter` da classe `Livro`, a tradução de tipos irá funcionar.

3.7 TRABALHE COM JAXB USANDO HERANÇA

Um recurso que também pode ser muito útil quando trabalhamos com JAXB é a questão da herança. Por exemplo, suponha que queiramos mapear uma classe chamada `EBook`, que estende `Livro`:

```
package com.knight.estoque.modelos;

import java.util.List;

public class EBook extends Livro {

    private String formato = "PDF";

    public EBook() {
        super();
    }

    public EBook(Integer anoDePublicacao, List<Autor> autores,
        String editora, String nome, String resumo) {
```

```
        super(anoDePublicacao, autores, editora, nome, resumo);
    }

    public String getFormato() {
        return formato;
    }

    public void setFormato(String formato) {
        this.formato = formato;
    }
}
```

Mesmo que façamos a inserção desta classe no arquivo `jaxb.index`, a *engine* do JAXB não realizará o mapeamento. Isso porque esta não é julgada como necessária, posto que não há, no código, nenhuma referência explícita a esta classe. Ou seja, ela não é referenciada como parâmetro em lugar algum ou como retorno; apenas referências à classe `Livro` são feitas. Desta forma, não é necessário incluir esta classe no arquivo `jaxb.index`. Para “forçar” o mecanismo do JAXB a reconhecê-la, você precisa utilizar a anotação `javax.xml.bind.annotation.XmlSeeAlso`.

Esta anotação fará com que o JAXB detecte a classe “extra” e faça o mapeamento. Por exemplo, para fazer o mecanismo reconhecer a classe `EBook`, podemos inserir a anotação na classe `Livro`, desta forma:

```
package com.knight.estoque.modelos;

//imports omitidos

@XmlAccessorType(XmlAccessType.FIELD)
@XmlSeeAlso({ EBook.class })
public class Livro {
    // conteúdo da classe
}
```

Isto fará com que o JAXB (e, consequentemente, o JAX-WS) entenda a presença da classe `EBook`. Se você der uma olhada no XML Schema gerado, perceberá o seguinte trecho:

```
<!-- Código referente ao mapeamento da classe Livro -->
<xs:complexType name="livro">
```

```

<xs:sequence>
  <xs:element name="anoDePublicacao" type="xs:int" minOccurs="0"/>
  <xs:element name="autores" minOccurs="0">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="autor"
          type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="editora" type="xs:string" minOccurs="0"/>
  <xs:element name="nome" type="xs:string" minOccurs="0"/>
  <xs:element name="resumo" type="xs:string" minOccurs="0"/>
  <xs:element name="dataDeCriacao" type="xs:anySimpleType"
    minOccurs="0"/>
</xs:sequence>
</xs:complexType>

<!-- Código referente ao mapeamento da classe EBook -->
<xs:complexType name="eBook">
  <xs:complexContent>
    <xs:extension base="tns:livro">
      <xs:sequence>
        <xs:element name="formato" type="xs:string" minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

```

Note que o conteúdo do `complexType eBook` contém a *tag* `extension`. Esta *tag* representa, em um XML Schema, a herança de elementos de um supertipo — exatamente da forma como acontece em Java. A partir desta extensão, o XML Schema inseriu mais elementos, neste caso, o elemento `formato`.

Para realizar o tráfego destas informações, no entanto, o JAXB introduz mais sutileza. Suponha que o *e-book* de SOA Aplicado seja inserido na base de dados. Ao recuperar estas informações utilizando o SoapUI, o seguinte é retornado:

```

<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:listarLivrosResponse
      xmlns:ns2="http://servicos.estoque.knight.com/"

```

```

        xmlns:ns3="http://servicos.estoque.knight.com/excecoes/">
<!-- outros livros -->
<livro xsi:type="ns2:eBook"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <anoDePublicacao>2012</anoDePublicacao>
    <autores>
        <autor>Alexandre Saudate</autor>
    </autores>
    <editora>Casa do Código</editora>
    <nome>SOA Aplicado</nome>
    <resumo>Aprenda SOA de forma prática</resumo>
    <dataDeCriacao>2012-12-14T17:10:48</dataDeCriacao>
    <formato>PDF</formato>
    </livro>
</ns2:listarLivrosResponse>
</S:Body>
</S:Envelope>

```

Note a declaração do prefixo `xsi`, para o *namespace* <http://www.w3.org/2001/XMLSchema-instance>. Este *namespace* é padronizado pela especificação XML Schema para que seja possível realizar a diferenciação de tipos em tempo de execução. Note que este tipo precisa referenciar o que está previsto no próprio XML Schema, ou seja, faz referência ao `complex type` de nome `eBook`, e não à classe `EBook` em si.

3.8 TRABALHE COM ENUMS

Outro recurso interessante do JAXB é a possibilidade de se trabalhar com *enums*. Este é um recurso presente na linguagem Java para trabalhar com enumerações de valores de maneiras preestabelecidas. Por exemplo, já vimos como trabalhar com formatos de *e-books*. Suponha que queiramos limitar nossas opções e trabalhar apenas com três formatos preestabelecidos: PDF, MOBI e EPUB. Podemos modelar um *enum* para trabalhar com estes formatos:

```

package com.knight.estoque.modelos;

public enum FormatoArquivo {
    PDF, MOBI, EPUB;
}

```

Desta forma, alteramos a classe `EBook` para trabalhar com este formato:

```

package com.knight.estoque.modelos;

import java.util.List;

public class EBook extends Livro {

    private FormatoArquivo formato = FormatoArquivo.PDF;

    public EBook() {
        super();
    }

    public EBook(Integer anoDePublicacao, List<Autor> autores,
        String editora, String nome, String resumo) {
        super(anoDePublicacao, autores, editora, nome, resumo);
    }

    // getters e setters

}

```

Ao inicializar este serviço, a seguinte modificação na tipagem do `complexType eBook` terá sido realizada:

```

<xs:complexType name="eBook">
  <xs:complexContent>
    <xs:extension base="tns:livro">
      <xs:sequence>
        <xs:element name="formato" type="tns:formatoArquivo"
          minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

```

Note que, agora, o elemento `formato` faz referência a um novo tipo, `formatoArquivo`. Mais à frente, no mesmo XML Schema, está a definição desse tipo:

```

<xs:simpleType name="formatoArquivo">
  <xs:restriction base="xs:string">
    <xs:enumeration value="PDF"/>
  </xs:restriction>
</xs:simpleType>

```



```

        <xs:enumeration value="MOBI"/>
        <xs:enumeration value="EPUB"/>
    </xs:restriction>
</xs:simpleType>

```

Ainda podemos realizar a customização do nome do tipo utilizando a anotação `javax.xml.bind.annotation.XmlEnum`. Também podemos customizar os valores utilizando a anotação `javax.xml.bind.annotation.XmlEnumValue`. Por exemplo, suponha que queiramos que todos os valores da enumeração estejam em letras minúsculas. Podemos escrevê-la da seguinte maneira:

```

package com.knight.estoque.modelos;

//imports omitidos

@XmlEnum
public enum FormatoArquivo {

    @XmlEnumValue("pdf")
    PDF,

    @XmlEnumValue("mobi")
    MOBI,

    @XmlEnumValue("epub")
    EPUB;

}

```

Ao realizarmos esta modificação, o seguinte aparecerá no XML Schema:

```

<xs:simpleType name="formatoArquivo">
  <xs:restriction base="xs:string">
    <xs:enumeration value="pdf"/>
    <xs:enumeration value="mobi"/>
    <xs:enumeration value="epub"/>
  </xs:restriction>
</xs:simpleType>

```

Se quisermos alterar o nome deste `simple type`, podemos também utilizar a anotação `javax.xml.bind.annotation.XmlType`. Ela dá o poder de alterar as definições de quaisquer tipos — simples ou complexos. Para alterar o nome

do `simple type` `formatoArquivo`, por exemplo, podemos realizar a seguinte definição do *enum*:

```
@XmlEnum
@XmlType(name = "formato")
public enum FormatoArquivo {
    //valores presentes no enum
}
```

Desta forma, o tipo alterado fica:

```
<xs:simpleType name="formato">
  <xs:restriction base="xs:string">
    <xs:enumeration value="pdf"/>
    <xs:enumeration value="mobi"/>
    <xs:enumeration value="epub"/>
  </xs:restriction>
</xs:simpleType>
```

3.9 MODELE SUAS CLASSES COM COMPORTAMENTOS DE NEGÓCIO E MANTENHA-AS MAPEADAS COM JAXB

Eric Evans, em seu livro *Domain-Driven Design* [5], e Martin Fowler em seu artigo sobre *Anemic Domain Model* [7] ressaltam a importância de criar uma camada relevante de classes, que modelam de forma adequada o negócio em uma aplicação. Manter classes que apenas mantenham dados, com *getters* e *setters* e nada mais, faz com que o sistema fique engessado, incapaz de ser responsivo a mudanças.

Quando desenvolvemos *web services* em Java e utilizamos a abordagem de geração do WSDL a partir do código (abordagem conhecida como *code first*), podemos (e devemos!) tirar proveito de técnicas de mapeamento avançadas de JAXB, de maneira que podemos modelar o comportamento de negócio nas próprias classes que utilizamos para trafegar nossos dados em XML.

RESTRIÇÕES EM RELAÇÃO A COMPORTAMENTO DE NEGÓCIO E WEB SERVICES

Aplicar comportamento de negócios em *web services* pode ser maravilhoso; no entanto, deve-se tomar muito cuidado com o modelo de desenvolvimento. Se você for criar as classes a partir de XML Schemas, você provavelmente perderá o código adicionado. Portanto, tenha certeza de que você está desenvolvendo utilizando o modelo *code first* ou alguma ferramenta que mantenha código preexistente (não é o caso do `wsimport`).

Alguns dos principais problemas relacionados a esse tipo de desenvolvimento estão ligados ao tráfego indesejado de informações de/para os clientes. Por exemplo, suponha que queiramos modelar uma classe `Autor`:

```
public class Autor {  
  
    private String nome;  
    private Date dataNascimento;  
  
    // getters e setters  
}
```

Portanto, parece lógico que queiramos modificar nossa classe `Livro` para trabalhar com essa classe nova:

```
@XmlAccessorType(XmlAccessType.FIELD)  
public class Livro {  
  
    @XmlElementWrapper(name = "autores")  
    @XmlElement(name = "autor")  
    private List<Autor> autores;  
    //private List<String> autores;  
  
    // outros campos  
}
```

No entanto, nós já possuíamos antes o mapeamento para uma lista de `Strings`, e não gostaríamos de substituir o mapeamento antigo. Assim, podemos criar um mapeamento para converter somente este campo:

```

public static class AdaptadorAutores extends XmlAdapter<String, Autor> {

    @Override
    public String marshal(Autor autor) throws Exception {
        return autor.getNome();
    }

    @Override
    public Autor unmarshal(String autor) throws Exception {
        return new Autor(autor, null);
    }

}

```

Agora, basta que modifiquemos a classe `Livro` para que o novo mapeamento esteja completo:

```

@XmlElementWrapper(name = "autores")
@XmlElement(name = "autor")
@XmlJavaTypeAdapter(value = AdaptadorAutores.class)
private List<Autor> autores;

```

Desta forma, repare que o XML Schema não será alterado, pois o JAXB detecta os adaptadores e interpreta as diferenças (ou seja, para todos os efeitos, é como se tivéssemos mantido o atributo `autores` como `List<String>`).

Mas e se, por acaso, não quiséssemos trafegar, nunca, o atributo `dataNascimento` da classe `Autor`? É claro, poderíamos anotar a classe com `@XmlAccessorType(XmlAccessType.NONE)` e, então, anotar os campos que gostaríamos que fossem mapeados com `@XmlElement` ou `@XmlAttribute`:

```

@XmlAccessorType(XmlAccessType.NONE)
public class Autor {

    @XmlElement(name="nome")
    private String nome;
    private Date dataNascimento;
}

```

Contudo, uma alternativa que seria, certamente, mais inteligente nesse caso é utilizar a anotação `@XmlTransient`:

```
public class Autor {  
  
    private String nome;  
  
    @XmlTransient  
    private Date dataNascimento;  
  
}
```

Desta forma, o campo `dataNascimento` será ignorado em arquivos `xsd`, e caso queiramos alterar qualquer coisa na classe `Autor`, estas mudanças ficarão sempre bem mais fáceis.

Com esses recursos, você pode colocar comportamentos orientados a negócios na sua própria classe, sem problemas com o tráfego de informações! Considere a classe `Autor`:

```
public class Autor {  
  
    private String nome;  
    private Date dataNascimento;  
  
    public Autor() {  
    }  
  
    public Autor(String nome, Date dataNascimento) {  
        super();  
        this.nome = nome;  
        this.dataNascimento = dataNascimento;  
    }  
  
    public List<URL> getRefs() throws HttpException, IOException {  
        // Realiza uma busca no Google pelas referências àquele autor.  
    }  
  
    // getters e setters  
}
```

A classe `Autor` não possui nenhuma informação a respeito de mapeamento JAXB. Neste caso, a configuração padrão é utilizar todos os pares de *getters* e *setters* e campos públicos. No entanto, o método `getRefs` não possui um equivalente (ou seja, não existe um método `setRefs`). Assim, o JAXB não irá considerar este

campo no tráfego de informações, e se criarmos um serviço de listagem de autores, este método será desconsiderado. Tente criar o serviço:

```
@WebService
public class AutoresService {

    public List<Autor> listarAutores() {
        Autor adrianoAlmeida = new Autor("Adriano Almeida", new Date());
        Autor pauloSilveira = new Autor("Paulo Silveira", new Date());
        Autor viniciusBaggio = new Autor("Vinicius Baggio", new Date());
        return new ArrayList<>(Arrays.asList(adrianoAlmeida, pauloSilveira,
            viniciusBaggio));
    }

    public static void main(String[] args) {
        Endpoint.publish("http://localhost:8080/autores",
            new AutoresService());
    }
}
```

Quando invocamos este código com o SoapUI, obtemos o seguinte retorno:

```
<ns2:listarAutoresResponse
  xmlns:ns2="http://servicos.estoque.knight.com/">
  <return>
    <dataNascimento>2012-10-06T12:11:59</dataNascimento>
    <nome>Adriano Almeida</nome>
  </return>
  <return>
    <dataNascimento>2012-10-06T12:11:59</dataNascimento>
    <nome>Paulo Silveira</nome>
  </return>
  <return>
    <dataNascimento>2012-10-06T12:11:59</dataNascimento>
    <nome>Vinicius Baggio Fuentes</nome>
  </return>
</ns2:listarAutoresResponse>
```

E ainda, se mudarmos de ideia, podemos modificar o método `getRefs` para ser mapeado, assim:

```
@XmlElementWrapper(name = "refs")
@XmlElement(name = "ref")
public List<URL> getRefs() throws HttpException, IOException
```

Dessa forma, se reiniciarmos o nosso serviço e refizermos a requisição, ela virá assim:

```
<return>
  <dataNascimento>2012-10-06T12:26:39</dataNascimento>
  <nome>Adriano Almeida</nome>
  <refs>
    <ref>https://twitter.com/ADRIANOalmeida</ref>
    <ref>http://twitter.com/adriano_</ref>
    <ref>http://www.crmlc.com/people_research.aspx</ref>
    <ref>http://en-gb.facebook.com/adriano.almeida.370177</ref>
  </refs>
</return>
```

O MAPEAMENTO DE MÉTODOS COM JAXB

O JAXB só possui a habilidade de mapear métodos que são *getters* e *setters*. No nosso caso, foi possível instalar o mapeamento devido ao fato de que o método `getRefs` pode ser considerado um *getter*. Se o nome deste método não começasse com `get`, não seria possível mapeá-lo para JAXB — no entanto, ainda seria perfeitamente possível mantê-lo na classe mapeada.

3.10 SUMÁRIO

Neste capítulo, você viu modos avançados de fazer mapeamentos utilizando JAX-WS e JAXB. Viu várias maneiras de fazer/refazer mapeamentos utilizando estas tecnologias, e viu como aliar estes conhecimentos a modos elegantes de modelar suas classes utilizando *Domain-Driven Design*.

No entanto, ainda está restando algo. No próximo capítulo, você verá como reunir todos estes conhecimentos para criar aplicações robustas, capazes de serem responsivas a diversos cenários de utilização em empresas. Vamos em frente?

CAPÍTULO 4

Embarcando no Enterprise - Application Servers

“Toda empresa precisa ter gente que erra, que não tem medo de errar e que aprende com o erro.”

– Bill Gates

Ao final da criação do seu modelo de autores, você decidiu criar mais um serviço, `AutoresService`:

```
package com.knight.estoque.servicos;

// imports omitidos

@WebService
public class AutoresService {

    public List<Autor> listarAutores() {
```

```
Autor adrianoAlmeida = new Autor("Adriano Almeida", new Date());
Autor pauloSilveira = new Autor("Paulo Silveira", new Date());
Autor viniciusBaggio = new Autor("Vinicius Baggio", new Date());
return new ArrayList<>(Arrays.asList(adrianoAlmeida, pauloSilveira,
    viniciusBaggio));
}

public static void main(String[] args) {
    Endpoint.publish("http://localhost:8080/autores",
        new AutoresService());
}
}
```

Mas quando você tenta inicializar este serviço em conjunto com o de listagem de livros, você tem a seguinte mensagem:

```
java.net.BindException: Address already in use
```

Basicamente, a mensagem significa que o método `publish`, da classe `Endpoint`, não é capaz de publicar mais de um serviço na mesma porta. É possível utilizar essa classe para publicar mais de um *web service*; no entanto, com o número de serviços crescendo, em diferentes partes do sistema, começa a ser complicado gerenciar tudo. O que fazer, então?

4.1 COMO USAR UM SERVLET CONTAINER - JETTY

Esse problema acontece quando tentamos inicializar um servidor HTTP numa porta que já está sendo utilizada. Entretanto, é possível modificar esta abordagem se for possível utilizar um servidor HTTP já inicializado que tenha a capacidade de redirecionar requisições para os *web services*.

Você leu que a melhor maneira de realizar esse gerenciamento é através de um *application server*, ou seja, um sistema que irá efetuar o gerenciamento de suas aplicações de maneira uniforme. Você faz uma pesquisa e descobre o `Jetty`, um *servlet container* (que é como se fosse um modelo simplificado de *application server*) realmente muito simples e leve, que deve servir bem às suas necessidades. No entanto, é preciso fazer algumas alterações no seu projeto para utilizá-lo.

O Java possui diversos modelos de arquivos para serem instalados em um *application server*, como o `EAR` (*Enterprise Archive*), o `ejb-jar` e o `WAR` (*Web Application Archive*). Você precisa transformar o seu projeto em `WAR` para instalá-lo no `Jetty`. Isso requer um diretório `WEB-INF`, na raiz do seu projeto, contendo um arquivo chamado `web.xml`.

A próxima alteração a ser feita é adicionar uma implementação JAX-WS ao seu *classpath*. Até o momento da escrita deste livro, existem várias implementações (`JBossWS`, `Metro`, `Apache CXF`, a implementação de referência, e outros). Por questão de simplicidade, você decide colocar no *classpath* o JAX-WS RI — já que, na realidade, quando utilizamos a classe `Endpoint`, estávamos utilizando esta implementação.

O CÓDIGO FONTE DO LIVRO

O código fonte deste livro (disponível em <https://github.com/alesaudate/soa>) está utilizando, como sistema de construção, um framework Java chamado Maven. Ele é utilizado para simplificar a adição/remoção de bibliotecas do *classpath*, e deve ser de grande utilidade na construção de aplicações Java, sejam elas do tipo JAR, WAR, ou EAR.

A adição do JAX-WS no *classpath* pode ser feita apenas adicionando uma entrada no arquivo de gerenciamento do Maven, chamado `pom.xml`.

O `web.xml` deve ter o seguinte conteúdo para habilitar a implementação:

```
<web-app>
  <listener>
    <listener-class>
      com.sun.xml.ws.transport.http.servlet.WSServletContextListener
    </listener-class>
  </listener>

  <servlet>
    <servlet-name>jaxws-servlet</servlet-name>
    <servlet-class>
      com.sun.xml.ws.transport.http.servlet.WSServlet
    </servlet-class>
  </servlet>
```

```
<servlet-mapping>
  <servlet-name>jaxws-servlet</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>
</web-app>
```

Este arquivo é responsável por dizer ao Jetty que ele deve inicializar o mecanismo do JAX-WS durante a inicialização do projeto. Isso requer um arquivo chamado `sun-jaxws.xml`, devidamente colocado na pasta `WEB-INF`. Este arquivo, por enquanto, deve ter o seguinte conteúdo:

```
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime"
  version="2.0">
  <endpoint name="livros"
    implementation="com.knight.estoque.servicos.ListagemLivros"
    url-pattern="/livros" />
  <endpoint name="autores"
    implementation="com.knight.estoque.servicos.AutoresService"
    url-pattern="/autores" />
</endpoints>
```

Este XML indica para a *engine* do JAX-WS RI que a classe `com.knight.estoque.servicos.LivrosService` deve ser mapeada para a URL (relativa) `/livros`, e a classe `com.knight.estoque.servicos.AutoresService`, para a URL `/autores`. Isto vai inicializar a *engine* corretamente, e ele irá procurar sempre pelo arquivo `sun-jaxws.xml` na pasta `WEB-INF`.

O próximo passo é inicializar o Jetty e navegar até as URLs dos WSDLs (<http://localhost:8080/livros?wsdl> para o serviço de livros e <http://localhost:8080/autores?wsdl> para o serviço de autores) para conferir se os dois foram criados corretamente.

UTILIZANDO O MAVEN PARA INICIALIZAR O JETTY

Uma das capacidades do Maven é facilitar o processo de instalação da aplicação Java em um *container* como o Jetty. Se você checar o código-fonte do livro (disponível em <https://github.com/alesaudate/soa>), verá que o conteúdo desse capítulo pode ser inicializado utilizando o comando `mvn jetty:run`.

4.2 INTRODUÇÃO A EJBs

Conforme sua aplicação vai crescendo, você sente a necessidade de ferramental mais completo. Por exemplo, a aplicação ainda não possui um bom gerenciamento de transações em banco de dados. A comunicação entre as próprias aplicações Java também ainda é problemática, pois existem outras aplicações Java utilizando a aplicação de gerenciamento de estoques, mas não parece “certo” que as mesmas se comuniquem via *web services* afinal, sendo a mesma linguagem, por que utilizar tradução de/para XML?

Você pesquisa a respeito do assunto quando descobre a especificação de *Enterprise JavaBeans* (EJB). Você aprende que pode adicionar não só isso, mas como desenvolver o *front-end* da aplicação utilizando a tecnologia *JavaServer Faces* (JSF), que se integra bem a EJBs. No entanto, existe um problema: o Jetty não suporta EJBs. O que fazer?

Fazendo uma pesquisa entre os *application servers* que suportam essa tecnologia, você tem várias opções: Oracle WebLogic, IBM Websphere, Glassfish e outros. Mas o que mais lhe agrada é o JBoss Application Server, por ser *open source*, fácil para usar e rápido para desenvolver (a última versão do JBoss Application Server pode ser inicializada em pouquíssimos segundos).

Antes de começar, você faz uma pesquisa para descobrir quais são os tipos de EJBs disponíveis:

- *Beans* de sessão com informação de estado;
- *Beans* de sessão sem informação de estado;
- *Beans* de entidades;
- *Beans* orientados a mensagens.

Os únicos que são adequados a *web services* são os *beans* com e sem informações de estado. Qual escolher?

Neste caso, um dos princípios de orientação a serviços que vêm à tona é o da falta de manutenção de estado. Este princípio diz que um serviço não deve, nunca, manter informações a respeito do estado. Em outras palavras, uma requisição N nunca pode ser dependente de uma requisição anterior. O que faz dos *beans* sem informação de estado a escolha adequada para *web services*.

Você decide fazer o teste transformando uma das classes de *web services* do seu projeto em EJB (que, na prática, consiste apenas de anotar a classe com `@Stateless`):

```
package com.knight.estoque.servicos;

import javax.ejb.Stateless;

//Outros imports excluídos

@WebService
@Stateless
public class LivrosService {

    // métodos do serviço

}
```

MAVEN E A TRANSIÇÃO ENTRE JETTY E JBOSS APPLICATION SERVER

Se você comparar o código fonte da aplicação, verá que existe uma diferença enorme entre os arquivos de infraestrutura do projeto feito para Jetty e do projeto feito para JBoss Application Server.

Quando utilizamos Jboss AS, não é necessário mais utilizar o arquivo `sun-jaxws.xml` (já que o JBoss AS faz descoberta dos recursos que devem ser expostos como *web services*), nem o arquivo `web.xml`. Também é necessário alterar o *plugin* de *deploy* e o gerenciamento de dependências, já que estamos trabalhando com JBoss AS e EJBs.

Agora, você só precisa inicializar o JBoss e, como você já deve ter o plugin do Maven instalado, utilizar o comando `mvn jboss-as:deploy`. Uma vez realizado o *deploy*, você enxerga no console da sua IDE o seguinte:

```
21:04:47,380 INFO [org.jboss.wsf.stack.cxf.metadata.MetadataBuilder]
id=com.knight.estoque.servicos.AutoresService
address=http://localhost:8080/soa-cap04-0.0.1-SNAPSHOT/AutoresService
implementor=com.knight.estoque.servicos.AutoresService
```

```
invoker=org.jboss.wsf.stack.cxf.JBossWSInvoker
serviceName={http://servicos.estoque.knight.com/}AutoresServiceService
portName={http://servicos.estoque.knight.com/}AutoresServicePort
wsdlLocation=null
mtomEnabled=false
21:04:47,409 INFO [org.jboss.wsf.stack.cxf.metadata.MetadataBuilder]
id=LivrosService
address=http://localhost:8080/soa-cap04-0.0.1-SNAPSHOT/LivrosService
implementor=com.knight.estoque.servicos.LivrosService
invoker=org.jboss.wsf.stack.cxf.JBossWSInvoker
serviceName={http://servicos.estoque.knight.com/}LivrosServiceService
portName={http://servicos.estoque.knight.com/}LivrosServicePort
wsdlLocation=null
mtomEnabled=false
```

Este texto é a “prova” de que seus serviços estão corretamente instalados. Você pode checar se eles foram instalados corretamente indo até o console de administração do JBoss, localizado em <http://localhost:9990/console> (caso seja a primeira vez que você acessa o console, vai precisar criar um usuário de gerenciamento para vê-lo. Para isso, siga as instruções da página de erro que vai aparecer ao invés do console).

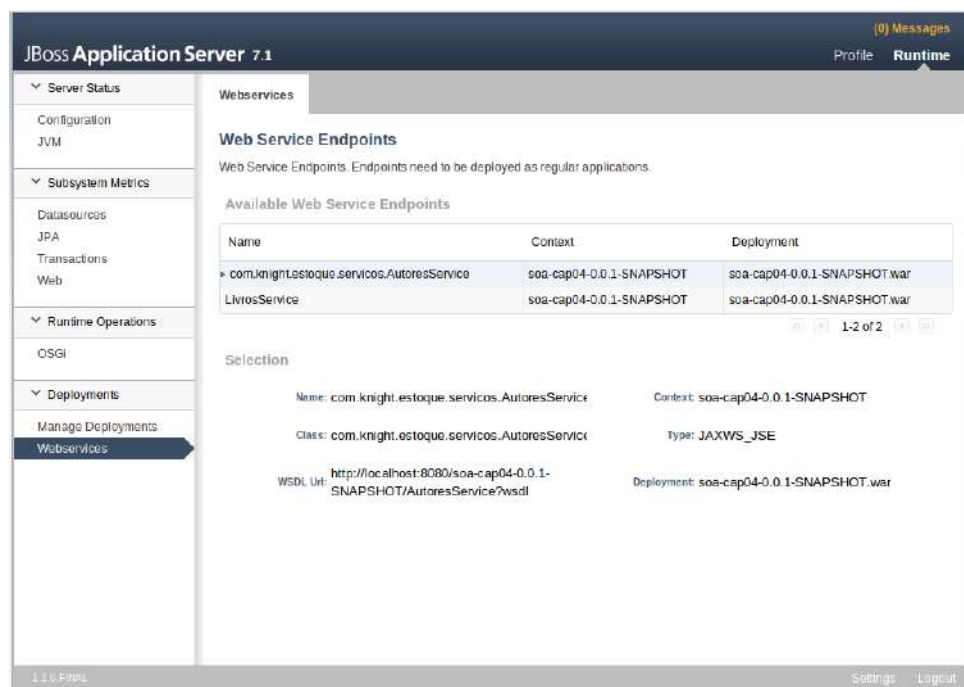


Figura 4.1: Console do JBoss com os serviços instalados

Ao explorar o console do JBoss é que você se dá conta do poder de gerenciabilidade que é conferido à sua aplicação: é possível alterar a porta em que seus *web services* serão disponibilizados, configurar uma porta segura (com HTTPS) para disponibilização, alterar o *hostname* que será utilizado para oferecer esses serviços e assim por diante. Além disso, também é possível criar *datasources* (ou seja, serviços de persistência), serviços de *cache* distribuído, transacionalidade e muito mais recursos.

Ao configurar seus serviços como EJBs, você está dando a eles a capacidade de serem gerenciáveis dessa forma. Isso facilitará a configuração de bancos de dados, transações, *caches* e outras funcionalidades requeridas em ambientes de desenvolvimento de empresas. Isso também facilitará a transição entre ambientes de desenvolvimento, homologação e produção.

4.3 HABILITANDO PERSISTÊNCIA E TRANSACIONALIDADE

Até então, a manutenção do estado do banco de dados é feita através de classes Java, sem haver uma persistência efetiva. Para habilitar persistência em banco de da-

dos, é necessário modificar as entidades do serviço, transformando-os em `Entity Beans` (ou seja, EJBs). Para isto, basta anotá-los com `@Entity`, definir uma identidade (do ponto de vista do banco de dados) e adicionar quais informações de mapeamento em relação a outras entidades que forem necessárias. Por exemplo, a classe `Autor` passa a ficar assim:

```
package com.knight.estoque.modelos;

import javax.persistence.*;

//outros imports omitidos

@Entity
public class Autor {

    @Id
    @GeneratedValue(
        strategy = GenerationType.IDENTITY)
    private Long id;
    private String nome;
    private Date dataNascimento;

    // O método refs, assim como getters e setters para os campos
}
```

Note que é a mesma classe utilizada nos serviços, apenas modificada para persistência em banco de dados. Em relação ao serviço de autores, basta incluir uma referência ao gerenciador de banco de dados (representado pela interface `EntityManager`), e anotá-la com `@PersistenceContext` para que o JBoss inclua as informações de persistência em banco de dados:

```
@WebService
@Stateless
public class AutoresService {

    @PersistenceContext
    private EntityManager em;

    public List<Autor> listarAutores() {
```

```

        return em.createQuery("select a from Autor a", Autor.class)
            .getResultList();
    }
}

```

Para completar essa configuração, é necessário adicionar um arquivo de especificação das informações de mapeamento destes objetos para o banco, chamado `persistence.xml`; e, ainda, um arquivo para descrição da conexão com o banco de dados em si (o conector em si é chamado de `datasource`).

O arquivo de persistência pode ser configurado como o seguinte:

```

<persistence version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="primario">
    <jta-data-source>java:jboss/datasources/KnightDS</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="create-drop" />
      <property name="hibernate.show_sql" value="true" />
    </properties>
  </persistence-unit>
</persistence>

```

Não se preocupe com as configurações gerais. Por enquanto, é importante notar apenas o elemento `jta-data-source`, que representa, do ponto de vista do *application server*, qual `datasource` deve ser utilizado. Esse `datasource` pode ser criado a partir de um arquivo chamado, no nosso exemplo, de `knight-estoque-ds.xml`:

```

<datasources xmlns="http://www.jboss.org/ironjacamar/schema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.jboss.org/ironjacamar/schema
    http://docs.jboss.org/ironjacamar/schema/datasources_1_0.xsd">
  <datasource jndi-name="java:jboss/datasources/KnightDS"
    pool-name="knight-datasource" enabled="true"
    use-java-context="true">
    <connection-url>
      jdbc:h2:mem:knight-estoque;DB_CLOSE_ON_EXIT=FALSE
    </connection-url>
  </datasource>
</datasources>

```

```
</connection-url>
<driver>h2</driver>
<security>
  <user-name>sa</user-name>
  <password>sa</password>
</security>
</datasource>
</datasources>
```

Note que este arquivo possui uma configuração específica para funcionamento com o JBoss, e não irá funcionar em outros *application servers*. Ele é útil, no entanto, para realizar transições entre ambientes de desenvolvimento, homologação e produção, já que existe a possibilidade de criar o `datasource` a partir do console de gerenciamento do JBoss, e não dentro da aplicação, como estamos fazendo.

COMO IR A FUNDO NA JPA

A JPA é uma especificação poderosíssima pra integrar aplicações Java com bancos de dados, fazendo o papel de um biblioteca que faz o mapeamento objeto relacional. Dessa forma, permite que você trabalhe orientado o objetos e abstraia toda a complexidade necessária para se integrar com bancos de dados.

Para aprender mais sobre essa maravilhosa especificação e suas implementações, como o Hibernate, recomendo a leitura do livro “Aplicações Java para web com JSF e JPA”, disponível pela Editora Casa do Código.

4.4 UM NOVO SISTEMA

A diretoria está apreciando seu trabalho com o sistema de estoques. No entanto, o sistema de criação de livros ainda é deficiente. Aos olhos da diretoria técnica, resta criar um gerenciamento avançado de usuários. É necessário criar um novo sistema para isso.

SOA tem algumas das boas práticas da programação orientada a objetos, como a alta coesão e o baixo acoplamento. Deste ponto de vista, é excelente criar uma nova aplicação para gerenciamento de usuários, já que a mesma vai oferecer chances de reusabilidade da lógica (visto que vários sistemas distintos podem utilizar um cadastro centralizado de usuários). Além disso, vai promover o baixo acoplamento

entre os serviços e o “acobertamento” de detalhes de usuários do ponto de vista de todas as aplicações (dado que, ao utilizar um *web service* para fazer o gerenciamento de usuários, qualquer sistema pode ser utilizado — LDAP, Banco de dados, outros *web services* — e, ainda assim, o cliente irá enxergar o mesmo *web service*, mesmo que o sistema mude). Ou seja, a partir dos princípios de orientação a serviços que você viu no capítulo 1, os serviços deste novo sistema atenderão:

- Baixo acoplamento;
- Abstração;
- Reutilização;
- Habilidade de poderem ser recompostos;

SOA e o baixo acoplamento

Quando utilizamos SOA (e principalmente quando nossos *web services* estão funcionando sobre HTTP), podemos recorrer a capacidades de elasticidade da nossa aplicação. Por exemplo, suponha que, num cenário de **produção**, o sistema de gerenciamento de usuários esteja hospedado no mesmo servidor de aplicação que o sistema de estoque, com um servidor de balanceamento de carga intermediando as requisições do cliente até o servidor. Ou seja, um servidor unificado para repassar requisições do cliente para vários *application servers* que hospedam, de fato, as aplicações. O *layout* geral do sistema ficaria assim:

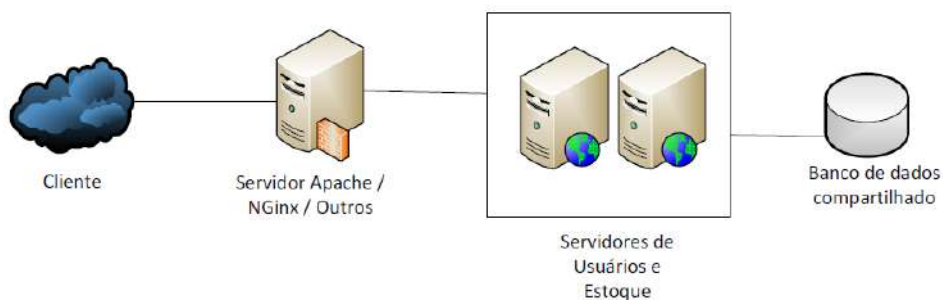


Figura 4.2: Estrutura simplificada dos serviços

No entanto, considere que a carga dos serviços aumente e que a carga do serviço de usuários cresça muito. É possível tomar as seguintes ações:

- Colocar o serviço de usuários em *application servers* separados;
- Colocar servidores de *cache* para que nem todas as requisições sejam feitas diretamente aos *application servers*;
- Modificar a estrutura de armazenamento: de banco de dados para LDAP;

Com essas modificações, o *layout* geral do sistema ficaria assim:

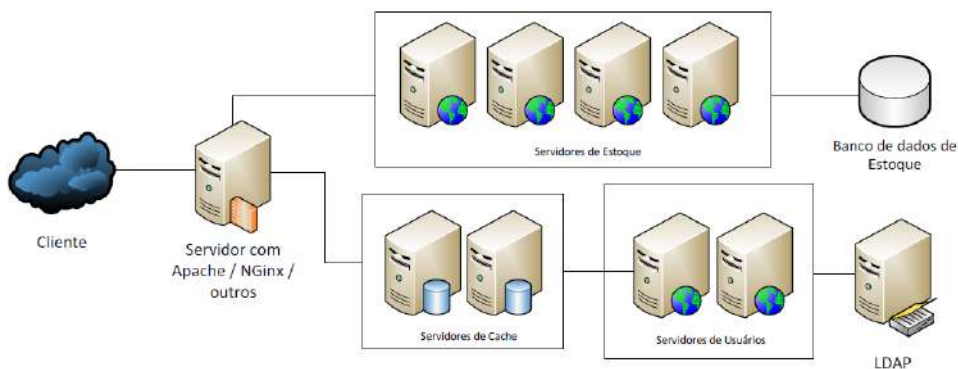


Figura 4.3: Estrutura separada dos serviços

Note que, do ponto de vista do cliente, não há diferença alguma. Ainda assim, a separação de regras de negócio em serviços diferentes nos dá a chance de redimensionar a infraestrutura necessária para atender a estes diferentes aspectos de negócio, sem que os clientes dos nossos serviços tenham quaisquer modificações para isso.

Note que isso não seria possível se nossos serviços fossem fortemente acoplados, ou mesmo interdependentes entre si. Neste ponto, um aspecto muito importante de SOA vem à tona: a não-manutenção de estado por parte dos serviços. Se estes serviços mantivessem estado, não seria tão simples redimensionar a capacidade de atendimento deles, já que cada servidor teria que “conhecer” o estado dos seus vizinhos. Como não é o caso, podemos adicionar ou retirar máquinas à vontade de cada grupo de serviços (esta capacidade é conhecida como *escalabilidade horizontal*).

Mas ainda assim, falta algo...

Desenvolver o novo serviço de gerenciamento de usuários, a essa altura, lhe parece simples. Porém, restam algumas questões a serem respondidas: será que não existe nada, na especificação de *web services*, que promova o gerenciamento de segurança a partir de um modelo universal (posto que esta é uma questão recorrente em desenvolvimento de *software*)?

Você pesquisa a respeito e encontra a especificação *WS-Security*. Parece bom, já que é um modelo padronizado, e não deve dar muitos problemas para colocar no projeto. No entanto, ao fazer uma busca na internet a respeito de como implementar esse modelo, você descobre que o *application server* que você está utilizando (JBoss) requer o desenvolvimento do WSDL **antes** do código (técnica conhecida como desenvolvimento *contract-first*).

A questão tem ainda alguns agravantes: a diretoria coloca um novo requisito, que é o desenvolvimento de uma camada visual para gerenciamento destes serviços. Mas lhe parece ainda meio complicado interagir com esses serviços a partir de uma camada visual, e não soa uma boa ideia interagir com a sua aplicação a partir do modelo tradicional, já que você quer ter certeza de que seus *web services* são plenamente funcionais do ponto de vista do cliente. Ao pesquisar sobre isso, você se depara com um modelo de desenvolvimento mais ágil deste ponto de vista, o *REpresentational State Transfer* (REST).

No decorrer dos próximos capítulos, você verá como dominar estes assuntos, de maneira que você será capaz de gerenciar da melhor maneira possível cada uma destas questões. Vamos em frente?

4.5 SUMÁRIO

Neste capítulo, você conheceu o Jetty, um *servlet container* extremamente ágil, para realizar desenvolvimento de serviços de maneira muitíssimo eficiente. Porém, você viu que, quando se trata de questões mais complexas, pode ser muito útil converter seus *web services* em EJBs. Para isso, você utiliza um *application server* que, além de ágil, também é capaz de endereçar estas questões: o JBoss Application Server.

Você também viu como SOA é utilizado para endereçar questões de escalabilidade horizontal, com redirecionamento de requisições HTTP para múltiplos *application servers*, e como estas habilidades são utilizadas para “esconder” do cliente detalhes de implementação, de maneira que este não seja afetado por modificações internas de infraestrutura.

Ao final, você pôde perceber que questões mais complexas de desenvolvimento ainda não foram respondidas: qual a melhor maneira de integrar segurança nas aplicações? Como manter a estabilidade de um cenário com **múltiplas** aplicações? E como resolver a questão de comunicação direta de uma interface visual com seus *web services*?

Nos próximos capítulos, você verá como endereçar cada uma destas questões. Você passará a ter contato com ambientes heterogêneos, multiprotocolo e aprenderá a interligar estas operações da maneira mais eficiente possível.

CAPÍTULO 5

Desenvolva aplicações para a web com REST

“Construímos muitos muros e poucas pontes”

– Sir Isaac Newton

Com a necessidade de construção de um novo sistema, dessa vez de usuários, você decide reaproveitar o que viu até aqui, já que você já sabe que esse é um sistema que será utilizado por diversas outras aplicações dentro da knight.com. No entanto, dessa vez o requisito mudou um pouco: a diretoria pediu uma interface de administração *web*, de maneira que seja possível criar, atualizar e desabilitar usuários pela interface gráfica do sistema.

Contudo, esse já se tornou um desafio diferente daquilo a que você está acostumado. Seus companheiros especialistas em interface gráfica e UX (*User Experience*) falaram que pode ser complicado desenvolver sistemas da maneira tradicional e usando *web services*. Isso porque, de acordo com a especificação Java EE 6, já é

possível desenvolver interfaces utilizando EJBs, da maneira como você viu anteriormente, e utilizando `JavaServer Faces` para se comunicar com EJBs. Porém, esta especificação é mais voltada ao desenvolvimento de interfaces pelos próprios desenvolvedores, e você gostaria de fazer algo com bom apelo visual — algo que só pode ser feito por um especialista em interfaces com usuários, não desenvolvedores.

Você gostaria, então, de apresentar ao pessoal de design suas ideias de comunicação com *web services* SOAP; porém, para que eles realizem essa comunicação, eles precisariam de novas bibliotecas para comunicação com os serviços. Ao conversar com seu pessoal, você vê que eles estão acostumados com uma biblioteca chamada `jQuery`, que, por sua vez, tem grande capacidade de comunicação com serviços REST.

5.1 O QUE É REST?

REST é uma sigla que significa *Representational State Transfer*. REST é um modelo arquitetural concebido por um dos autores do protocolo HTTP (o doutor Roy Fielding), e tem como plataforma justamente as capacidades do protocolo, onde se destacam:

- Diferentes **métodos** (ou **verbos**) de comunicação (`GET`, `POST`, `PUT`, `DELETE`, `HEAD`, `OPTIONS`);
- Utilização de *headers* HTTP (tanto padronizados quanto customizados);
- Definição de arquivos como **recursos** (ou seja, cada um com seu próprio endereço);
- Utilização de *media types*.

Você pode notar que a própria *web* é baseada nesses princípios, e quando você navega por uma página, automaticamente está utilizando esses conceitos.

Por exemplo, ao realizar uma requisição para o site <http://www.casadocodigo.com.br/> pelo seu *browser*, na verdade você está realizando uma requisição para o recurso <http://www.casadocodigo.com.br/index.html>, utilizando o método `GET`. Ao realizar esta requisição, o conteúdo do arquivo `index.html` será carregado. Este arquivo é carregado com o *media type* `text/html`, o que indica ao *browser* que este é um arquivo do tipo HTML.

Por ser um HTML, o *browser* sabe que deve procurar referências por outros arquivos, como folha de estilos, *scripts* e imagens. As imagens, especificamente, são referenciadas pela *tag* `img`, que dá indicações ao *browser* de onde estão essas imagens propriamente ditas. Ao puxar essas indicações, o *browser* faz novas requisições, utilizando o método `GET`. Desta vez, as imagens propriamente ditas são retornadas com *media types* de acordo com o tipo da imagem (JPEG, GIF, PNG, BMP etc.), que indicam como as renderizar apropriadamente.

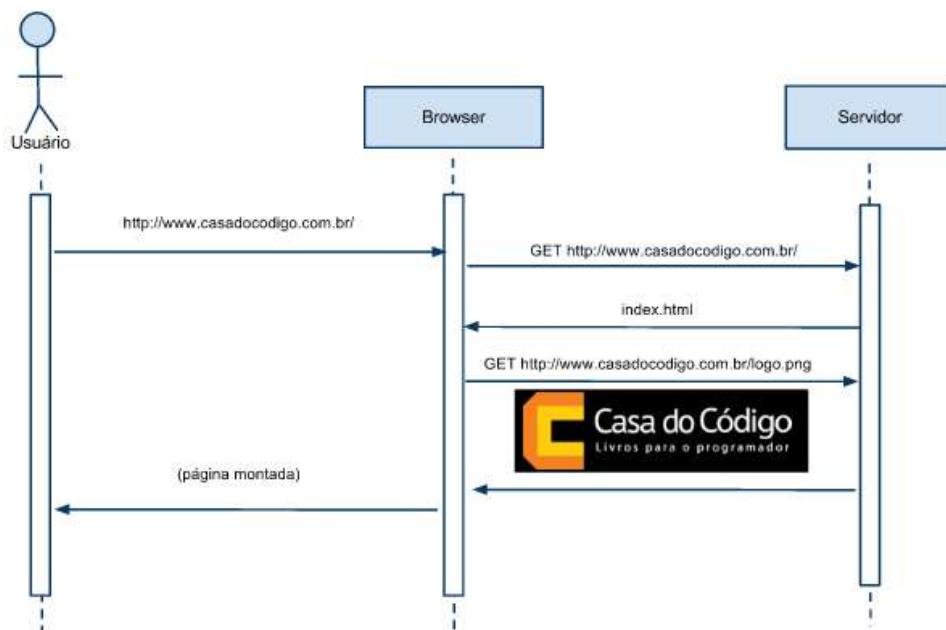


Figura 5.1: Interação entre o usuário, o browser e o servidor

A partir desses conceitos, é possível visualizar o mecanismo de funcionamento de uma possível aplicação baseada em REST: a partir de uma requisição para uma URL predefinida, o conteúdo retornado pode ser pré-acordado entre o cliente e o servidor e, utilizando indicações presentes no resultado, o cliente pode carregar dados novos.

5.2 ENTENDA O HTTP

Para compreender corretamente como utilizar REST, no entanto, deve-se notar que a chave do sucesso da técnica está nos padrões e as capacidades do protocolo HTTP. Começando pelos métodos utilizados, note que cada um possui uma finalidade diferente. O método `GET`, por exemplo, é utilizado para carregar recursos existentes.

Quando você faz uma requisição `GET` para <http://www.casadocodigo.com.br/index.html>, você está carregando o recurso localmente. Da mesma forma, suponha que você faça uma requisição para <http://www.knight.com/usuarios>. Esta requisição poderia te trazer como resultado, digamos, um XML com o seguinte formato:

```
<usuarios>
  <usuario>
    <id>1</id>
    <nome>Alexandre</nome>
    <sobrenome>Saudate</sobrenome>
    <login>alesaudate</login>
  </usuario>
  <usuario>
    <id>2</id>
    <nome>Adriano</nome>
    <sobrenome>Almeida</sobrenome>
    <login>adrianoalmeida7</login>
  </usuario>
  <usuario>
    <id>3</id>
    <nome>Paulo</nome>
    <sobrenome>Silveira</sobrenome>
    <login>paulo_caelum</login>
  </usuario>
</usuarios>
```

O formato HTTP da requisição seria assim:

```
GET /usuarios HTTP/1.1
Host: www.knight.com
Accept: application/xml
```

Lembre-se de que, na realidade, muitos outros cabeçalhos são enviados. Estes são apenas os mais básicos.

A resposta da nossa requisição ficaria assim:

```
HTTP/1.1 200 OK
Date: Sun, 14 Oct 2012 01:21:35 GMT
Last-Modified: Thu, 11 Oct 2012 20:40:32 GMT
Content-Length: 450
Content-Type: application/xml; charset=UTF-8
```

```
<usuarios>
<!-- conteúdo -->
</usuarios>
```

Note também que é possível diminuir o conjunto retornado através da própria requisição. Por exemplo, se quisermos apenas o usuário de ID 1, podemos fazer uma requisição para <http://www.knight.com/usuarios/1>. Desta maneira, o nosso resultado seria apenas o seguinte:

```
<usuario>
  <id>1</id>
  <nome>Alexandre</nome>
  <sobrenome>Saudate</sobrenome>
  <login>alesaudate</login>
</usuario>
```

Da mesma forma que utilizamos o método `GET` para recuperar os recursos existentes, podemos utilizar o método `POST` para realizar a criação de novos recursos. Assim, podemos fazer a seguinte requisição:

```
POST /usuarios HTTP/1.1
Host: www.knight.com
Content-Length: 149
Content-Type: application/xml

<usuario>
  <nome>Usuário</nome>
  <sobrenome>Novo</sobrenome>
  <login>usuario_novo</login>
  <senha>aY3BnUicTk23PiinE+qwew==</senha>
</usuario>
```

Note que, diferente do método `GET`, o `POST` suporta que o conteúdo seja enviado como corpo da requisição. Isso elimina vários problemas, como limites de tamanho da requisição, codificação etc.

GET E O TAMANHO DA URL

Não há nenhum tamanho formalmente definido das URLs requisitadas através do método `GET`. No entanto, existe uma limitação prática por parte de vários servidores e *browsers* (que ainda não é definida e varia entre estes). É preferível que URLs maiores do que 255 caracteres sejam evitadas.

Com a requisição `POST` enviada, estamos dizendo ao servidor que desejamos **criar** um novo usuário, com os dados passados como parâmetro. De forma semelhante, o método `PUT` pode ser utilizado; no entanto, este é preferencialmente utilizado para **atualizar** recursos, ao invés de criá-los. Desta forma, podemos realizar uma requisição para a URL <http://www.knight.com/usuarios/1>, sinalizando nosso desejo de atualizar o usuário de ID 1. Nossa requisição ficaria assim:

```
PUT /usuarios/1 HTTP/1.1
Host: www.knight.com
Content-Length: 149
Content-Type: application/xml

<usuario>
  <nome>Usuário</nome>
  <sobrenome>Novo</sobrenome>
  <login>usuario_novo</login>
  <senha>aY3BnUicTk23PiinE+qwew==</senha>
</usuario>
```

Desta forma, se fizéssemos outra requisição `GET` para a URL <http://www.knight.com/usuarios>, nosso resultado seria o seguinte:

```
<usuarios>
  <!-- Este é o usuário que foi atualizado pela requisição PUT -->
  <usuario>
    <id>1</id>
    <nome>Usuário</nome>
    <sobrenome>Novo</sobrenome>
    <login>usuario_novo</login>
  </usuario>
  <usuario>
```

```
<id>2</id>
<nome>Adriano</nome>
<sobrenome>Almeida</sobrenome>
<login>adrianoalmeida7</login>
</usuario>
<usuario>
  <id>3</id>
  <nome>Paulo</nome>
  <sobrenome>Silveira</sobrenome>
  <login>paulo_caelum</login>
</usuario>
</usuarios>
```

E, finalmente, podemos utilizar o método `DELETE` para apagar um recurso:

```
DELETE /usuarios/1 HTTP/1.1
Host: www.knight.com
```

Ao realizar essa requisição, apagaríamos o usuário de ID 1, e passaríamos a ter somente o seguinte conjunto de usuários:

```
<usuarios>
  <!-- 0 usuário de ID 1 foi apagado -->
  <usuario>
    <id>2</id>
    <nome>Adriano</nome>
    <sobrenome>Almeida</sobrenome>
    <login>adrianoalmeida7</login>
  </usuario>
  <usuario>
    <id>3</id>
    <nome>Paulo</nome>
    <sobrenome>Silveira</sobrenome>
    <login>paulo_caelum</login>
  </usuario>
</usuarios>
```

Qual a relação do HTTP com REST?

REST é fortemente ligado ao HTTP, a ponto de não poder ser executado com sucesso em outros protocolos. Ele é baseado em diversos princípios que fizeram com que a própria *web* fosse um sucesso. Estes princípios são:

- URLs bem definidas para recursos;
- Utilização dos métodos HTTP de acordo com seus propósitos;
- Utilização de *media types* efetiva;
- Utilização de *headers* HTTP de maneira efetiva;
- Utilização de códigos de status HTTP;
- Utilização de Hipermissão como motor de estado da aplicação.

5.3 URLs PARA RECURSOS

Em REST, cada recurso deve ter uma URL bem definida. Por exemplo, o conjunto dos nossos usuários pode ter mapeada para si uma URL <http://www.knight.com/usuarios>. Caso queiramos apenas o usuário de ID 1, essa URL se torna <http://www.knight.com/usuarios/1>.

Parâmetros adicionais, que não fazem parte da definição do recurso propriamente dito e/ou sejam opcionais podem ser passados em formato de *query string*, ou seja, dados “anexados” à URL. Por exemplo, caso queiramos utilizar paginação em nossa listagem de usuários, podemos realizar uma requisição para <http://www.knight.com/usuarios?dadoInicial=o&tamanhoPagina=1>.

Note que nem sempre é fácil definir essas URLs. Elas geralmente podem ser identificadas quando temos em mãos um caso clássico de sistema para mapeamento de CRUDs (*Create, Retrieve, Update, Delete* — as famosas quatro operações em banco de dados); no entanto, casos mais complicados surgem quando precisamos, por exemplo, definir operações de pura lógica.

Tome como exemplo um sistema de validação de CPF, por exemplo: não existe uma operação de banco de dados. Não existe um recurso a ser recuperado de lugar algum, mas apenas um algoritmo a ser executado remotamente. A requisição para este algoritmo é facilmente definida em SOAP, ou qualquer protocolo RPC (*Remote Procedure Call* — Chamada a Procedimento Remoto). No entanto, em REST este é um problema mais amplo. Neste caso, devemos nos perguntar: qual é o recurso que estamos chamando? O que queremos dele? Quais são as possíveis respostas que ele pode nos dar?

No caso de um sistema de validação, portanto, sabemos que estamos realizando uma invocação a um validador, de subtipo CPF, e queremos que ele retorne se o dado é válido ou não. Neste caso, podemos mapear a URL deste recurso assim:

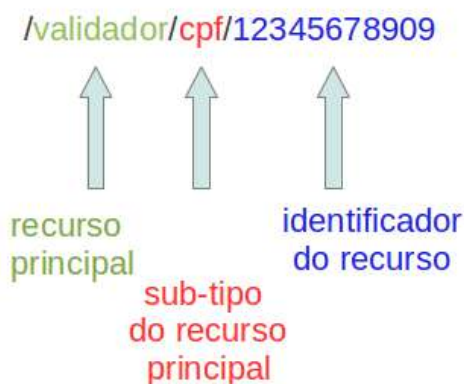


Figura 5.2: Estrutura de definição de URL

Note que, assumindo esta mesma orientação, é possível definir URLs para a maioria dos problemas. Voltando ao problema de nossos usuários, por exemplo, note que, se temos uma URL `/usuarios/1`, `/usuarios` é nosso **recurso principal** e `1` é o identificador do nosso recurso (e não temos quaisquer subtipos mapeados). No entanto, se quisermos, por exemplo, apenas o endereço de um determinado usuário, o senso comum ditaria que teríamos uma URL `/usuarios/endereco/1`, certo? Errado.

Note que as URLs seguem uma estrutura **hierárquica**, ou seja, o elemento seguinte obedece a um relacionamento com o elemento anterior. Ou seja, no exemplo dos endereços dos usuários, devemos obter primeiro o usuário em questão e, depois, o endereço. Assim sendo, a URL correta seria `/usuarios/1/endereco`. No entanto, se quiséssemos obter apenas o endereço de todos os usuários, aí teríamos uma URL `/usuarios/endereco` — `endereco` obedece a uma estrutura hierárquica em relação a usuários.

Caso queiramos adicionar algum dado à nossa requisição que não necessariamente obedece à hierarquia, ou não é um identificador, podemos utilizar as chamadas *query strings*. Elas são muito utilizadas em sites da internet para passar parâmetros adicionais a recursos. Dessa forma, em nosso exemplo de validação do CPF, podemos supor que haja um serviço realmente muito completo: ele valida tanto o número em si, utilizando o algoritmo de módulo 11 (disponível em <http://pt.wikipedia.org/wiki/CPF>), quanto a existência do CPF na Receita Federal. Queremos dar ao usuário a possibilidade de usar qualquer

dos algoritmos, ou ambos. Assim, podemos definir uma URL com o formato `/validador/cpf/(número)?algoritmo=todos`: os dados que vêm depois da interrogação contém os valores adicionais, em pares chave-valor (ou chave=valor), separados por `&`.

5.4 MÉTODOS HTTP E USO DE MIME TYPES

Além de URLs bem definidas, os *web services* REST também devem ter métodos HTTP bem definidos para realizar as operações.

Voltando ao questionamento em relação aos recursos: o que queremos dele? Se quisermos que algo seja inserido em nossa base de recursos, utilizamos o método `POST`. Se quisermos recuperar dados, utilizamos o método `GET`. Se quisermos atualizar/apagar dados, utilizamos `PUT` ou `DELETE`, respectivamente.

Podemos, além disso, utilizar `HEAD` em casos particulares de uma requisição `GET` sem dados a serem retornados. Por exemplo, no nosso sistema de validação, é possível efetuar uma requisição `GET` para obter as mensagens relativas ao resultado da validação. Os códigos de *status* HTTP são fornecidos para dar uma “visão geral” do resultado (ou seja, se a requisição teve sucesso ou não). No caso de uma requisição `GET`, é possível retornar mensagens de erro mais detalhadas no corpo da requisição — o que, no entanto, aumenta o tráfego de rede. Caso o cliente não deseje obter essas mensagens, é suficiente utilizar `HEAD`. Também podemos utilizar `OPTIONS` para descobrirmos quais operações podemos realizar com uma determinada URL.

E por falar em obtenção e inserção de recursos, o quê, exatamente, quer dizer, **inserir** um recurso no servidor? Note que, ao realizarmos uma requisição `GET` ao servidor, não esperamos receber a representação daquele recurso, mas sim a representação em **formato agnóstico**. Por exemplo, em Java, é muito comum utilizarmos páginas JSP para criarmos saídas em formato HTML do servidor. Em outras palavras, construímos um mecanismo (JSP) que realizará produção de um recurso em formato agnóstico (HTML). Assim, o HTML produzido pode ser lido por diversos *browsers* diferentes.

Da mesma forma, quando construímos serviços REST estamos construindo mecanismos semelhantes. Porém, não estamos limitados a simplesmente solicitar recursos do servidor. Através dos diferentes métodos HTTP, podemos tanto solicitar recursos existentes como criar novos; tanto atualizar quanto apagar.

Quanto à representação do que queremos, isso é negociável através de tipos MIME. Os tipos MIME são o mecanismo padrão de negociação de conteúdo na in-

ternet, e são representados por *strings* contendo o tipo genérico <barra> um tipo mais especializado. Segue tabela dos tipos mais comuns:

- HTML - `text/html`;
- XML - `text/xml` ou `application/xml`;
- PNG - `image/png`;
- JPG - `image/jpg`;
- JSON - `application/json`;
- PDF - `application/pdf`.

Ao realizar essa negociação de conteúdos com o servidor, é possível ter diversas representações da mesma coisa. Por exemplo, se fizermos uma requisição do tipo GET para uma URL `/usuarios/1`, podemos obter a representação do usuário de ID 1 como XML, JSON ou até mesmo como imagem (o que, neste caso, retornaria o retrato do usuário em questão).

Esta negociação é feita através de dois *headers* HTTP padronizados: `Accept` e `Content-Type`. `Accept` é enviado do cliente para o servidor, informando qual tipo MIME será aceito como resultado da requisição; `Content-Type` é enviado do servidor para o cliente informando qual tipo de representação está enviando para o cliente. Por exemplo, se quisermos que o servidor nos envie a representação de um usuário como XML, podemos enviar a seguinte requisição HTTP:

```
GET /usuarios/1 HTTP/1.1
Host: www.knight.com
Accept: application/xml
```

Note que, caso queiramos o retrato do usuário, podemos enviar uma requisição para a mesma URL, com o mesmo método HTTP; basta que alteremos o tipo de retorno esperado:

```
GET /usuarios/1 HTTP/1.1
Host: www.knight.com
Accept: image/png
```

Note também, que não precisamos restringir a nossa requisição a um tipo específico. Por exemplo, se quisermos receber uma representação em formato de imagem, mas sem estarmos presos a nenhum tipo específico, podemos utilizar * (asterisco), que funciona como um curinga. A mesma requisição poderia, portanto, ser feita assim:

```
GET /usuarios/1 HTTP/1.1
Host: www.knight.com
Accept: image/*
```

Desta maneira, o servidor seleciona o tipo de conteúdo e nos envia uma resposta com o seguinte formato:

```
HTTP/1.1 200 OK
Date: Sun, 14 Oct 2012 01:21:35 GMT
Content-Length: 18695
Content-Type: image/png

<!-- imagem -->
```

5.5 UTILIZAÇÃO EFETIVA DE HEADERS HTTP

Além de utilizar métodos HTTP e MIME types, um conceito sempre presente em REST é o uso de *headers*. Os *headers* HTTP possuem algumas metainformações a respeito daquilo que se está comunicando/recebendo do servidor, e são um mecanismo muito útil para se transmitir informações quando a representação do nosso recurso não suporta essas informações a mais. Por exemplo, se quisermos inserir a descrição da imagem na criação da mesma, basta que enviemos a seguinte requisição para o servidor:

```
POST /usuarios/1 HTTP/1.1
Host: www.knight.com
Content-Length: 18695
Content-Type: image/png
Descricao: Meu retrato no dia 21/10/2012

<!-- imagem -->
```

Note que o cabeçalho é totalmente customizado, e seu envio **não é obrigatório**. Esta é uma informação importante: por se tratar de um metadado, você deve fazer o possível para não tornar informações no cabeçalho obrigatórias.

Além de cabeçalhos customizados, certos *headers* são especialmente úteis. A especificação HTTP habilita, entre outras coisas, o uso nativo de mecanismos de *cache*. Desta forma, para economizar o uso de rede, é possível “negociar” com o servidor a passagem de informações.

Isso é possível, dentre outras formas, através do *header* `If-Modified-Since`. Note que, sempre que realizamos uma requisição qualquer, por padrão o servidor nos manda uma requisição contendo a data em que o recurso foi enviado. Esta data está contida no *header* `Date`.

A partir desse momento, o cliente pode manter a representação desse recurso, em conjunto com a data. Se porventura o cliente precisar solicitar o recurso novamente, basta enviar este *header*, com a data em que o cliente recebeu a última atualização da mensagem. Se o recurso não tiver sido atualizado, o servidor enviará um código de status 304 e deixará de enviar novamente a representação do recurso, indicando que o cliente pode reutilizar a anterior.

Por exemplo, suponha a seguinte requisição `GET`:

```
GET /usuarios/1 HTTP/1.1
Host: www.knight.com
Accept: image/png
```

A resposta desta requisição será algo como:

```
HTTP/1.1 200 OK
Date: Sun, 14 Oct 2012 01:21:35 GMT
Content-Length: 18695
Content-Type: image/png
```

```
<!--imagem-->
```

Supondo que mantenhamos o resultado da primeira requisição em *cache*, podemos enviar requisições subsequentes da seguinte forma:

```
GET /usuarios/1 HTTP/1.1
Host: www.knight.com
If-Modified-Since: Sun, 14 Oct 2012 01:21:35 GMT
Accept: image/png
```

Desta maneira, podemos receber tanto uma resposta idêntica à primeira, quanto uma resposta assim:

HTTP/1.1 304 Not Modified

Date: Sun, 14 Oct 2012 01:23:22 GMT

Note que uma resposta com status 304, diferentemente de uma com status 200, não traz os resultados da requisição no corpo, economizando os recursos de rede.

5.6 UTILIZAÇÃO DE CÓDIGOS DE STATUS

Note que, para utilização efetiva de nossos serviços REST, podemos (e devemos) utilizar os códigos de status HTTP para realizar a comunicação com nossos serviços. Isso facilita o reconhecimento do estado da requisição: se teve sucesso ou não, se teve falha ou não, e se tiver ocorrido alguma falha, qual foi. Por exemplo, no nosso mecanismo de usuários, você já sabe que, quando solicitamos a listagem de usuários, o código de status é 200 e, quando é 304, significa que nada mudou desde a última vez em que fizemos a solicitação. No entanto, pode ser útil, na criação de um novo usuário, utilizar um código de status diferente: 201 (Created). Este código de status indica a presença de um *header Location*, que traz a URI onde o novo recurso está hospedado. Se enviarmos a requisição com campos requeridos faltando, é possível enviar, na resposta para o cliente, o código 400 (Bad Request).

Em geral, os códigos de status seguem a seguinte ordem: os que começam com 1 são informativos (não se preocupe com estes, geralmente não são utilizados). Os que começam com 2 indicam sucesso da operação, com variações. Por exemplo, 200 é uma resposta de sucesso genérica. 201 significa que o recurso foi criado, e está presente em uma determinada URL, e assim por diante. Os que começam com 3 indicam que o cliente deve tomar alguma espécie de ação. Você já viu o campo 304, por exemplo, que indica que o recurso não foi modificado e que o cliente deve reutilizar o conteúdo que já tinha. Os códigos que começam com 4 indicam alguma espécie de falha na requisição e os códigos que começam com 5 indicam falhas no processamento da requisição.

Os códigos mais utilizados são:

- 200 OK
- 201 Created
- 202 Accepted
- 204 No Content

- 304 Not Modified
- 400 Bad Request
- 401 Unauthorized
- 403 Forbidden
- 404 Not Found
- 405 Method Not Allowed
- 409 Conflict
- 415 Unsupported Media Type
- 500 Internal Server Error

Na prática, os códigos que são encarados como “sucesso” são os das famílias 2xx e 3xx. Como você já viu antes, um código 200 OK é um código genérico que indica que a requisição foi processada com sucesso. Já o 201 Created é utilizado como resposta para requisições `POST`. Indica que a requisição foi processada com sucesso e o recurso está presente em uma URL (apresentada no *header Location*).

O status 202 Accepted indica que a requisição foi aceita; no entanto, ela não será processada imediatamente. É comumente utilizado em serviços assíncronos, sendo que a URL onde se confere o resultado pode estar presente no *header Location*. Além disso, o status 202 pode retornar conteúdo no corpo da resposta, contendo ao menos o tempo estimado de processamento (segundo [10]).

Já o status 204 No Content é comumente utilizado como resultado de requisições `POST`, `PUT` ou `DELETE`, indicando que a requisição foi processada mas não há nenhum corpo da requisição a ser preenchido. E, finalmente, o código 304 Not Modified é usado para indicar que o recurso não foi alterado.

Os códigos de erro são os das famílias 4xx e 5xx. O código 400 Bad Request é um código genérico para indicar algum problema no conteúdo da requisição. É comumente utilizado quando algum campo não foi preenchido, ou preenchido incorretamente. O código 401 Unauthorized significa que o cliente não está autorizado a visualizar um determinado recurso (tendo fornecido credenciais que não dão acesso àquele recurso ou mesmo não tendo fornecido credencial alguma).

O código 403 Forbidden é parecido com 401, com a diferença de que é utilizado quando o servidor reconhece que o recurso solicitado existe. O código 404

Not Found, por sua vez, indica que o recurso não existe. O código 405 Method Not Allowed indica que o método HTTP utilizado não é suportado. Por exemplo, suponha que um determinado recurso não pode ser apagado, depois de criado; mesmo assim, o cliente envia uma requisição `DELETE`. Assim, o servidor pode responder com o status 405. O status 409 Conflict, por sua vez, indica que há um conflito entre recursos. Para garantir que os clientes estejam trabalhando com a versão mais atual de um determinado recurso, o servidor pode utilizar um campo de versão. Assim, considere o seguinte cenário:

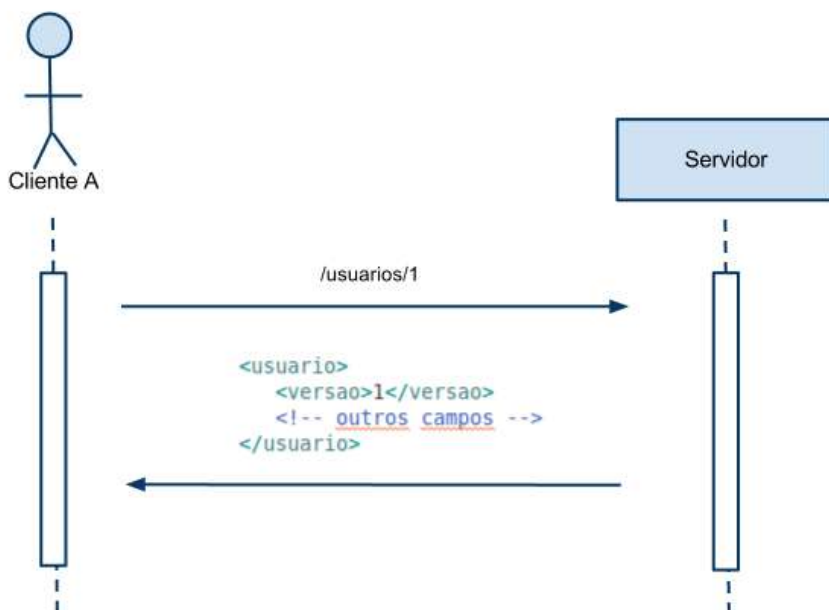


Figura 5.3: Solicitação de recurso com versionamento

Note que, a partir desse momento, o cliente A tem uma cópia de trabalho do usuário 1. Se um cliente B fizer a mesma requisição, ele também obterá uma cópia desse usuário, com a mesma versão (note que este é o padrão de projeto *Optimistic Offline Lock*, como descrito por Martin Fowler em seu livro *Patterns of Enterprise Application Architecture* [8]).

Se o cliente B fizer uma atualização, a versão do servidor será automaticamente incrementada para 2. No entanto, se o cliente A fizer uma alteração, ele deverá dizer

ao servidor que a versão que possui é 1 (que está, portanto, desatualizada). Esta nova requisição será respondida com um código 409 Conflict, indicando ao cliente A que este deve obter uma versão mais atualizada (neste caso) antes de realizar uma edição.

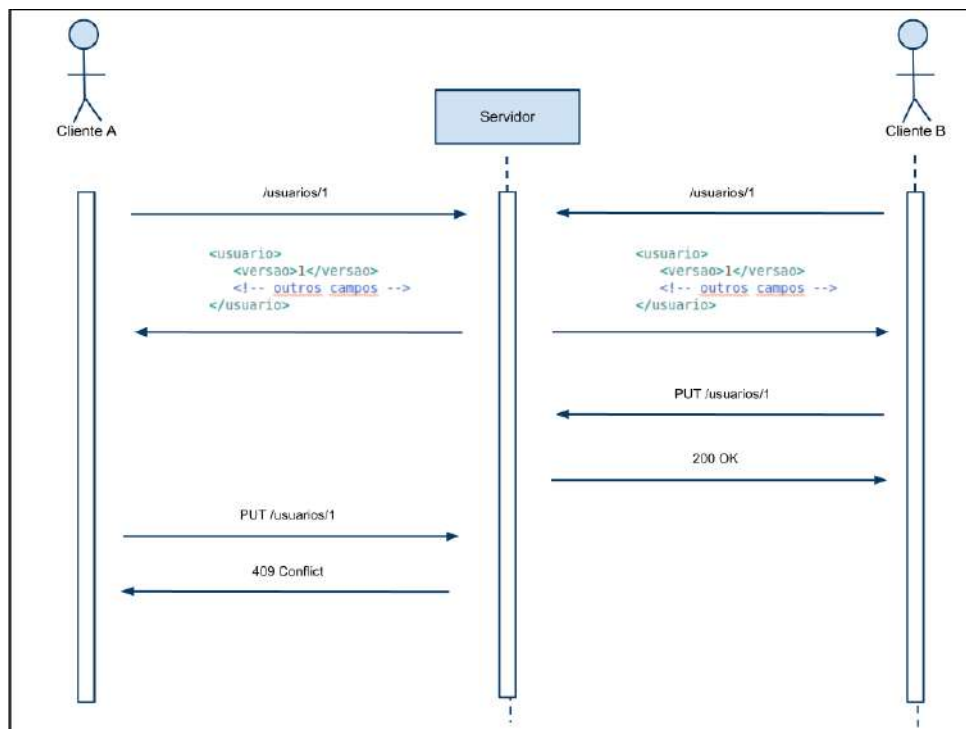


Figura 5.4: Controle de versões com REST

Ainda na família 4xx, o código 415 Unsupported Media Type é enviado quando o cliente manda uma requisição com formato não reconhecido pelo servidor. Por exemplo, supondo que o cliente envie uma requisição em formato JSON (*JavaScript Object Notation*), mas o servidor não suporte este formato, este código de status é enviado pelo servidor.

Finalmente, o código de status 500 Internal Server Error significa que alguma condição inesperada, por parte do servidor, ocorreu. Este código de status é enviado quando alguma exceção ocorreu internamente no servidor, e ele não soube o que fazer para se recuperar. No entanto, a requisição pode ser reenviada pelo cliente — lembre-se, porém, de evitar reenviar a requisição muitas vezes, para que o servidor não seja “derrubado” por falta de recursos.

5.7 UTILIZAÇÃO DE HIPERMÍDIA COMO MOTOR DE ESTADO DA APLICAÇÃO

Um recurso muito subestimado, porém previsto na dissertação de Roy Fielding [6], é o uso de hipermídia como motor de estado da aplicação (conhecido na sigla em inglês como HATEOAS - *Hypermedia As The Engine Of Application State*). Consiste de uma técnica já muito conhecida de todos nós, no uso de páginas web.

Quando solicitamos uma página web, em geral obtemos, em conjunto com o texto propriamente dito, inúmeros recursos adicionais: folhas de estilo, *scripts*, imagens etc. Porém, quando solicitamos a página, geralmente estes recursos adicionais não estão presentes no HTML, mas apenas as referências a estes. Quando o *browser* solicita a página, o mesmo faz uma varredura no HTML para descobrir onde estão os recursos adicionais que estão presentes nesta. Assim, obtendo as referências, ele tem condições de fazer novas requisições para carregá-los.

Da mesma maneira, Roy Fielding previu que seria excelente para as aplicações se, ao invés de manter as URLs para novos recursos na própria aplicação, seria mais interessante se cada retorno de recurso trouxesse consigo URLs para novas operações. Para descobrir o que fazer com cada URL, basta que a aplicação utilize os métodos já padronizados: `GET` para obter o recurso, `POST` para criar, e assim por diante.

Por exemplo, quando requisitamos a listagem de usuários do sistema, o ideal seria que obtivéssemos algo como o seguinte:

```
<usuarios xmlns:links="http://knight.com/links">
  <usuario>
    <id>1</id>
    <nome>Alexandre</nome>
    <login>asaudate</login>
    <links:link href="/usuarios/1" rel="self" />
  </usuario>
  <!-- Outros 19 usuários -->
  <links:link href="/usuarios" rel="self" />
  <links:link href="/usuarios?offset=20&limit=20" rel="nextPage" />
  <links:link href="/usuarios?offset=40&limit=20" rel="lastPage" />
</usuarios>
```

Note a presença dos elementos `link`. Eles contêm tanto a referência para o que se pode fazer com o recurso em questão quanto a relação deles com o recurso. Por exemplo, os elementos que contêm `rel="self"` referem-se ao próprio recurso. Repare, no entanto, na presença dos elementos com `rel="nextPage"` e

`rel="lastPage"`. Eles indicam que provavelmente esse recurso é paginado, e eles contêm os links tanto da próxima página quanto da última.

Note que, desta forma, a aplicação não precisa conhecer quais são os links para as próximas páginas, mas apenas conhecer os links com as relações `nextPage` e `lastPage`. Isso faz com que o cliente fique mais desacoplado do servidor, já que não é necessário atualizar o código caso algum destes links mude.

COMO DEFINIR OS LINKS

Para definir seus links, recomendo utilizar algo similar aos links HTML. Não há uma definição formal a esse respeito, embora exista um esforço da W3C neste sentido, chamado XML Linking Language (XLink), que pode ser conferido em <http://www.w3.org/TR/xlink/>. Particularmente, eu considero este modelo pouco intuitivo, e prefiro alternativas, como o já mencionado modelo utilizado em HTML para carregar folhas de estilo.

5.8 COMO DESENVOLVER SERVIÇOS REST

Atualmente, os serviços REST estão padronizados sob uma única especificação, em Java, chamada JAX-RS. Esta especificação funciona no mesmo molde que outras especificações, ou seja, provê um conjunto de anotações/classes e deixa a cargo de vários *frameworks* o funcionamento propriamente dito. Dentre os principais *frameworks*, posso destacar:

- Jersey (que é a implementação de referência);
- RESTEasy;
- Apache CXF;

Além disso, existe também uma implementação brasileira de REST, que é o *framework Restfulie* (criado por Guilherme Silveira).

No nosso caso, quando desenvolvemos utilizando o JBoss AS, normalmente utilizamos o RESTEasy, que já está incluso na distribuição padrão.

Em geral, todo serviço JAX-RS precisa conter:

- Uma ou mais anotações `@Path` (que pode ser incluída tanto em classes quanto em interfaces e métodos);
- Uma ou mais anotações `@Consumes` (que segue a mesma regra de `@Path`);
- Um número qualquer de anotações `@Produces`;
- Definição dos métodos HTTP a serem utilizados, através das anotações `@GET`, `@POST`, `@PUT`, `@DELETE`, `@HEAD` ou `@OPTIONS`.

Por exemplo, para definirmos um método para listagem de usuários que sempre produza código com o tipo MIME `application/xml`, podemos utilizar o seguinte código:

```
import javax.ws.rs.*;
import javax.ws.rs.core.*;

// Outros imports

@Path("/usuarios")
@Produces(MediaType.APPLICATION_XML)
@Consumes(MediaType.APPLICATION_XML)
public class UsuariosService {

    @PersistenceContext
    private EntityManager em;

    @GET
    public Usuarios listarUsuarios() {
        return new Usuarios(em.createQuery("select u from Usuario u")
                                .getResultList());
    }
}
```

Note que uma classe adicional, `Usuarios`, foi criada para que possamos retornar a coleção de usuários. O código dela é o seguinte:

```
package com.knight.usuarios.modelos;

//Imports omitidos
```

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Usuarios {

    @XmlElement(
        name = "usuario")
    private Collection<Usuario> usuarios;

    public Usuarios() {
    }

    public Usuarios(Collection<Usuario> usuarios) {
        this.usuarios = usuarios;
    }

    // getter e setter para o campo usuarios

}
```

Além disso, é necessário incluir a seguinte informação no arquivo WEB-INF/web.xml:

```
<web-app>
  <servlet-mapping>
    <servlet-name>javax.ws.rs.core.Application</servlet-name>
    <url-pattern>/services/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Ao executar esse código, basta apontar o *browser* para a URL <caminho da aplicação>/services/usuarios e conferir o resultado:

```
<usuarios>
  <usuario>
    <id>1</id>
    <nome>Alexandre</nome>
    <login>admin</login>
    <senha>admin</senha>
  </usuario>
</usuarios>
```

Também queremos que haja uma URL para busca de um usuário específico (com o formato <caminho da aplicação>/services/usuarios/<id do

`usuario>`), então, podemos incluir uma anotação `@Path` adicional. Essa anotação adicional irá conter a especificação do que fazer com o ID do usuário. Para que possamos receber esse ID, anotamos o parâmetro com `@PathParam`:

```
@Path("/usuarios")
@Produces(MediaType.APPLICATION_XML)
@Consumes(MediaType.APPLICATION_XML)
public class UsuariosService {

    @PersistenceContext
    private EntityManager em;

    //Método de busca de todos os usuários

    @GET
    @Path("/{id}")
    public Usuario find(@PathParam("id") Long id) {
        return em.find(Usuario.class, id);
    }
}
```

A classe Response

Note que, como temos vários códigos de status HTTP diferentes, precisamos de um mecanismo para que possamos controlar o tráfego HTTP. A especificação JAX-RS fornece a classe `javax.ws.rs.core.Response` para isso. Por exemplo, o código para recuperar um usuário pode ser reescrito assim:

```
@GET
@Path("/{id}")
public Response find(@PathParam("id") Long id) {
    Usuario usuario = em.find(Usuario.class, id);

    if (usuario != null) {
        return Response.ok(usuario).build();
    }
    return Response.status(
        javax.ws.rs.core.Response.Status.NOT_FOUND
    ).build();
}
```

Nesta situação, caso o usuário não seja encontrado, o servidor retorna o código de status 404 (representado pela constante `javax.ws.rs.core.Response.Status.NOT_FOUND`). Caso ele seja encontrado, utilizamos o método `ok()` da classe `Response`, passando como parâmetro o usuário encontrado. O uso deste método será traduzido como uma resposta de status 200 OK, contendo o usuário encontrado no corpo da resposta.

Construção de URI

No caso do método de criação de novos usuários, podemos utilizar a classe `Response` da mesma maneira como nos métodos de localização de usuários. No entanto, como dito na seção 5.6, o melhor código de status a ser utilizado na resposta de criação de recursos é o 201 Created, que é a resposta padrão para invocação através do método HTTP `POST`. Lembre-se de que este código deve trazer o *header* `Location`, que contém a a URI onde o recurso recém-criado pode ser encontrado. Ou seja, precisamos de um mecanismo que faça a construção da URI para nós.

QUAL A DIFERENÇA ENTRE URL E URI?

Segundo a RFC 3986 (<http://www.ietf.org/rfc/rfc3986.txt>), uma URI pode ser classificada como um localizador, um nome, ou ambos. Já uma URL é um subconjunto das URIs que, além de identificar um recurso, provê meios para localizar o recurso através da descrição do mecanismo de acesso primário. Em geral, podemos identificar se um determinado caminho é uma URL se a mesma contiver o protocolo utilizado para acesso.

A especificação JAX-RS dá acesso a algumas funcionalidades desse tipo através da interface `javax.ws.rs.core.UriInfo`. Esta interface é gerenciada pela própria *engine* do JAX-RS, e não precisamos instanciá-la; basta apenas injetá-la no nosso serviço, utilizando a anotação `javax.ws.rs.core.Context`. Podemos utilizar esse mecanismo para injetar uma instância de `UriInfo` na própria classe de serviço, como um atributo, ou como parâmetro de um método.

Dentre os vários métodos utilitários fornecidos pela interface `UriInfo`, está `getAbsolutePathBuilder`. Este método traz a URL que foi invocada (exceto *query strings*) para chegar ao método atual em um *builder*; ou seja, é possível adicionar dados à URI invocada. Desta forma, se realizarmos uma requisição `POST` para

a criação de um recurso, podemos utilizar esses métodos para a construção da URI onde o cliente pode buscar o dado.

Com base nessas informações, o seguinte código é escrito:

```
@POST
public Response create(@Context UriInfo uriInfo, Usuario usuario) {
    em.persist(usuario);
    // Constrói a URL onde o recurso estará disponível
    UriBuilder uriBuilder = uriInfo.getAbsolutePathBuilder();
    URI location = uriBuilder.path("/{id}").build(usuario.getId());

    return Response.created(location).build();
}
```

Como pode ser visto, a notação utilizada para construir a URI é idêntica à notação utilizada para descrever o método que busca um único usuário. O usuário passado no corpo da requisição é persistido e, caso a persistência tenha sucesso, o ID gerado é utilizado para compor a URI do recurso. Na sequência, o método `created` da classe `Response` é utilizado para retornar uma resposta 201 Created para o cliente (e o parâmetro para este método será utilizado no *header* `Location`).

Recebendo e enviando headers

Para enviar e receber *headers* HTTP, a anotação `@javax.ws.rs.HeaderParam` pode ser utilizada. Ela é usada em assinaturas de métodos, e recebe como parâmetro o nome do cabeçalho em questão.

Vejamos uma aplicação prática: você se lembra do cabeçalho `If-Modified-Since`, apresentado na seção 5.5? Ele é utilizado para enviar uma data para o servidor, dizendo a este que o cliente só deseja o resultado da requisição se o recurso tiver sido modificado desde então. Nesse caso, podemos utilizar esta anotação em nossos métodos de busca para interagir diretamente com este cabeçalho, e então, alteramos nosso método de busca para ter a seguinte assinatura:

```
@GET
@Path("/{id}")
public Response find(@PathParam("id") Long id,
    @HeaderParam("If-Modified-Since") Date modifiedSince)
```

Note que o retorno do método foi alterado para utilizar a classe `Response`, para que possamos manipular o código de status utilizado na resposta HTTP. Além

disso, estamos explicitamente declarando o cabeçalho `If-Modified-Since` como `java.util.Date`, para que possamos interagir com essa data.

Para que isso seja efetivo, precisamos, no entanto, alterar nossa classe `Usuario` para conter a data de atualização do recurso. Repare que esta é uma informação que pode ser aproveitada por todas as nossas classes de modelo, então, podemos modelar essa informação numa superclasse, `EntidadeModelo`:

```
@MappedSuperclass
public abstract class EntidadeModelo {

    @Temporal(TemporalType.TIMESTAMP)
    private Date dataAtualizacao;

    @XmlTransient
    public Date getDataAtualizacao() {
        return dataAtualizacao;
    }

    @PreUpdate
    @PrePersist
    protected void ajustarDataAtualizacao() {
        this.dataAtualizacao = new Date();
    }
}
```

Perceba que estamos usando algumas anotações básicas da JPA, para tornar as informações persistentes. Para saber mais sobre JPA, recomendo o livro da Casa do Código sobre o assunto.

A classe está devidamente anotada com `@MappedSuperclass`, indicando para o JPA que os dados contidos nela devem ser persistidos. Além disso, note que o *getter* do campo `dataAtualizacao` está marcado com `@XmlTransient`. Isso porque este *getter* será lido pela *engine* do JAXB, ainda que esta classe, ou suas subclasses, estejam anotadas para mapear os campos diretamente.

Ainda, note o método `ajustarDataAtualizacao`: por conter as anotações `@PreUpdate` e `@PrePersist`, este método será invocado sempre que uma de suas subclasses seja atualizada (comportamento ajustado por `@PreUpdate`) ou criada no banco de dados (comportamento ajustado por `@PrePersist`).

Tendo acesso à data de atualização do usuário, podemos completar o nosso método:

```

@GET
@Path("/{id}")
public Response find(@PathParam("id") Long id,
    @HeaderParam("If-Modified-Since") Date modifiedSince) {
    Usuario usuario = em.find(Usuario.class, id);

    if (usuario != null) {
        if (modifiedSince == null
            || (modifiedSince != null && usuario.getDataAtualizacao()
                .after(modifiedSince))) {
            return Response.ok(usuario).build();
        }

        return Response.notModified().build();
    }
    return Response.status(Status.NOT_FOUND).build();
}

```

Repare que é necessário testar se o parâmetro foi passado; se não tiver sido, o valor de `modifiedSince` será `null`.

5.9 MAPEAMENTO AVANÇADO: TRATANDO IMAGENS

Até aqui, você viu como utilizar as anotações JAX-RS para fazer o básico, ou seja, manipulação básica de *headers* vindos do cliente, métodos HTTP e status de resposta, além de ter visto o básico da manipulação de tipos MIME. No entanto, e se os usuários do sistema tiverem retratos? Como realizar a manipulação correta desse tipo de informação, no modelo REST?

Vamos começar modelando a classe que contém a imagem. Fazemos isso numa classe separada, para que possamos armazenar, junto da imagem, uma descrição da mesma:

```

@Entity
public class Imagem extends EntidadeModelo {

    @Id
    @GeneratedValue(
        strategy = GenerationType.IDENTITY)
    private Long id;
}

```

```
@Lob
private byte[] dados;
private String tipo;
private String descricao;

//getters e setters
}
```

Note que não é necessário colocar anotações JAXB nessa classe, já que ela não será trafegada como XML ou como qualquer linguagem de marcação. Ao invés disso, colocamos um campo `tipo` para armazenar o tipo MIME dessa imagem, bem como um campo `dados`, que irá conter a imagem propriamente dita. Esse campo precisa conter a anotação `@Lob` para indicar à *engine* JPA que este campo deve ser mapeado para um BLOB, ou seja, um campo de armazenamento de dados binários.

Feita a modelagem da classe de imagens, inserimos o mapeamento desta para usuários. Como esta é uma classe que tem um relacionamento direto com apenas uma instância de um usuário, e cada usuário tem apenas um retrato, este é chamado de relacionamento um-para-um. Este tipo de relacionamento é criado utilizando a anotação `@OneToOne`. Assim, mapeamos o retrato da seguinte maneira:

```
@OneToOne(
    cascade = { CascadeType.ALL }, orphanRemoval = true)
@XmlTransient
private Imagem imagem;
```

Note que os atributos `cascade` e `orphanRemoval` foram inseridos para garantir que as operações sobre um usuário sejam refletidas para a imagem e para garantir que não haja nenhuma imagem “órfã”, ou seja, sem relacionamento com nenhum usuário. Além disso, a anotação `@XmlTransient` foi inserida para que o JAXB não interprete este campo como parte do usuário (novamente, lembre-se que a imagem não será trafegada como XML, mas em seu formato puro).

MAPEAMENTO COM JPA

Se quiser conhecer mais a fundo o mapeamento com JPA, sugiro verificar o livro “Aplicações Java para a web com JSF e JPA”, disponível em <http://www.casadocodigo.com.br/products/livro-jsf-jpa>.

Feito este mapeamento, precisamos escrever o código que fará a busca/criação de imagens. Nossos requisitos são:

- Fazer com que o cliente tenha a percepção de que a imagem binária e os dados trafegados como XML pertençam à mesma entidade;
- Tratar imagens de uma maneira genérica, ou seja, não fazer distinção entre PNG, JPG, GIF ou outros formatos;
- Fazer com que o cliente tenha a possibilidade de cachear o resultado de busca;
- Poder inserir a descrição da imagem, bem como retornar esta descrição em conjunto com a imagem quando solicitada.

Perceba que, quando criamos o serviço de usuários, utilizamos as anotações `@Consumes` e `@Produces` diretamente na declaração de classe, para sinalizar que gostaríamos que o tipo MIME padrão para tratar os recursos fosse `application/xml`. No entanto, gostaríamos que essa imagem tivesse o mesmo endereço do serviço de usuários, apenas alternando os modos de resposta quando o cliente solicitar essa mudança. Desta forma, criamos nosso serviço de busca para sinalizar que ele produz imagens, utilizando o MIME type `image/*` (onde o “*” é um curinga, ou seja, não especifica qual o tipo de imagem):

```
@Produces("image/*")
@GET
public Response recuperarImagem()
```

Assim, esse método será selecionado pelo cliente, quando este enviar o *header* `Accept` com valor `image/*`. Isso quer dizer, do ponto de vista do cliente, que ele aceita como resposta do serviço, qualquer tipo de imagem (diferentemente do padrão deste serviço, `application/xml`).

Resta fazer com que o cliente obtenha a imagem de um cliente específico. Para isso, utilizamos novamente as anotações `@Path` e `@PathParam`:

```
@GET
@Path("/{id}")
@Produces("image/*")
public Response recuperarImagem(@PathParam("id") Long id)
```

Por último, acrescentamos o *header* `If-Modified-Since`, para que possamos acrescentar a possibilidade do cliente fazer *cache* da imagem retornada:

```

@GET
@Path("/{id}")
@Produces("image/*")
public Response recuperarImagem(@PathParam("id") Long id,
                                @HeaderParam("If-Modified-Since") Date modifiedSince)

```

Agora, precisamos codificar o corpo da requisição. Lembre-se de que o ID pertence ao usuário (e não à própria imagem), bem como devemos incluir o campo `descricao` no retorno da busca. Para isso, utilizamos o método `header()` da classe `Response`, o que fará com que a descrição da imagem seja incluída como *header* HTTP na resposta da requisição.

Além disso, no método `ok()`, passamos como parâmetro tanto o *array* de *bytes* de que a imagem é formada como o tipo. O tipo pode ser incluído como um segundo parâmetro para este método; assim, o código completo do método de busca pela imagem fica:

```

@GET
@Path("/{id}")
@Produces("image/*")
public Response recuperarImagem(@PathParam("id") Long id,
                                @HeaderParam("If-Modified-Since") Date modifiedSince) {
    Usuario usuario = em.find(Usuario.class, id);
    if (usuario == null) {
        return Response.status(Status.NOT_FOUND).build();
    }
    Imagem imagem = usuario.getImagem();

    if (modifiedSince != null && imagem.getDataAtualizacao()
        .before(modifiedSince)) {
        return Response.notModified().build();
    }

    return Response.ok(imagem.getDados(), imagem.getTipo())
        .header("Descricao", imagem.getDescricao()).build();
}

```

Agora, resta criar o método de criação/atualização da imagem. Como um usuário já precisa existir para que possamos atribuir um retrato a ele, não faz sentido atribuir isso a um método `POST`; é mais adequado para um método `PUT`. Além disso, nosso método precisa estar ciente de que só consome requisições de imagens. Assim, nós criamos o método e o anotamos com `@Consumes`, da seguinte forma:

```
@PUT
@Path("/{id}")
@Consumes("image/*")
public Response adicionarImagem(@PathParam("id") Long idUsuario)
```

Note que, como imagens são de tipo binário, declaramos o conteúdo como um *array* de *bytes*. Como precisamos também da descrição, podemos incluir a definição desse campo na assinatura do método. Dessa forma, a assinatura do nosso método fica assim:

```
@PUT
@Path("/{id}")
@Consumes("image/*")
public Response adicionarImagem(
    @HeaderParam("Descricao") String descricao,
    @PathParam("id") Long idUsuario,
    byte[] dadosImagem)
```

Finalmente, devemos ter acesso ao tipo de conteúdo enviado para que possamos incluir na instância de *Imagem* (e utilizar como retorno do método de busca). Poderíamos mapear o *header* *Content-Type* da mesma maneira como fizemos anteriormente, mas por fins didáticos, faremos de outro jeito.

A classe *HttpServletRequest*, definida pela API de *Servlets* Java, possui o método *getContentType*, que é utilizado para recuperar esse conteúdo. Iremos acessar esse método, injetando a instância correta de *HttpServletRequest* no nosso método através da anotação *@Context*. Desse modo, a assinatura completa do método fica assim:

```
@PUT
@Path("/{id}")
@Consumes("image/*")
public Response adicionarImagem(
    @HeaderParam(CAMPO_DESCRICAO_IMAGEM) String descricao,
    @PathParam("id") Long idUsuario,
    @Context HttpServletRequest httpRequest,
    byte[] dadosImagem)
```

Resta apenas a codificação deste método. Basta apenas que façamos a busca do usuário (retornando 404 Not Found caso ele não exista) e a atribuição da imagem a ele. Na sequência, precisamos atualizar o usuário no banco de dados, utilizando

o método `merge` do nosso `EntityManager`. Desta forma, o código completo do método fica assim:

```
@PUT
@Path("/{id}")
@Consumes("image/*")
public Response adicionarImagem(
    @HeaderParam(CAMPO_DESCRICAO_IMAGEM) String descricao,
    @PathParam("id") Long idUsuario,
    @Context HttpServletRequest httpRequest,
    byte[] dadosImagem) {
    Usuario usuario = em.find(Usuario.class, idUsuario);
    if (usuario == null) {
        return Response.status(Status.NOT_FOUND).build();
    }
    Imagem imagem = new Imagem();
    imagem.setDados(dadosImagem);
    imagem.setDescricao(descricao);
    imagem.setTipo(httpServletRequest.getContentType());
    usuario.setImagem(imagem);
    em.merge(usuario);
    return Response.noContent().build();
}
```

5.10 INCLUINDO LINKS HATEOAS

O último passo para que possamos deixar nosso serviço inteiramente de acordo com o proposto por Roy Fielding [6] é a inclusão de links em nossas entidades. Esses links devem levar a outros recursos, que de alguma forma estão relacionados ao recurso atual. Por exemplo, se incluirmos paginação em nossos recursos, podemos modelar a URL de paginação da seguinte forma:

```
/usuarios* *?inicio=0&tamanhoPagina=5* *
```

Como explicado anteriormente, esses parâmetros são opcionais (podem assumir valores-padrão). Se incluídos na URL, eles vão modificar a maneira como o recurso se apresenta (no caso da paginação, por exemplo, o tamanho da entidade `usuarios` será limitado a certo tamanho, além de conter valores diferentes de página para página). Assim, o resultado da nossa requisição com paginação poderá conter os links para a primeira página, última página, página anterior, posterior, e assim por diante.

Para utilizar valores-padrão em nossas requisições, podemos utilizar a anotação

`javax.ws.rs.DefaultValue`. Esta anotação pode ser utilizada para definir valores de tipos primitivos, classes bem conhecidas da API (como `String`, classes `wrapper`, `BigDecimal`, e assim por diante) e até *enums*. Isso faz com que a API fique limpa, sem necessidade de utilizar *if's* para testar se o valor foi passado ou não. Por exemplo, para que possamos fazer a paginação da nossa listagem de usuários, podemos modificar a assinatura do nosso método de listagem para ficar assim:

```
public Response listarUsuarios(
    @HeaderParam("If-Modified-Since") Date date,
    @QueryParam("inicio") @DefaultValue("0") Integer inicio,
    @QueryParam("tamanhoPagina") @DefaultValue("20")
    Integer tamanhoPagina)
```

Dessa maneira, se os parâmetros `inicio` e `tamanhoPagina` não forem informados pelo cliente, ele irá assumir os valores 0 e 20, respectivamente.

Considerando cinco usuários cadastrados no banco de dados, e as devidas modificações no código para incluir a paginação, uma requisição para `/usuarios?inicio=0&tamanhoPagina=2` traria os seguintes resultados:

```
<usuarios>
  <usuario>
    <id>1</id>
    <nome>Alexandre</nome>
    <login>admin</login>
    <senha>admin</senha>
  </usuario>
  <usuario>
    <id>2</id>
    <nome>Rafael</nome>
    <login>rafael</login>
    <senha>rafael</senha>
  </usuario>
</usuarios>
```

Note que, dessa forma, não temos como saber se existe uma próxima página ou qual é a última página etc. O modelo HATEOAS transformaria o caso da paginação para ficar da seguinte maneira:

```
<usuarios>
  <usuario>
    <id>1</id>
```



```

    <nome>Alexandre</nome>
    <login>admin</login>
    <senha>admin</senha>
  </usuario>
  <usuario>
    <id>2</id>
    <nome>Rafael</nome>
    <login>rafael</login>
    <senha>rafael</senha>
  </usuario>
  <link href="usuarios?inicio=0&tamanhoPagina=2"
    rel="primeiraPagina"/>
  <link href="usuarios?inicio=2&tamanhoPagina=2"
    rel="proximaPagina"/>
  <link href="usuarios?inicio=4&tamanhoPagina=2"
    rel="ultimaPagina"/>
</usuarios>

```

Repare que, agora, os links para que o cliente saiba quais passos a seguir estão presentes no corpo da próxima requisição. Os links apresentam os atributos `href` (para que o cliente saiba qual o endereço a ser seguido), `rel` (para saber qual a relação deste link com o recurso atual) e, opcionalmente, `type` (para que o cliente saiba qual o MIME type oferecido pelo link). Por exemplo, no caso da representação dos nossos usuários como imagens, o retorno da listagem de usuários pode ficar assim:

```

<usuarios>
  <usuario>
    <id>1</id>
    <nome>Alexandre</nome>
    <login>admin</login>
    <senha>admin</senha>
    <link href="usuarios/1" rel="imagem" type="image/*"/>
  </usuario>
  <usuario>
    <id>2</id>
    <nome>Rafael</nome>
    <login>rafael</login>
    <senha>rafael</senha>
    <link href="usuarios/2" rel="imagem" type="image/*"/>
  </usuario>

```

```

<link href="usuarios?inicio=0&tamanhoPagina=2"
      rel="primeiraPagina"/>
<link href="usuarios?inicio=2&tamanhoPagina=2"
      rel="proximaPagina"/>
<link href="usuarios?inicio=4&tamanhoPagina=2"
      rel="ultimaPagina"/>
</usuarios>

```

Para produzir essas alterações, no entanto, é necessário alterar nossas classes de modelo, para incluir os links a serem seguidos.

HATEOAS E A MODIFICAÇÃO DAS CLASSES DE MODELO

Alguns frameworks já têm suporte a HATEOAS, de maneira que estes links não sejam intrusivos em relação às classes de modelo. Todavia, este suporte ainda é difícil de ser utilizado mesmo nesses frameworks, e nada disso é padronizado pela especificação JAX-RS 1.0. Este é um recurso que está previsto para lançamento (ou seja, padronização) na especificação JAX-RS 2.0.

Por exemplo, a representação dos links é feita pela classe `Link`:

```

package com.knight.usuarios.modelos.rest;

// imports omitidos

@XmlType(namespace = "http://knight.com/links")
@XmlAccessorType(XmlAccessType.FIELD)
public class Link {

    @XmlAttribute
    private String href;

    @XmlAttribute
    private String rel;

    @XmlAttribute
    private String type;
}

```

```
public Link(String href, String rel) {
    this.href = href;
    this.rel = rel;
}

public Link(String href, String rel, String type) {
    this(href, rel);
    this.type = type;
}

public Link() {}

// getters e setters
}
```

Note que se trata de uma classe comum, a ser utilizada pelas nossas próprias classes de modelo. Por ser algo comum entre os modelos, vamos representar uma entidade REST com a seguinte interface:

```
package com.knight.usuarios.modelos.rest;

public interface RESTEntity {

    void adicionarLink(Link link);

}
```

Perceba que esta interface pode ser modificada para atender a outras necessidades, que sejam comuns a modelos REST.

Nossa classe `Usuarios`, ao implementar esta interface, fica assim:

```
//declaração de pacote e imports omitidos

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Usuarios implements RESTEntity {

    @XmlElement(name = "usuario")
    private Collection<Usuario> usuarios;

    @XmlElement(name = "link")
```

```

private Collection<Link> links;

public Usuarios() { }

public Usuarios(Collection<Usuario> usuarios, Link... links) {
    this.usuarios = usuarios;
    this.links = new ArrayList<>(Arrays.asList(links));
}

@Override
public void adicionarLink(Link link) {
    if (links == null)
        links = new ArrayList<>();
    links.add(link);
}

//getters e setters
}

```

Observe que, dessa forma, quando a classe `Usuarios` for serializada de volta para o cliente, ela vai conter os links que programarmos para serem transmitidos. O código da classe `UsuariosService` deve ser atualizado para utilizar a interface `UriInfo`, de maneira a conferir mais flexibilidade para nossa classe.

Por exemplo, para criar os links de paginação, é necessário descobrir o caminho onde o usuário está atualmente para criar o link. A partir da página atual e do número de usuários, faço alguns cálculos para descobrir quais são a primeira página, a página anterior à atual, a próxima e a última. O método de criação dos links de paginação fica assim:

```

private Link[] criarLinksUsuarios(UriInfo uriInfo,
    Integer tamanhoPagina, Integer inicio, Long numeroUsuarios) {
    Collection<Link> links = new ArrayList<>();

    double numeroUsuariosDouble = numeroUsuarios;
    double tamanhoPaginaDouble = tamanhoPagina;

    // Arrendondamento para cima, para fornecer o número certo de
    // páginas
    Long numeroPaginas = (long) Math.ceil(numeroUsuariosDouble
        / tamanhoPaginaDouble);
}

```

```
// O resultado da divisão será um int.
Long paginaAtual = new Long(inicio / tamanhoPagina);

Link linkPrimeiraPagina = new Link(
    UriBuilder.fromPath(uriInfo.getPath())
        .queryParams(PARAM_INICIO, 0)
        .queryParams(PARAM_TAMANHO_PAGINA, tamanhoPagina).build()
        .toString(), "primeiraPagina");
links.add(linkPrimeiraPagina);

//criação dos outros links

return links.toArray(new Link[] {});
}
```

Já o código alterado da listagem fica assim:

```
@GET
public Response listarUsuarios(
    @HeaderParam("If-Modified-Since") Date modifiedSince,
    @QueryParam(PARAM_INICIO) @DefaultValue("0") Integer inicio,
    @QueryParam(PARAM_TAMANHO_PAGINA) @DefaultValue("20")
    Integer tamanhoPagina,
    @Context UriInfo uriInfo) {
    Collection<Usuario> usuarios = em
        .createQuery("select u from Usuario u", Usuario.class)
        .setFirstResult(inicio).setMaxResults(tamanhoPagina.intValue())
        .getResultList();

    // Recuperamos o número de usuários presentes em nossa base
    // para que possamos realizar o cálculo de páginas
    Long numeroUsuarios = em.createQuery(
        "select count(u) from Usuario u", Long.class).getSingleResult();

    boolean atualizado = false;

    if (modifiedSince != null) {
        for (Usuario usuario : usuarios) {
            if (usuario.getDataAtualizacao().after(modifiedSince)) {
                atualizado = true;
                break;
            }
        }
    }
}
```

```
    }  
  } else {  
    // Se a data não tiver sido passada, deve considerar os recursos  
    // como 'mais atuais'  
    atualizado = true;  
  }  
  
  if (atualizado) {  
  
    for (Usuario usuario : usuarios) {  
      Link link = criarLinkImagemUsuario(usuario);  
      usuario.adicionarLink(link);  
    }  
  
    return Response.ok(  
      new Usuarios(usuarios, criarLinksUsuarios(uriInfo,  
        tamanhoPagina, inicio, numeroUsuarios))).build();  
  } else {  
    return Response.notModified().build();  
  }  
}
```

Por questões de tamanho, o código completo não será postado aqui. Caso você queira conferir, confira o repositório de código do livro: <http://github.com/alesaudate/soa>.

5.11 TESTANDO TUDO

Para que possamos desenvolver qualquer aplicação eficientemente, é necessário que tenhamos meios eficientes de testes. Com aplicações orientadas a serviço, isso não é diferente.

Para testar serviços desenvolvidos com REST, no entanto, devemos adaptar nossas estratégias. O SoapUI possui funcionalidades para desenvolvimento de testes de serviços REST; no entanto, eu não o considero a melhor ferramenta para isso. Existem três ferramentas que eu gosto de utilizar para realizar testes, cada qual com vantagens e desvantagens (que devem ser exploradas de acordo com o cenário). Essas ferramentas são: Poster, curl e a própria API do framework que estamos utilizando para desenvolvimento dos serviços.

Testes com Poster

O Poster é um *plugin* do Firefox. Como tal, deve ser instalado no próprio navegador (pode ser obtido em <https://addons.mozilla.org/pt-br/firefox/addon/poster/>). Uma vez instalado, é notável que é uma ferramenta de uso bem simples e intuitivo, como pode ser visto na imagem:

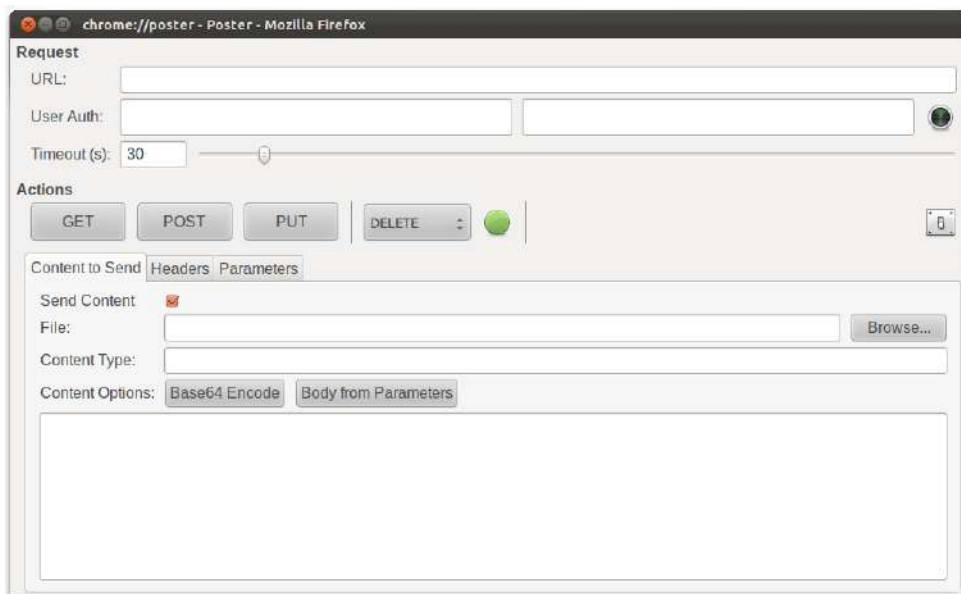


Figura 5.5: Poster

Na tela inicial, com a aba `Content to Send` habilitada, a área digitável mais abaixo da tela deve conter os dados a serem enviados no corpo da requisição. Por exemplo, para enviar um novo usuário para o serviço, essa área deve conter o XML a ser enviado para o serviço. Também é possível enviar dados de arquivos (como imagens, por exemplo), utilizando o campo `File`.

O ponto fraco do Poster é o tratamento em relação a dados binários. Quando uma imagem é solicitada, por exemplo, não se obtém a imagem em si, mas a representação binária da mesma. Essa combinação faz com que o Poster seja uma ferramenta fácil de utilizar, porém incompleta.

Testes com curl

Com curl, é possível escrever *scripts* (`.sh` no Linux e `.bat` no Windows) para realizar os testes. No Ubuntu, é possível instalar o curl utilizando o seguinte comando:

```
sudo apt-get install curl
```

Para outras plataformas, confira o site do programa: <http://curl.haxx.se/>.

Para utilizar o curl, basta escrever os comandos no terminal. Por exemplo, para obter a listagem de usuários, você pode utilizar o seguinte comando:

```
curl http://localhost:8080/soa-cap05-0.0.1-SNAPSHOT/services/usuarios
```

Note que, desta forma, o `curl` assumiu, automaticamente, o uso do método GET para obtenção dos dados. Caso queiramos utilizar um método diferente, basta adicionar o parâmetro `-X`; se quisermos adicionar dados, podemos utilizar o parâmetro `--data`, seguido dos dados (se esses dados forem precedidos por `@`, ele assume que deve enviar os dados de um arquivo).

Ainda, para adicionar cabeçalhos, podemos utilizar o parâmetro `-H`. Por exemplo, para testar a atualização de imagens, podemos utilizar o seguinte comando:

```
curl -X PUT --data @saudate.png -H"Content-Type: image/png"  
http://localhost:8080/soa-cap05-0.0.1-SNAPSHOT/services/usuarios/1
```

Desta forma, ele fará o envio da imagem referenciada pelo arquivo `saudate.png` para a URL <http://localhost:8080/soa-cap05-0.0.1-SNAPSHOT/services/usuarios/1>, utilizando o método POST. Repare, também, que ajustamos o cabeçalho `Content-Type` para ser do tipo `image/png`.

A desvantagem do curl é que, por ser uma aplicação de linha de comando, deve-se executar à parte do programa (ou seja, realizar a instalação e executar manualmente os testes) ou adaptar o próprio *script* de *deploy* para executar o programa (ferramentas de integração contínua como o Jenkins também podem ser adaptadas para executar este tipo de *script*). Ainda assim, é uma boa pedida para automatizar seus testes de integração.

Testes com a API do RESTEasy e Maven

Uma boa abordagem para testar seus serviços pode ser utilizar a própria API do *framework* que você está utilizando para desenvolver seus serviços (no nosso caso, RESTEasy). Quase todos os *frameworks* Java para desenvolvimento de serviços REST

do lado do servidor também possuem APIs para desenvolvimento do lado do cliente, e é uma boa alternativa combinar essa capacidade com o Maven, que possui um sistema de execução de testes integrados embutido.

Tenha em mente que os testes de que estamos falando até o momento são testes simples para verificar a funcionalidade do sistema. O que o Maven possui embutido são testes de integração automatizados e “repetíveis”, ou seja, eles sempre serão executados de maneira automatizada.

Para habilitar esse tipo de teste no Maven, basta incluir o seguinte trecho no arquivo `pom.xml`:

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-failsafe-plugin</artifactId>
      <version>2.6</version>
      <executions>
        <execution>
          <goals>
            <goal>integration-test</goal>
            <goal>verify</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <!-- Outros plugins -->
  </plugins>
</build>
```

A seguir, basta incluir uma classe com nomes que terminam com `IT` ou `ITCase` na pasta de fontes de testes (`src/test/java`), devidamente configurado de acordo com nosso framework de testes. Por exemplo, a classe de testes do serviço de usuários (configurada para utilizar o JUnit) pode ter o seguinte código:

```
package com.knight.usuarios.servicos.testes.integração;

// imports omitidos

public class UsuariosServiceIT {

    public static String SERVICES_CONTEXT =
```

```

        "http://localhost:8080/soa-cap05-0.0.1-SNAPSHOT/services";

    public static String USUARIOS_CONTEXT =
        SERVICES_CONTEXT + "/usuarios";

    private byte[] fotoSaudate;

    @Before
    public void setup() throws IOException {
        // Ajustes iniciais, como a carga da imagem, por exemplo
    }

    @Test
    public void testRecepcaoImagens() throws Exception {
        // códigos dos testes
    }
}

```

Estes testes serão realizados utilizando a API do próprio *framework*, assim como qualquer cliente.

5.12 PROGRAMANDO CLIENTES

A API do RESTEasy provê facilidades para desenvolver clientes de serviços tanto desenvolvidos em Java quanto em qualquer outra linguagem. A diferença entre as duas abordagens é que, caso o serviço tenha sido feito em Java, é possível gerar um *proxy* baseado em uma interface para realizar essa comunicação, de maneira bastante semelhante a clientes remotos EJB, por exemplo. No entanto, note que esta abordagem promove alto acoplamento com a API.

Caso você queira realizar o desenvolvimento de seus clientes Java utilizando esta abordagem, basta extrair os métodos da sua classe de serviço para uma interface (incluindo as anotações JAX-RS). Você pode, inclusive, remover estas anotações da implementação, deixando sua classe de serviço mais limpa. A interface vai conter, então, vários métodos relacionados a formas de pesquisa, alguns para modificação de dados e outros para tratamento de imagens.

A seção de tratamento de buscas contém os métodos `listarUsuarios` e `find`, da seguinte forma:

```
package com.knight.usuarios.servicos;

// imports omitidos

@Path("/usuarios")
@Produces(MediaType.APPLICATION_XML)
@Consumes(MediaType.APPLICATION_XML)
public interface UsuariosServiceInterface {

    static final String PARAM_INICIO = "inicio";

    static final String PARAM_TAMANHO_PAGINA = "tamanhoPagina";

    @GET
    public Response listarUsuarios(
        @HeaderParam("If-Modified-Since") Date modifiedSince,
        @QueryParam(PARAM_INICIO) @DefaultValue("0") Integer inicio,
        @QueryParam(PARAM_TAMANHO_PAGINA) @DefaultValue("20")
        Integer tamanhoPagina,
        @Context UriInfo uriInfo);

    @GET
    @Path("/{id}")
    public Response find(@PathParam("id") Long id,
        @HeaderParam("If-Modified-Since") Date modifiedSince);

    // outros métodos
}
```

Observe que os métodos de busca são aqueles que têm maior complexidade, por aceitarem diversas formas. No nosso caso, ainda poderíamos expandir as possibilidades para uma busca por exemplos, por *ranges* específicos de dados etc.; para fins de simplicidade, mantemos apenas estes dois.

Os próximos métodos presentes na interface são os relativos a modificação de dados — inserção, atualização e remoção. Note que os métodos de atualização e remoção permitem que o ID do usuário a ser removido venha tanto na URL quanto diretamente no corpo da requisição (ou seja, dentro do XML representando o usuário):

```
@POST
public Response create(@Context UriInfo uriInfo, Usuario usuario);
```

```

@PUT
public Response update(Usuario usuario);

@PUT
@Path("/{id}")
public Response update(@PathParam("id") Long id, Usuario usuario);

@DELETE
public Response delete(Usuario usuario);

@DELETE
@Path("/{id}")
public Response delete(@PathParam("id") Long id);

```

Por último, temos os métodos de manipulação de imagens:

```

public static final String CAMPO_DESCRICAO_IMAGEM = "Descricao";

@PUT
@Path("/{id}")
@Consumes("image/*")
public Response adicionarImagem(
    @HeaderParam(CAMPO_DESCRICAO_IMAGEM) String descricao,
    @PathParam("id") Long idUsuario,
    @Context HttpServletRequest httpRequest, byte[]
        dadosImagem);

@GET
@Path("/{id}")
@Produces("image/*")
public Response recuperarImagem(@PathParam("id") Long id,
    @HeaderParam("If-Modified-Since") Date modifiedSince);
}

```

Com esta interface, o teste com RESTEasy pode ser feito através da criação de um *proxy*. Para isso, basta utilizar a classe `org.jboss.resteasy.client.ProxyFactory`, passando como parâmetro o endereço da aplicação e a interface do serviço. Dessa forma, o seguinte código pode ser criado:

```
UsuariosServiceInterface usuariosService = ProxyFactory.create(
    UsuariosServiceInterface.class,
    "http://localhost:8080/soa-cap05-0.0.1-SNAPSHOT/services");
```

Os métodos de serviço podem ser testados como se estivessem sendo executados localmente. Por exemplo, caso nossa classe de serviço retorne um POJO, esse POJO é automaticamente mapeado. Suponha que nosso serviço de listagem de usuários retorne uma instância da classe `Usuarios`. Dessa maneira, o seguinte código poderia ser utilizado:

```
Usuarios usuarios = usuariosService.listarUsuarios();
```

Não é o caso dos nossos serviços. Todas as entidades estão mapeadas com `Response`. Neste caso, nossa invocação deve esperar uma resposta desse tipo. Para evitar problemas com `cast` de retorno, devemos fazer o `cast` dessa resposta para a classe `org.jboss.resteasy.client.ClientResponse`, que contém métodos que facilitam a recuperação dos dados.

Lembre-se que nosso método de listagem de usuários, por exemplo, possuía quatro parâmetros: o `header If-Modified-Since`, dois parâmetros para promover paginação e `UriInfo`, para que possamos recuperar dados do contexto da invocação. Não temos acesso a este último pelo cliente, então, deixaremos sempre em branco (`null`). Dessa forma, a invocação para este método fica assim:

```
ClientResponse<Usuarios> response =
    (ClientResponse<Usuarios>) usuariosService
        .listarUsuarios(null, null, null, null);
```

Uma vez que temos acesso a essa resposta, devemos testar o status da resposta. Caso seja 200, podemos recuperar os usuários utilizando o método `getEntity`, passando como parâmetro a classe para qual deve ser feito o `cast` da resposta (no caso, `Usuarios`):

```
Usuarios usuarios = response.getEntity(Usuarios.class);
```

Assim, podemos interagir normalmente com nosso serviço; no entanto, se quisermos, também podemos ter uma interação avançada com serviços REST feitos em outras linguagens, também. Para isso, utilizamos uma abordagem diferente: a classe `org.jboss.resteasy.client.ClientRequest.ClientRequest` (no caso de utilizarmos `RESTEasy` como *framework*). Essa classe recebe como parâmetro para o construtor um *template* de URI (semelhante ao utilizado no lado do servidor). Para popularmos esse *template*, utilizamos o método `pathParameters`.

Uma vez criado o formato da requisição, basta executar o método (da classe) de nome igual ao método HTTP (GET se torna `get()`, POST se torna `post()`, e assim por diante). Podemos passar como parâmetro para este método a classe a ser utilizada para ajustar o tipo de retorno. Por exemplo, para executarmos a busca por um usuário específico, podemos executar o seguinte código:

```
ClientResponse<Usuario> response = new ClientRequest(
    "http://localhost:8080/soa-cap05-0.0.1-SNAPSHOT" +
    "/services/usuarios/{id}").pathParameters(1).get(Usuario.class);

Usuario usuario = response.getEntity();
```

Podemos também ajustar o tipo de conteúdo aceito utilizando o método `accept()`, o método `header()` para ajustar *headers* HTTP e o método `body` para ajustar o corpo da requisição. Por exemplo, para solicitar a representação do usuário como imagem, utilizamos o seguinte código:

```
ClientResponse<byte[]> response = new ClientRequest(
    "http://localhost:8080/soa-cap05-0.0.1-SNAPSHOT/" +
    "services/usuarios/{id}")
    .pathParameters(1).accept("image/*").get(byte[].class);

byte[] imagem = response.getEntity();
```

E para inserir uma nova imagem:

```
ClientResponse<?> response = new ClientRequest(
    "http://localhost:8080/soa-cap05-0.0.1-SNAPSHOT/" +
    "services/usuarios/{id}")
    .pathParameters(1).header("Descricao", "descricao da imagem")
    .body("image/png", arrayDeBytesContendoUmaFoto)
    .put();
```

Assim, nosso cliente já está flexível o bastante para tratar serviços tanto desenvolvidos em Java quanto em qualquer outra linguagem, concluindo nosso caso de uso. Também é possível escrever clientes em Javascript para serviços REST, tornando possível desenvolver interfaces (páginas web) completamente desacopladas do *backend* da aplicação. Não faz parte do escopo deste livro tratar desses casos. Caso deseje, consulte livros especializados em JavaScript/jQuery para saber mais a respeito.

5.13 SUMÁRIO

Neste capítulo, você viu em detalhes como desenvolver, consumir e testar serviços REST. Nosso caso de uso para isso foi a criação, recuperação, atualização e deleção de usuários, com variações para cobrir casos como paginação, tratamento de imagens etc.

Para isso, você viu desde o começo os princípios do funcionamento do protocolo HTTP. Você conferiu em detalhes como funcionam os códigos de status, *headers*, tipos MIME e outros detalhes do protocolo. Dessa forma, você obteve embasamento sobre a base de REST. Depois, você conferiu os detalhes de REST — funcionamento de recursos, tipos MIME, HATEOAS e outros.

Ao final, você conferiu como realizar a criação de clientes, tanto para serviços desenvolvidos na própria linguagem Java quanto em outras linguagens, tornando-os tanto fáceis de desenvolver, porém acoplados, como mais difíceis, porém flexíveis e com capacidades de comunicação com outras linguagens.

Ainda resta saber como utilizar a criação desses usuários na segurança de nossos aplicativos. No próximo capítulo, você verá como realizar o tratamento de segurança em serviços, tanto utilizando WSDLs quanto os modelos em REST.

CAPÍTULO 6

Segurança em web services

“Há mais coisas entre o céu e a terra, Horácio, do que sonha a nossa vã filosofia”

– William Shakespeare

Finalmente, seu sistema de gerenciamento de usuários está completo. Mas fica a dúvida, como utilizar estes serviços através dos seus sistemas? Vários desafios estão presentes neste momento:

- Como prevenir o roubo dessas informações?
- Como prevenir a criação de dados de maneira não-autorizada?
- Como prevenir a alteração desses dados de maneira não-autorizada?
- Como impedir um atacante de usar informações legítimas?
- e outros...

Neste momento, portanto, sua principal tarefa é proteger seu sistema de gerenciamento de usuários. No entanto, antes, você precisa saber quais problemas você deseja evitar.

Pela natureza cliente-servidor de aplicações SOA, seu problema principal é proteger o mecanismo de transporte (já que a segurança do lado do servidor é questão de proteção em outros níveis — por exemplo, usuários com acesso ao sistema de arquivos do sistema operacional — não cobertos por este livro). Como estamos usando o protocolo HTTP como transporte, vamos utilizar o mecanismo padrão de proteção, o HTTPS. Contudo, vamos ver primeiro quais os problemas de segurança dos quais devemos nos proteger.

6.1 ATAQUES: MAN-IN-THE-MIDDLE E EAVESDROPPING

Quando utilizamos a internet (e o protocolo HTTP, em geral), nossas requisições geralmente passam por vários equipamentos de rede. Muitas vezes, não sabemos ao certo quais eles são, muito menos suas intenções. Ao executar o comando `tracert` no Windows (ou `traceroute` no Linux), temos uma breve visão desses caminhos. Ao usar esse caminho apontado para `www.google.com.br`, por exemplo, podemos obter resultados como os seguintes (dados ligeiramente alterados):

```
tracert to www.google.com.br (190.98.170.25), 30 hops max,
  60 byte packets
 1  my.router (192.168.1.1)  4.938 ms  5.028 ms  5.023 ms
 2  * * *
 3  201-0-90-86.dsl.telesp.net.br (201.0.90.86)  13.770 ms  13.863 ms
    20.184 ms
 4  201-0-6-5.dsl.telesp.net.br (201.0.6.5)  8.062 ms
 5  187-100-53-13.dsl.telesp.net.br (187.100.53.13)  72.049 ms
 6  213.140.51.113 (213.140.51.113)  38.417 ms
 7  Xe9-0-6-0-grtsanem1.red.telefonica-wholesale.net (84.16.15.25)
    17.514 ms
```

Esta é uma listagem parcial do caminho percorrido pelos pacotes entre o meu computador até o site do Google. No entanto, a certo ponto as informações começam a ficar nebulosas, com o aparecimento de servidores desconhecidos, como é o caso do servidor de número 6, na listagem acima, que não possui um endereço DNS (ou seja, não tem nome).

Essas situações ocorrem por conta da própria natureza da internet, que é a de uma rede distribuída, sem um controlador determinado. Desta forma, foge ao nosso

controle determinar quais recursos estão ou não autorizados a controlar o redirecionamento dos pacotes de rede, e mais ainda, controlar a leitura destes recursos. Esse problema acaba sendo a raiz de dois ataques na internet: *man-in-the-middle* e *eavesdropping*.

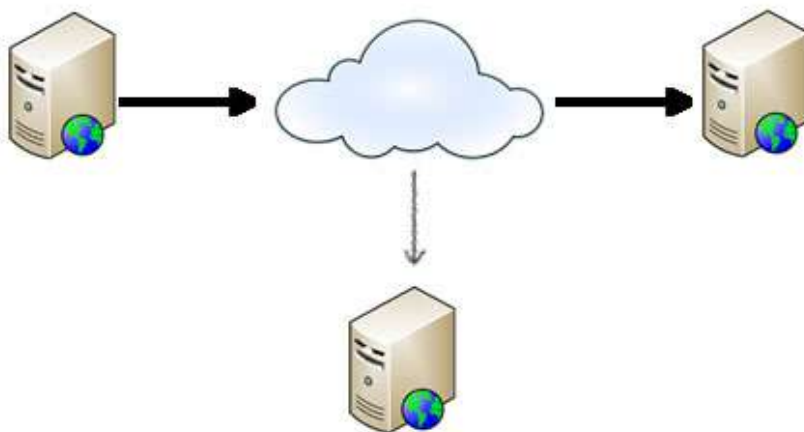


Figura 6.1: Atacante interceptando requisições

Eavesdrop, em uma tradução literal do inglês, significa ouvir escondido, e remete a ações como ouvir por detrás de uma porta ou utilizar um grampo telefônico. Em computação, significa a pura e simples interceptação de mensagens e ler seu conteúdo — sem, no entanto, alterá-lo. Este ataque pode ser utilizado para interceptar senhas, por exemplo, para que o atacante utilize em mensagens separadas (subsequentes à mensagem interceptada).

Já o ataque *man-in-the-middle* é uma interceptação controlada: um atacante intercepta as chamadas e modifica os dados, de maneira que tanto o cliente quanto o servidor não sabem que a conexão está sendo controlada por um terceiro ator. Pode ser utilizada para alterar o número de volumes comprados em nossa livreria, por exemplo.

INTERCEPTAÇÃO DE MENSAGENS E SNIFFERS

Não é necessário ir longe para verificar o estado de suas conexões. Softwares como o Wireshark (<http://www.wireshark.org/>) podem realizar essa interceptação de dados sem muito esforço. Este é um famoso *sniffer* (ou seja, software de rastreamento e escuta), disponível para Windows e Mac.

6.2 PROTEÇÃO CONTRA INTERCEPTAÇÃO COM HTTPS

A solução mais usada para o problema da interceptação é o uso de HTTPS (*Hypertext Transfer Protocol over Secure Sockets Layer*), e baseia-se na aplicação de uma camada de segurança com SSL (*Secure Socket Layer*) sobre o HTTP. Esta camada utiliza certificados digitais para encriptar as comunicações e garantir a autenticidade do servidor e, opcionalmente, do cliente.

Estes certificados são amplamente utilizados na internet. Toda vez que você acessa uma URL cujo endereço começa com `https`, este mecanismo está presente. Por exemplo, ao acessar o Gmail (cujo endereço real está presente em <https://accounts.google.com>), é possível visualizar as informações do certificado:



Figura 6.2: Informações do certificado Google

Este certificado é usado para que o cliente confirme a autenticidade do servidor (pelas informações contidas na seção **Emitido para**) e criptografe as informações utilizando-o. Para que o cliente possa ter certeza da confiabilidade desse certificado, ele pode ser emitido por uma autoridade certificadora (*Certification Authority* — CA). Esta autoridade é bem conhecida entre os clientes, que confiam na mesma. Em alguns casos, o certificado também pode ser autoassinado, ou seja, emitido pelo próprio site — o que o torna menos confiável.

Da mesma forma, estes certificados também podem ser levantados pelo cliente para que o servidor confirme a identidade do cliente — mais ou menos do mesmo

modo que uma autenticação utilizando usuário e senha. Esta maneira, no entanto, é mais complicada de ser utilizada na internet e, em geral, é utilizada apenas em casos muito específicos.

Ao utilizar certificados digitais, o servidor provê um mecanismo de criptografia baseado em chaves públicas e privadas. Este mecanismo é baseado em algoritmos que utilizam duas chaves, uma pública e uma privada. A chave pública é utilizada para encriptar dados; e a privada, para decriptar. Elas recebem estes nomes porque a ideia geral é que a chave pública seja conhecida por todos os clientes de um determinado serviço, e a privada, apenas pelo provedor do serviço.

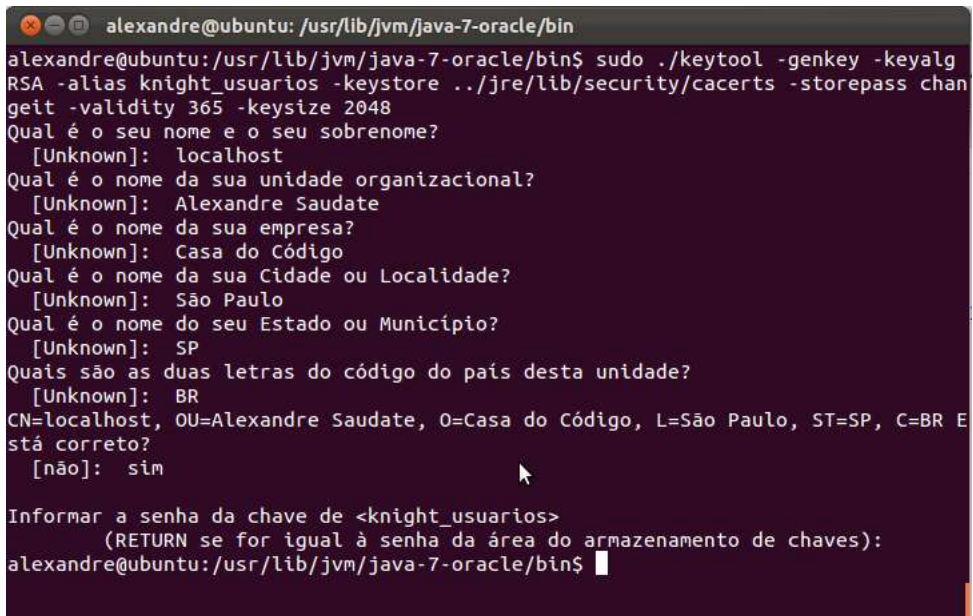
A chave pública dos certificados digitais são geralmente utilizadas para encriptar um outro tipo de chaves, que são conhecidas como chaves **simétricas**. Estas chaves, por sua vez, são as mesmas tanto para encriptar quanto para decriptar dados. Com o sistema de chave pública e privada, o cliente e o servidor têm a chance de negociar esta chave simétrica — que é utilizada nas duas pontas para encriptar as mensagens.

A linguagem Java provê um sistema próprio para armazenamento de certificados, chamado JKS - *Java Key Store*. Um arquivo JKS contém uma coleção de certificados, cada um devidamente identificado por um *alias*. Esses certificados podem ser criados e armazenados utilizando uma ferramenta chamada `keytool`, presente na JDK.

Para utilizar o JKS, você deve navegar até a pasta `bin` da sua JDK, além de possuir uma série de informações, como o domínio da web onde ele será utilizado, o período durante o qual esse certificado será válido e o algoritmo que será utilizado para encriptar e decriptar seus dados. Por exemplo, para gerar um certificado válido durante um ano, com algoritmo RSA (com comprimento de chave de 2048 bits) e válido para o endereço `localhost`, o seguinte código pode ser utilizado:

```
keytool -genkey -keyalg RSA -alias knight_usuarios -keystore
../jre/lib/security/cacerts -storepass changeit -validity 360
-keysize 2048
```

Este comando levará a uma série de questionamentos em relação à empresa. Estes dados podem ser populados normalmente, de acordo com as questões, exceto o primeiro (em uma JDK em português, a pergunta é “Qual é o seu nome e sobrenome?”). Neste campo, **deve ser inserido o domínio onde a aplicação será hospedada** — no nosso caso, `localhost`.



```
alexandre@ubuntu: /usr/lib/jvm/java-7-oracle/bin
alexandre@ubuntu: /usr/lib/jvm/java-7-oracle/bin$ sudo ./keytool -genkey -keyalg
RSA -alias knight_usuarios -keystore ../jre/lib/security/cacerts -storepass chan
geit -validity 365 -keysize 2048
Qual é o seu nome e o seu sobrenome?
[Unknown]: localhost
Qual é o nome da sua unidade organizacional?
[Unknown]: Alexandre Saudate
Qual é o nome da sua empresa?
[Unknown]: Casa do Código
Qual é o nome da sua Cidade ou Localidade?
[Unknown]: São Paulo
Qual é o nome do seu Estado ou Município?
[Unknown]: SP
Quais são as duas letras do código do país desta unidade?
[Unknown]: BR
CN=localhost, OU=Alexandre Saudate, O=Casa do Código, L=São Paulo, ST=SP, C=BR E
stá correto?
[não]: sim
Informar a senha da chave de <knight_usuarios>
(RETURN se for igual à senha da área do armazenamento de chaves):
alexandre@ubuntu: /usr/lib/jvm/java-7-oracle/bin$
```

Figura 6.3: Questionamentos em relação à criação do certificado

6.3 USANDO MECANISMOS DE AUTENTICAÇÃO HTTP

Em aplicações que utilizam segurança, geralmente é necessário fornecer mecanismos para que o cliente possa se identificar. A partir desta identificação, o servidor checa se as credenciais fornecidas são válidas e quais direitos estas credenciais dão a ele. Estes procedimentos recebem o nome de autenticação (ou seja, o cliente fornecendo credenciais válidas) e autorização (o servidor checando quais direitos o cliente tem).

O protocolo HTTP oferece dois tipos de autenticação: *basic* e *digest*. A autenticação *basic* é realmente muito simples; consiste em passar o *header* HTTP *Authorization* obedecendo ao seguinte algoritmo:

```
Basic Base64({#usuário}:{#senha})
```

Considerando usuário `alexandre` e senha `alexandre`, o cabeçalho passaria a ter o seguinte valor:

```
Authorization: Basic YWxleGFuZHZHJlOmFsZXhhbmRyZQ==
```

Note que este sistema não apresenta muita segurança: caso um atacante intercepte esta requisição, o algoritmo Base64 é facilmente reversível, levando aos dados originais.

O ALGORITMO BASE64

O algoritmo Base64 é amplamente conhecido pela internet. Apesar de ser (tecnicamente falando) um algoritmo de criptografia, ele é mais utilizado na internet para tráfego de dados binários (já que converte qualquer espécie de formato binário em código alfanumérico). Para mais informações a respeito, leia <http://tools.ietf.org/html/rfc989>, seção 4.3.

Desta forma, a maior parte dos sistemas que utilizam autenticação *basic*, utilizam também HTTPS, para que um atacante não descubra o usuário e senha que estão sendo trafegados. Mesmo assim, a requisição fica exposta a interceptações, que um atacante pode usar para reenviar uma requisição como se fosse o cliente legítimo. Apesar de o atacante não saber exatamente o que está enviando, ele pode perfeitamente reenviar uma requisição que já havia sido enviada anteriormente pelo cliente legítimo.

Para contornar estes e outros problemas, o algoritmo *digest* pode ser utilizado. Este algoritmo usa várias informações além do usuário para realizar a autenticação: uma *string* aleatória (conhecida como *nonce*), uma *string* contendo o nível de proteção (conhecida como *qop*), o contexto do servidor (conhecido como *realm*), e outros parâmetros opcionais, que são fornecidos pelo servidor após uma primeira tentativa de obter um recurso protegido.

Suponha a seguinte interação: um cliente tenta acessar uma URI `/services/usuarios`, utilizando o método `GET`. Ao fazer isso (sem autenticação alguma presente), ele recebe a seguinte resposta:

```
401 Unauthorized
WWW-Authenticate: Digest realm="MyRealm",
    qop="auth",
    nonce="0cc194d2c0f4b6b765d448a578773543",
    opaque="92ea7eadd41225f43d534bc544c2223"
```

Essa resposta é chamada de *desafio*. O servidor, para se certificar de que o cliente é quem diz ser, manda uma descrição do contexto onde o cliente deve se autenticar (neste caso, `MyRealm`), o nível de qualidade da proteção (neste caso `auth`)

e uma *string* aleatória, que será utilizada como garantia de que a requisição utilizada não passará pelos chamados *replay attacks*, ou seja, um atacante não interceptará a requisição e reenviará, como se fosse o cliente legítimo.

Para calcular a resposta do desafio, o cliente tem um trabalho consideravelmente maior. Tendo como parâmetro os dados do desafio acima e usuário e senha alexandre, o cálculo a ser feito é o seguinte:

```
#método=GET
#path=/services/usuarios

#realm=MyRealm
#nonce=0cc194d2c0f4b6b765d448a578773543
#qop=auth

#contador=00000001
#nonceDoCliente=4a7a0b7765d657a756e2053d2822
#usuário=alexandre
#senha=alexandre

a1 = HEXADECIMAL(MD5({#usuário}:{#realm}:{#senha}))
a2 = HEXADECIMAL(MD5({#método}:{#path}))
a3 = HEXADECIMAL(MD5({a1}:{#nonce}:{#contador}:
    {#nonceDoCliente}:{#qop}:{a2}))

// Valor de a3: f08528d7e052986e6563a905ab5b1b7b
```

O ALGORITMO MD5

O algoritmo MD5, assim como Base64, também é amplamente conhecido. Todavia, não é um algoritmo de criptografia, e sim, de *hash*. Por ter essa natureza, uma vez calculado, não pode ser revertido. Para mais informações, consulte <http://www.ietf.org/rfc/rfc1321.txt>.

Com a variável `a3`, o cliente pode “devolver” a requisição com a devida autorização:

```
GET /services/usuarios HTTP/1.1
Host: localhost
Authorization: Digest username="alexandre",
```

```
realm="MyRealm",
nonce="0cc194d2c0f4b6b765d448a578773543",
uri="/services/usuarios",
qop=auth,
nc=00000001,
cnonce="4a7a0b7765d657a756e2053d2822",
response="f08528d7e052986e6563a905ab5b1b7b",
opaque="92ea7eedd41225f43d534bc544c2223"
```

Com esses dados, o servidor calcula o *hash* da mesma forma. Caso a resposta obtida seja igual, o cliente está autorizado; caso contrário, não. Note que o servidor precisa ter conhecimento do usuário e senha (além dos dados passados pelo cliente) para ter condições de calcular esse *hash*. Sem que o mesmo tenha conhecimento, antecipadamente, de qual é o usuário e senha, não é possível calculá-lo. Além disso, um possível interceptador da requisição não consegue reenviar a mesma informação, já que elementos aleatórios são introduzidos na mensagem e descartados quando utilizados.

6.4 HABILITANDO SEGURANÇA EM APLICAÇÕES REST

Tendo estes conceitos em mente, você é plenamente capaz de aplicá-los em seus serviços REST. Lembre-se de que, por tratar-se de serviços baseados em HTTP, os conceitos de segurança aplicam-se a REST, também. Então, da mesma forma como você habilitaria segurança em páginas JSP ou Servlets, você habilita a segurança em REST.

Para isso, a primeira coisa a ser feita é fazer com que seu servidor JBoss enxergue o certificado que você criou anteriormente. Apesar de não ser o recomendado, você pode fazer essa edição diretamente no arquivo `standalone.xml`.

EDIÇÃO DE PARÂMETROS JBOSS

O modo mais recomendado para desenvolvedores editarem configurações no servidor JBoss é pela interface gráfica, presente em `http://<endereço-da-máquina>:9990/console`. Se a configuração desejada não for acessível pela interface, deve-se utilizar o CLI (*Command-Line Interface*). Para mais detalhes, consulte a documentação.

Para instalar o certificado no servidor, você deve encontrar a definição do subsistema `web` e definir um conector HTTPS. Para encontrar o subsistema, utilize um

editor de texto e ache a entrada `urn:jboss:domain:web:1.1`. A definição do conector fica assim:

```
<connector name="https" protocol="HTTP/1.1" scheme="https"
  socket-binding="https" secure="true" />
```

Não se preocupe com essas informações. A única referência a dados localizados na própria configuração está no atributo `socket-binding`, mas essa informação já está presente na configuração.

O próximo passo é configurar este conector, para que toda vez que um recurso HTTPS for acessado, o certificado criado seja utilizado. Para isso, basta utilizar a definição do certificado como a seguinte:

```
<connector name="https" protocol="HTTP/1.1" scheme="https"
  socket-binding="https" secure="true">
  <ssl name="ssl" key-alias="knight_usuarios"
    certificate-key-file=
      "<localização da JDK>/jre/lib/security/cacerts"
    protocol="TLSv1"
    ca-certificate-file=
      "<localização da JDK>/jre/lib/security/cacerts"/>
</connector>
```

Note que esta definição utiliza o mesmo *alias* do certificado criado, com a localização do mesmo. Para fins de testes, é possível utilizar o repositório já embarcado na JDK (ou seja, o `cacerts`). No entanto, é altamente recomendável criar um repositório separado, para que o gerenciamento destes certificados seja facilitado.

Além disso, uma configuração adicional deve ser inserida no subsistema de segurança do JBoss (que pode ser localizado utilizando a entrada `urn:jboss:domain:security:1.1`). É necessário colocar algumas informações relativas ao certificado de segurança também no contexto da aplicação *web*, de acordo com o seguinte:

```
<subsystem xmlns="urn:jboss:domain:security:1.1">
  <security-domains>
    <!-- Configurações de domínios de segurança -->
    <security-domain name="jboss-web-policy" cache-type="default">
      <authorization>
        <policy-module code="Delegating" flag="required"/>
      </authorization>
    </security-domain>
  </security-domains>
</subsystem>
```

```

<jsse keystore-password="changeit"
      keystore-url=
        "<localização da JDK>/jre/lib/security/cacerts"
      truststore-password="changeit"
      truststore-url=
        "<localização da JDK>/jre/lib/security/cacerts"
      server-alias="knight_usuarios" protocols="TLS"/>
</security-domain>
</subsystem>

```

Substitua os locais onde encontra a localização da JDK pelo caminho da sua instalação. Por exemplo, se sua JDK estiver instalada em `/usr/lib/jvm/java-7-oracle` (no caso de um sistema operacional Linux), então o caminho do arquivo `cacerts` será `/usr/lib/jvm/java-7-oracle/jre/lib/security/cacerts`. Em caso de um sistema operacional Windows, será algo similar a `C:\Program Files\Java\jdk1.7.0_07\jre\lib\security\cacerts`. A senha (referenciada na listagem como `changeit`

Feito isso, seu JBoss está pronto para responder as requisições com o certificado certo.

O próximo passo é configurar os serviços para que eles só possam atender requisições utilizando HTTPS. Para isso, é necessário editar o arquivo `web.xml`, descrevendo a URL que será protegida, o modelo de autenticação, o sistema de proteção da camada de transporte e o papel que o cliente que acessar esta URL deverá ter. Para isso, descrevemos o papel como `admin`, a URL como qualquer uma a partir de `/services` (ou seja, todos os serviços deste pacote) e utilizando qualquer método HTTP. Além disso, informamos que a camada de transporte será protegida por HTTPS e autenticação `BASIC`. Desta forma, o seguinte conteúdo deve ser inserido:

```

<security-role>
  <description>Administrador</description>
  <role-name>admin</role-name>
</security-role>

<security-constraint>
  <web-resource-collection>
    <web-resource-name>Todos os serviços</web-resource-name>
    <url-pattern>/services/*</url-pattern>
    <http-method>GET</http-method>
  
```

```
<http-method>POST</http-method>
<http-method>PUT</http-method>
<http-method>DELETE</http-method>
<http-method>HEAD</http-method>
<http-method>OPTIONS</http-method>
</web-resource-collection>
<auth-constraint>
  <role-name>admin</role-name>
</auth-constraint>
<user-data-constraint>
  <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
</security-constraint>

<login-config>
  <auth-method>BASIC</auth-method>
</login-config>
```

A primeira seção desta configuração (`security-role`) declara o papel `admin`. A segunda parte, `security-constraint`, determina que todos que quiserem acessar a URL `/services` (e subURLs) utilizando os métodos `GET`, `POST`, `PUT`, `DELETE`, `HEAD` e `OPTIONS` deverão ter o papel `admin`. Além disso, esta seção determina que o protocolo de transporte deve ser seguro (ou seja, deve utilizar HTTPS). Por último, a seção `login-config` determina que as URLs protegidas deverão usar autenticação `BASIC`.

O próximo passo é determinar como o servidor realizará a obtenção dos usuários. O mecanismo padrão para realizar este tipo de tarefas em Java é o JAAS — *Java Authentication and Authorization Service*. No JBoss 7, para realizar a declaração do mecanismo, basta acrescentar um domínio de segurança no subsistema de segurança, que pode ser encontrado realizando uma busca por `urn:jboss:domain:security:1.1`.

Vários tipos de sistemas de login são aceitos pelo JAAS. O Jboss 7 já vem pré-configurado para realizar armazenamento de usuários em arquivos de propriedades; no entanto, gostaríamos de manter nossos dados armazenados em banco de dados, já que se trata de um mecanismo mais flexível. Para realizar essa definição, contudo, precisamos de algumas informações sobre esse banco, tais como: nome JNDI do *datasource*, *query* para localização da senha, *query* para localização de *roles* de segurança, algoritmo de *hash* utilizado para armazenagem e algoritmo para codificação da senha.

No nosso caso, utilizaremos um *datasource* que será instalado pela própria aplicação, de nome JNDI `java:jboss/datasources/KnightUsuariosDS`. Utilizaremos, para armazenamento de nossos usuários e senhas, uma tabela chamada `usuario_privilegiado`, com colunas `login` e `senha`, respectivamente. O algoritmo de *hash* utilizado para armazenar as senhas será o `SHA-256`, e o algoritmo de codificação, `Base64`. Vários valores são aceitos como algoritmo de *hash*, como `MD5` e `SHA-1`. Isso é dependente da JVM, de maneira que é possível instalar novos algoritmos. Assim, a definição do domínio de segurança ficaria assim:

```
<security-domain name="MyRealm">
  <authentication>
    <login-module code="Database" flag="required">
      <module-option name="dsJndiName"
        value="java:jboss/datasources/KnightUsuariosDS"/>
      <module-option name="principalsQuery"
        value="
          select senha from usuario_privilegiado where login=?"/>
      <module-option name="rolesQuery"
        value="select 'admin','Roles' from
          usuario_privilegiado where login=?"/>
      <module-option name="hashAlgorithm" value="SHA-256"/>
      <module-option name="hashEncoding" value="base64"/>
    </login-module>
  </authentication>
</security-domain>
```

O QUE SIGNIFICAM OS PARÂMETROS NA QUERY DE ROLES?

Como você pôde notar, a *query* para seleção de papéis deve buscar duas colunas. A primeira irá conter a *role* em si (no caso, sempre será `admin`). A segunda contém uma descrição da *role*, e é comum manter `Roles`.

Finalmente, para realizar a ligação da aplicação com o domínio de segurança, basta criar um arquivo chamado `jboss-web.xml` na pasta `WEB-INF` do projeto, referenciando o *security domain* que foi criado no JBoss (através do nome: `MyRealm`):

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>
  <security-domain>MyRealm</security-domain>
</jboss-web>
```

Note que o conteúdo da tag `security-domain` é o nome do domínio de segurança.

Feito isso, basta realizar o *deploy* da aplicação no servidor e navegar para a URL do serviço de usuários: <https://localhost:8443/<contexto-da-aplicação>/services/usuarios> e comprovar tanto a utilização do certificado quanto a solicitação de senha (caso você esteja fazendo o teste com a aplicação contida no GitHub, o usuário e senha são `admin`).

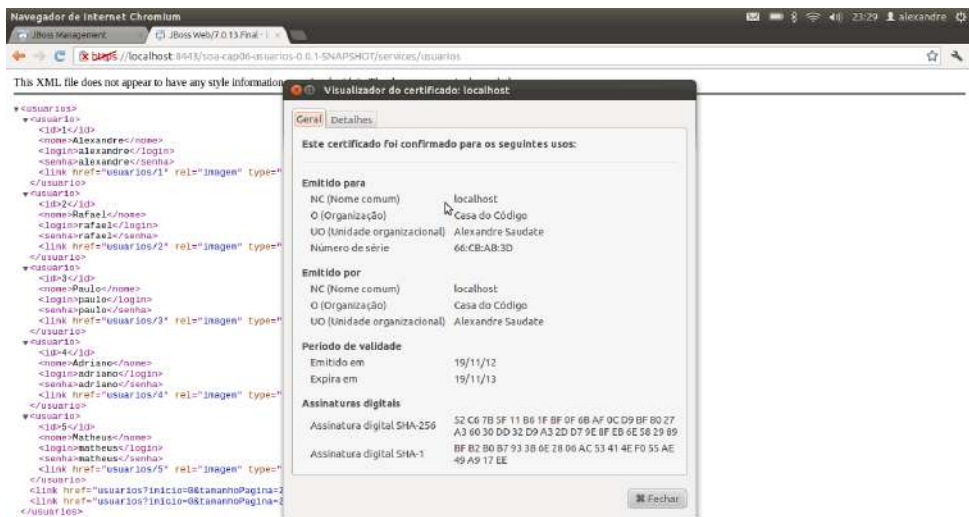


Figura 6.4: O serviço de fornecimento de usuários, já com certificado instalado

6.5 CONHECENDO WS-Security

Você aprendeu a criar e proteger serviços REST. Isso porque esse tipo de serviço funciona baseado no protocolo HTTP; mas e serviços baseados em SOAP? Estes não funcionam sempre baseados em HTTP; são livres de protocolo de transporte. Apesar de ser perfeitamente possível inserir autenticação e autorização HTTP em serviços baseados em WS-* e SOAP, este não é o melhor mecanismo para proteção

de serviços desse tipo.

Pensando nesse problema, a OASIS (*Organization for the Advancement of Structured Information Standards* — Organização para o Avanço de Padrões em Informações Estruturadas) desenvolveu o `WS-Security` (também abreviado como `WSS`), que é uma extensão para o já conhecido sistema de comunicação entre serviços baseados em contratos `WSDL`. Essa extensão contém vários mecanismos que permitem a realização de comunicações seguras mesmo quando o protocolo de transporte não é seguro, garantindo assim confiabilidade para as mensagens sob qualquer sistema de transporte utilizado.

SEGURANÇA DE INFORMAÇÕES E A NF-E

No Brasil, um sistema de geração de assinatura digital voltado a XML (*XML Signature*) tem sido muito utilizado em projetos de integração com o sistema de Nota Fiscal Eletrônica dos Governos Estaduais e Federal. Apesar de ser uma forma de proteger a integridade dos dados, não é um sistema completo de segurança, já que um atacante pode interceptar os dados e simplesmente refazer a assinatura. A NF-e conta também com a proteção oferecida pelo HTTPS, de maneira mútua, ou seja, o cliente deve apresentar seu certificado para o servidor antes de poder realizar a comunicação.

O `WS-Security` possui um sistema bem rico para prover segurança, tanto em termos de confidencialidade (como mencionado no *box* Segurança de informações e a NF-e), quanto em termos de autenticação/autorização. O sistema de autenticação funciona com diversos mecanismos (inclusive customizados), sendo os mais conhecidos: autenticação via *token* `SAML` (*Security Assertion Markup Language*), via *ticket* `Kerberos`, via fornecimento de usuário e senha (tanto com senha em texto puro quanto com hash) e via certificado `X.509`. Além disso, o `WS-Security` possui várias extensões (como `WS-Trust`, `WS-SecureConversation`, `WS-Federation` e outras) para o mecanismo como um todo, de maneira que estas extensões endereçam alguns dos problemas que a própria especificação `WS-Security` não faz.

O lado negativo do `WS-Security` para desenvolvedores Java é que o JAX-WS não oferece padronização para esta especificação. Isso faz com que o desenvolvimento de serviços seguros seja dependente do *framework* utilizado, ou até mesmo do

servidor utilizado (dependendo da capacidade deste de modificação do *framework* de serviços utilizados).

O `WS-Security` funciona em conjunto com as especificações `WS-Policy` e `WS-SecurityPolicy`. Estas duas especificações têm por objetivo, respectivamente, estabelecer políticas gerais a respeito de segurança, SLA, qualidade de serviço, confiabilidade etc.; e estabelecer, especificamente, quais são as políticas de segurança aplicáveis a um determinado serviço. A especificação da `WS-Policy` pode ser encontrada em <http://www.w3.org/TR/ws-policy-attach/>, e a da `WS-SecurityPolicy`, em <http://bit.ly/Y4oi4H>. Cada uma é razoavelmente grande e complicada — recomendo a leitura apenas em caso de necessidade.

Utilizando as definições de políticas

A `WS-Policy`, por si só, é razoavelmente fácil de ser utilizada (posto que se trata apenas de uma “cápsula” para outras especificações). Para definir uma política de segurança, basta incluir o elemento `Policy` em uma posição do seu WSDL (preferencialmente, após a definição de serviço, ou mesmo em um WSDL à parte) e definir um ID para essa política. Por exemplo, para definir uma política em um WSDL à parte, pode-se definir o documento todo assim:

```
<wsdl:definitions
  targetNamespace="http://servicos.estoque.knight.com/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-
200401-wss-wssecurity-utility-1.0.xsd">
  <wsp:Policy wsu:Id="DefaultSecureServicePolicy">
    <!-- conteúdo da política -->
  </wsp:Policy>
</wsdl:definitions>
```

Note que a definição desse ID é, na verdade, originária de um *XML Schema* de apoio, localizado em <http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd>.

Dentro do elemento `Policy`, existe um conjunto de elementos que pode ser utilizado, onde os mais importantes são `All` (todos) e `ExactlyOne` (exatamente um).

Estes elementos são utilizados para que seja possível criar combinações de políticas aplicáveis. Como os próprios nomes indicam, dos elementos encapsulados em

`ExactlyOne`, exatamente um deve ser atendido por uma requisição. Já dos elementos encapsulados em `All`, todas as políticas compreendidas devem ser atendidas.

Para que possamos definir um serviço seguro, várias questões devem ser consideradas: qual protocolo de transporte será utilizado? Em caso de interceptação da mensagem por um atacante, ele será capaz de interpretar os dados passados? Sendo ou não capaz de interpretar esses dados, ele será capaz de reenviar a requisição, passando-se pelo cliente legítimo do serviço? E quanto ao servidor, terá proteção suficiente para que uma senha não seja interceptada ao chegar no servidor? E quanto a um ambiente mais amplo, de vários serviços, como garantir que um elo fraco de uma comunicação feita entre vários serviços não acabe comprometendo a segurança de todos?

Vamos revisar a arquitetura das aplicações da Knight.com e entender o que será feito:

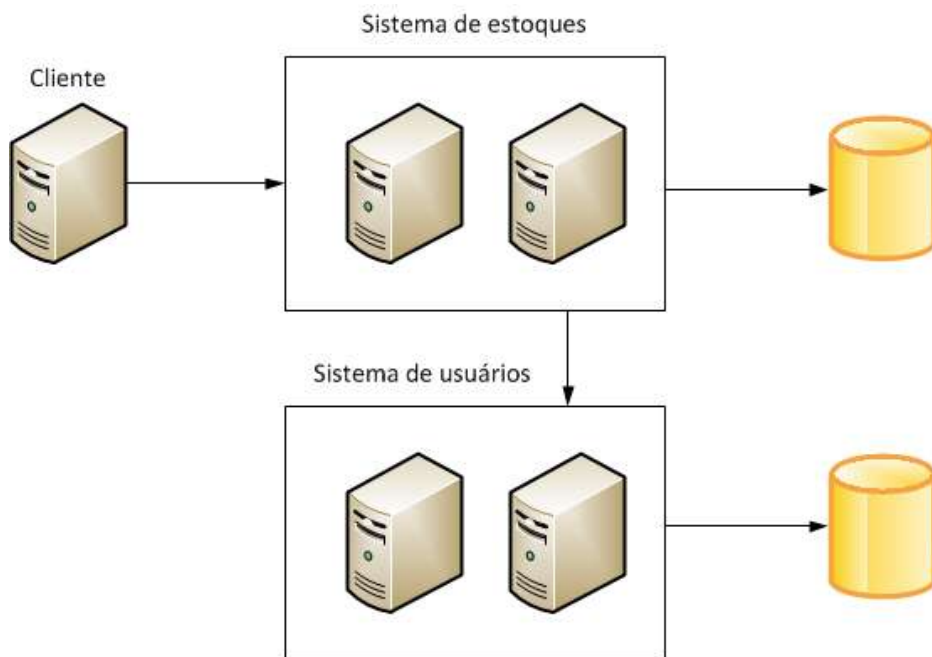


Figura 6.5: Arquitetura das aplicações knight.com

Como você pôde conferir na figura 6.5, nosso serviço de usuários deverá permanecer isolado do restante das aplicações, de maneira que seja possível alterar qual-

quer coisa no *back-end* do serviço de usuários (lembre-se de que, mantendo o contrato, detalhes da implementação do serviço são apenas detalhes).

Nessa arquitetura, portanto, vários desafios estarão presentes, pela natureza de comunicações entre os diversos serviços. Vamos começar respondendo a questões fundamentais, relativas apenas ao nosso serviço de estoque de livros. Depois, nos concentraremos no ecossistema das aplicações.

Nosso serviço utiliza HTTP como camada de transporte. Assim, o melhor mecanismo que podemos utilizar para realizar a proteção dessa camada é o SSL, transformando nosso serviço em HTTPS. Isso irá impedir o atacante de interpretar os dados trafegados.

Para impedir o atacante de reenviar esses dados, utilizaremos um mecanismo definido na *WS-Security*, chamado *Timestamp*. Este mecanismo define uma janela de tempo durante o qual a requisição será válida (sendo automaticamente invalidada se estiver fora dessa janela). Essa janela de tempo é informada pelo cliente para o servidor. Como esses dados serão trafegados de maneira encriptada (pelo mecanismo do HTTPS), um atacante não tem como modificá-los.

Quanto à proteção dos dados, nosso serviço estará comprometido, desde já, a não receber senhas em estado puro, mas apenas aquelas que passarem por mecanismo de *hashing*. Isso fará com que seja impossível reverter o processo, e o servidor deverá comparar apenas os *hashes* das senhas, e não as senhas em estado puro.

No caso do ambiente mais amplo, o ideal seria que utilizássemos *tokens* SAML ou Kerberos, de maneira que o serviço nunca tivesse acesso às senhas dos usuários, em formato algum. No entanto, para fins de simplicidade, utilizaremos passagem de usuário e senha (esta, tratada com *hash*), e eu mostrarei como utilizar criptografia de chave pública e privada para trafegar estes dados.

Aplique políticas de transporte

Para começar a utilizar as definições de políticas de segurança, precisamos “incrementar” nosso WSDL com o *namespace* dessas políticas. Este *namespace* é o <http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702>, e ele deixará o WSDL que vimos anteriormente assim:

```
<wsdl:definitions
  targetNamespace="http://servicos.estoque.knight.com/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-
```

```
200401-wss-wssecurity-utility-1.0.xsd"
  xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
    <wsp:Policy wsu:Id="DefaultSecureServicePolicy">
      <!-- conteúdo da política -->
    </wsp:Policy>
  </wsdl:definitions>
```

Feita esta modificação, incluiremos as definições em relação ao mecanismo de transporte. Como você já viu anteriormente, a melhor maneira de proteger nossos serviços é utilizando o HTTPS. No entanto, os clientes desse serviço precisam saber desse detalhe, então, vamos adicionar a definição de uso do HTTPS por parte do servidor utilizando as *tags* `TransportToken` e `HttpsToken`. Essas definições são instaladas dentro de uma definição mais genérica, relativa ao mecanismo de transporte, que é a *tag* `TransportBinding`. Dessa forma, a definição fica assim:

```
<!-- Definições de namespaces omitidas -->
<wsp:Policy wsu:Id="DefaultSecureServicePolicy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:TransportBinding>
        <wsp:Policy>
          <sp:TransportToken>
            <wsp:Policy>
              <sp:HttpsToken RequireClientCertificate="false" />
            </wsp:Policy>
          </sp:TransportToken>
        </wsp:Policy>
      </sp:TransportBinding>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

DEFINIÇÕES DE POLÍTICAS, A TAG `EXACTLYONE` E `ALL`

Você deve ter notado que, nas definições de políticas, a *tag* `ExactlyOne` encapsula a *tag* `All`. A verdade é que a razão para isso não está explícita na documentação — mas sabe-se que, se estas *tags* não tiverem esse formato, o JBossWS e o Apache CXF (mecanismos utilizados no JBoss AS 7) não conseguem detectar corretamente as configurações.

Portanto, ao incluir a *tag* `HttpsToken`, fica definido que o cliente deve utilizar o HTTPS como mecanismo de transporte. Lembre-se de que o HTTPS utiliza certificados para garantir a segurança. Neste caso, o atributo `RequireClientCertificate` serve para assinalar se é necessário ou não que o cliente envie o próprio certificado (neste caso, nosso WSDL demonstra que não é necessário).

O próximo passo é realizar a definição de um mecanismo que nos proteja contra *replay attacks*, ou seja, algo que impeça que um atacante intercepte a requisição e a reenvie para o servidor. Neste caso, podemos utilizar uma definição de um *timestamp*, ou seja, o período durante o qual a mensagem é válida. Para isso, utilizamos a *tag* `IncludeTimestamp`, diretamente na definição política de transporte. Esta *tag* irá forçar o cliente a incluir um cabeçalho (ou seja, conteúdo dentro da *tag* `Header`) contendo este período. A definição de políticas ficará desta forma:

```
<!-- Definições de namespaces omitidas -->
<wsp:Policy wsu:Id="DefaultSecureServicePolicy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:TransportBinding>
        <wsp:Policy>
          <sp:TransportToken>
            <wsp:Policy>
              <sp:HttpsToken RequireClientCertificate="false" />
            </wsp:Policy>
          </sp:TransportToken>
          <sp:IncludeTimestamp/>
        </wsp:Policy>
      </sp:TransportBinding>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Por último, incluímos uma definição de que ordem o mecanismo deverá efetuar o processamento dos cabeçalhos da mensagem. Incluímos este dado caso desejemos passar mais alguma informação no cabeçalho, e para não sermos limitados em relação à ordem. Para remover esta limitação, incluímos uma *tag* denominada `Layout`, contendo a *tag* `Lax`:

```
<!-- Definições de namespaces omitidas -->
<wsp:Policy wsu:Id="DefaultSecureServicePolicy">
```

```

<wsp:ExactlyOne>
  <wsp:All>
    <sp:TransportBinding>
      <wsp:Policy>
        <sp:TransportToken>
          <wsp:Policy>
            <sp:HttpsToken RequireClientCertificate="false" />
          </wsp:Policy>
        </sp:TransportToken>
        <sp:IncludeTimestamp/>
        <sp:Layout>
          <wsp:Policy>
            <sp:Lax />
          </wsp:Policy>
        </sp:Layout>
      </wsp:Policy>
    </sp:TransportBinding>
  </wsp:All>
</wsp:ExactlyOne>
</wsp:Policy>

```

Desta forma, nosso mecanismo de transporte está totalmente protegido. Resta, então, a definição em relação à camada de aplicação, ou seja, como realizaremos a proteção de nosso serviço forçando o cliente a passar usuário e senha.

Defina a política de autenticação

Políticas de autenticação por usuário e senha são denominadas, no jargão WS-*, de autenticação por usuário e *token*. Vários tipos de *token*, na verdade, são permitidos, mas o que nos interessa no momento é *HashPassword*. Este tipo de *token* é representado pela senha, que é passada por um algoritmo de *hash*. Este algoritmo segue a seguinte regra:

`Password_Digest = Base64 (SHA-1 (nonce + created + password))`

Ou seja, o *hash* da senha deverá ser feito através da concatenação do *nonce*, a data de criação e a senha, propriamente dita. Esses dados também serão passados para o serviço. O conjunto passará por algoritmo de *hash* SHA-1 e, depois, codificada em *Base64*. Este algoritmo irá realizar a proteção de nossa senha para que ela não seja passada em seu estado puro (seguindo os princípios da autenticação *Basic* do HTTPS, por exemplo).

Além disso, podemos realizar a definição quanto à nossa vontade de receber de volta o usuário e senha utilizados para a autenticação. Como não é necessário, realizamos o ajuste utilizando o atributo `IncludeToken`, na tag `UsernameToken`. Ao utilizar o valor <http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRecipient> na definição deste atributo, informamos ao mecanismo que não é necessário devolver o conjunto usuário/senha.

Para realizar estas definições, incluímos o seguinte na nossa política de segurança:

```
<wsp:Policy wsu:Id="DefaultSecureServicePolicy">
  <wsp:ExactlyOne>
    <wsp:All>
      <!-- Definições do mecanismo de transporte -->
      <sp:SignedSupportingTokens>
        <wsp:Policy>
          <sp:UsernameToken
            sp:IncludeToken="http://docs.oasis-open.org/
              ws-sx/ws-securitypolicy/200702/IncludeToken/
              AlwaysToRecipient">
            <wsp:Policy>
              <sp:HashPassword />
            </wsp:Policy>
          </sp:UsernameToken>
        </wsp:Policy>
      </sp:SignedSupportingTokens>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Assim, a definição completa da política de segurança fica:

```
<wsp:Policy wsu:Id="DefaultSecureServicePolicy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:TransportBinding>
        <wsp:Policy>
          <sp:TransportToken>
            <wsp:Policy>
              <sp:HttpsToken RequireClientCertificate="false"/>
            </wsp:Policy>
          </sp:TransportToken>
        </wsp:Policy>
      </sp:TransportBinding>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

```

        <sp:Layout>
            <wsp:Policy>
                <sp:Lax/>
            </wsp:Policy>
        </sp:Layout>
        <sp:IncludeTimestamp/>
    </wsp:Policy>
</sp:TransportBinding>
<sp:SignedSupportingTokens>
    <wsp:Policy>
        <sp:UsernameToken
            sp:IncludeToken="http://docs.oasis-open.org/ws-sx/
ws-securitypolicy/200702/IncludeToken/AlwaysToRecipient">
            <wsp:Policy>
                <sp:HashPassword/>
            </wsp:Policy>
        </sp:UsernameToken>
    </wsp:Policy>
</sp:SignedSupportingTokens>
</wsp:All>
</wsp:ExactlyOne>
</wsp:Policy>

```

Note o atributo `Id` na definição da política. Ele identifica o conjunto da política de segurança como um todo. Para aplicá-la, basta utilizar a *tag* `PolicyReference`. Esta *tag* deve ser utilizada para referenciar a política em questão — o que é feito através do atributo `URI`. Esta *tag* pode ser inserida tanto imediatamente dentro da *tag* `binding`, no WSDL, quanto na definição de operações. Isto irá diferenciar a aplicação das políticas, tanto para todas as operações definidas no `binding`, quanto para alguma operação específica. No nosso caso, para realizar a aplicação para todas as operações, fazemos o seguinte:

```

<wsdl:binding name="AutoresServiceServiceSoapBinding"
    type="tns:AutoresService">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsp:PolicyReference URI="#DefaultSecureServicePolicy"/>
    <wsdl:operation name="listarAutores">
        <soap:operation soapAction="" style="document"/>
        <wsdl:input name="listarAutores">
            <soap:body use="literal"/>
        </wsdl:input>
    </wsdl:operation>
</wsdl:binding>

```



```

    </wsdl:input>
    <wsdl:output name="listarAutoresResponse">
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>

```

Perceba a presença da *tag* `PolicyReference`, assim como o atributo `URI`. Este atributo deverá ter o valor do ID da política, precedido por `#`. Desta forma, o contrato está pronto. Salve tudo (incluindo as definições de políticas de segurança) na pasta `WEB-INF/wsdl`, com o nome de `AutoresService.wsdl`. Mais tarde, faremos a implementação do *web service* trabalhar com esta nova versão do contrato.

6.6 AJUSTES DE INFRA-ESTRUTURA

Alguns ajustes extras, além dos já realizados, são necessários. Você deve abrir novamente o arquivo `standalone.xml` e localizar o subsistema `urn:jboss:domain:webservices:1.1`. Feito isso, basta determinar quais são as portas que devem ser utilizadas para o servidor para servir tanto serviços sem proteção quanto com proteção. Isso é feito através das *tags* `wsdl-port` e `wsdl-secure-port`, respectivamente (ressaltando, novamente, que este procedimento pode ser feito apenas para máquinas de desenvolvimento. Caso o servidor em questão seja de produção, utilize o JBoss CLI). Dessa forma, o conteúdo do subsistema fica assim:

```

<subsystem xmlns="urn:jboss:domain:webservices:1.1">
  <modify-wsdl-address>true</modify-wsdl-address>
  <wsdl-host>${jboss.bind.address:127.0.0.1}</wsdl-host>
  <wsdl-port>8080</wsdl-port>
  <wsdl-secure-port>8443</wsdl-secure-port>
  <endpoint-config name="Standard-Endpoint-Config"/>
  <endpoint-config name="Recording-Endpoint-Config">
    <pre-handler-chain name="recording-handlers"
      protocol-bindings=##SOAP11_HTTP
        ##SOAP11_HTTP_MTOM ##SOAP12_HTTP ##SOAP12_HTTP_MTOM">
      <handler name="RecordingHandler"
        class="org.jboss.ws.common.invocation.RecordingServerHandler"/>
    </pre-handler-chain>
  </endpoint-config>
</subsystem>

```

Em seguida, é necessário configurar a própria aplicação para interceptar as requisições aos serviços, de maneira que a *engine* de segurança faça a validação dos dados. A validação dos dados de usuário e senha é feita de maneira programática, através de um *callback* instalado especialmente para este propósito. Ele é definido através de uma interface chamada `javax.security.auth.callback.CallbackHandler`. Por exemplo, para definirmos este *callback*, definimos a seguinte classe:

```
package com.knight.estoque.servicos.seguranca

import javax.security.auth.callback.*;

public class CallbackSeguranca implements CallbackHandler {
    public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {

    }
}
```

A definição de uso deste *callback* será colocada num arquivo chamado `jaxws-endpoint-config.xml`. Este arquivo deverá definir uma propriedade especial, chamada `ws-security.callback-handler`. Ela dirá ao Apache CXF qual classe deverá ser usada para realizar o *callback*. O conteúdo do arquivo será o seguinte:

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:javaee="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="urn:jboss:jbossws-
jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
  <endpoint-config>
    <config-name>Endpoint WS-Security</config-name>
    <property>
      <property-name>ws-security.callback-handler</property-name>
      <property-value>
        com.knight.estoque.servicos.seguranca.CallbackSeguranca
      </property-value>
    </property>
  </endpoint-config>
</jaxws-config>
```

Mas nossa configuração ainda não está finalizada. A última coisa que devemos fazer é editar o *web service* para realizar o seguinte: encontrar o contrato que definimos manualmente, encontrar a definição do *callback* e adaptá-lo para ser seguro. Estas definições são dadas pelas anotações `@WebService`, `@EndpointConfig` e `@WebContext`, respectivamente.

Assumindo que o WSDL editado foi salvo na pasta `WEB-INF/wsdl`, com o nome `AutoresService.wsdl`, a nova definição da implementação é feita da seguinte forma:

```
package com.knight.estoque.servicos;

// imports omitidos
@WebService(portName = "AutoresServicePort",
            serviceName = "AutoresServiceService",
            targetNamespace = "http://servicos.estoque.knight.com/",
            wsdlLocation = "WEB-INF/wsdl/AutoresService.wsdl")
@Stateless
public class AutoresService {

    public List<Autor> listarAutores() {
        // Implementação omitida
    }

}
```

Note que as definições dos atributos da anotação `@WebService` foram feitas com base no WSDL: `portName` é diretamente equivalente à definição de `port` (localizada dentro da *tag* `service`); `serviceName` é diretamente equivalente à definição encontrada na *tag* `service`; `targetNamespace` é a própria definição de *namespace* do serviço, encontrada na *tag* `definitions`; e `wsdlLocation` é referente à localização do contrato já editado.

O próximo passo é configurar este serviço para trabalhar com o *callback* implementado anteriormente. Para isso, utilizamos a anotação `org.jboss.ws.api.annotation.EndpointConfig`, onde podemos definir o nome do arquivo de configuração a ser utilizado para rastrear o *callback* e o nome da definição, dentro deste arquivo, a ser utilizada. No nosso caso, devemos referenciar o arquivo `WEB-INF/jaxws-endpoint-config.xml`, e a definição `Endpoint WS-Security`. A implementação do serviço fica assim:

```
package com.knight.estoque.servicos;

// imports omitidos
@WebService(portName = "AutoresServicePort",
            serviceName = "AutoresServiceService",
            targetNamespace = "http://servicos.estoque.knight.com/",
            wsdlLocation = "WEB-INF/wsdl/AutoresService.wsdl")
@EndpointConfig(configFile = "WEB-INF/jaxws-endpoint-config.xml",
                configName = "Endpoint WS-Security")
@Stateless
public class AutoresService {

    public List<Autor> listarAutores() {
        // Implementação omitida
    }

}
```

E, finalmente, devemos dizer ao JBoss que este serviço deve ser seguro através de HTTPS (além de arrumar a URL do serviço, posto que a *engine* do JBoss AS modificou). Isso é feito através da anotação `org.jboss.ws.api.annotation.WebContext`, da seguinte forma:

```
package com.knight.estoque.servicos;

// imports omitidos
@WebService(portName = "AutoresServicePort",
            serviceName = "AutoresServiceService",
            targetNamespace = "http://servicos.estoque.knight.com/",
            wsdlLocation = "WEB-INF/wsdl/AutoresService.wsdl")
@EndpointConfig(configFile = "WEB-INF/jaxws-endpoint-config.xml",
                configName = "Endpoint WS-Security")
@WebContext(secureWSDLAccess = true,
            transportGuarantee = "CONFIDENTIAL",
            urlPattern = "AutoresService")
@Stateless
public class AutoresService {

    public List<Autor> listarAutores() {
        // Implementação omitida
    }

}
```

```
}
```

Desta forma, o serviço está finalmente pronto para trabalhar com segurança.

6.7 O CALLBACK DE VERIFICAÇÃO DA SENHA

Conforme definido no arquivo `jaxws-endpoint-config.xml` (visto em 6.6, você deve agora criar uma classe que fará a verificação da senha passada pelo cliente. Esta classe deverá se chamar `com.knight.estoque.servicos.seguranca.CallbackSeguranca` e, de acordo com a *engine* do Apache CXF, deve implementar a interface `javax.security.auth.callback.CallbackHandler`. O código inicial fica assim:

```
package com.knight.estoque.servicos.seguranca;

import java.io.IOException;
import javax.security.auth.callback.*;

public class CallbackSeguranca implements CallbackHandler {

    public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {

    }
}
```

O método `handle` será invocado a cada chamada protegida dos serviços. Nesta invocação, um *array* de *callbacks* é passado como parâmetro. Cada *callback* tem uma responsabilidade diferente, de acordo com a verificação de segurança desejada (lembre-se de que é possível inserir várias verificações diferentes). Para realizar a verificação da senha, o Apache CXF fornece uma implementação de *Callback* através da classe `org.apache.ws.security.WSPasswordCallback`. Para utilizá-la corretamente, deve-se, portanto, fazer *cast* de um *callback*, desta forma:

```
public void handle(Callback[] callbacks) throws IOException,
    UnsupportedCallbackException {
    for (int i = 0; i < callbacks.length; i++) {
        if (callbacks[i] instanceof WSPasswordCallback) {
```

```

        WSPasswordCallback pc = (WSPasswordCallback) callbacks[i];
    }
}

```

Ao utilizar esta classe, deve-se recuperar o usuário que fez a requisição através do método `getIdentifier`. Este método retorna uma *string* contendo o nome do usuário. A partir dessa informação, deve-se recuperar a senha do usuário e informá-la ao *callback*, de maneira que, por motivos de segurança, o programador não tem acesso à senha informada. Desta forma, o código pode ser implementado como:

```

public void handle(Callback[] callbacks) throws IOException,
    UnsupportedCallbackException {
    for (int i = 0; i < callbacks.length; i++) {
        if (callbacks[i] instanceof WSPasswordCallback) {
            WSPasswordCallback pc = (WSPasswordCallback) callbacks[i];
            Usuario usuario = encontreUsuario(pc.getIdentifier());
            if (usuario == null)
                return;
            pc.setPassword(usuario.getSenhaDecodificada());
        }
    }
}

```

Como estamos falando de SOA e temos duas aplicações distintas (de usuários e de gerenciamento de estoque), o correto a se fazer é buscar o usuário no sistema de gerenciamento de usuários, através dos serviços REST expostos. No entanto, precisamos realizar algumas alterações nele, para que isso funcione.

6.8 A ATUALIZAÇÃO DO SERVIÇO DE USUÁRIOS

A primeira modificação a ser realizada no nosso serviço é incluir uma operação que recupere o usuário corretamente. Como você viu no capítulo 5, isso deve ser realizado a partir de um método `GET`, onde o *login* do usuário pode ser passado como parte da URL. Além disso, também vamos realizar o mapeamento necessário para que possamos nos beneficiar do *cache* de usuários, com o *header* `If-Modified-Since`. A versão inicial do nosso método fica assim:

```

@GET
@Path("/{login}")

```

```
public Response find(@PathParam("login") String login,  
    @HeaderParam("If-Modified-Since") Date modifiedSince);
```

Entretanto, existe o problema da senha. Note que, quando realizamos a criação do usuário anteriormente, salvamos as senhas em sua forma pura (o que é errado, do ponto de vista da segurança — procure sempre salvar as senhas com alguma forma de criptografia ou, ainda melhor, aplique sistemas de *hashes* seguros). Como estamos usando SSL, não necessariamente precisamos modificar o tráfego da senha; porém, eu gostaria de apresentar um mecanismo de criptografia para ilustrar alguns conceitos.

Para encriptar as senhas que serão trafegadas, nossa aplicação de gerenciamento de usuários deve aceitar um mecanismo de chave pública e privada (gerenciado manualmente) para que seja possível obter tanto flexibilidade no tratamento dessas senhas quanto segurança.

XML ENCRYPTION E JAVA

Existe um padrão (definido pela W3C — *World Wide Web Consortium*) para tratamento de criptografia em XML, que pode ser encontrado em <http://www.w3.org/TR/xmlenc-core/>. Esse padrão contém um mecanismo para tratamento de comunicações baseadas em RSA (um mecanismo de criptografia de chave pública/privada); contudo, realizar o uso desse mecanismo em Java é demasiadamente complicado, e este autor não localizou nenhum mecanismo que fizesse esse tratamento de uma maneira satisfatória para a linguagem Java.

Uma chave pública RSA tem basicamente duas partes, *modulus* e *exponent*. Dessa forma, podemos modelar a transmissão dessa chave pública utilizando o seguinte mapeamento no JAXB:

```
package com.knight.usuarios.servicos.seguranca;  
  
// imports omitidos  
  
@XmlAccessorType(XmlAccessType.FIELD)  
@XmlRootElement(name = "RSA")  
public class RSAPublica {
```

```
private BigInteger modulus;  
  
private BigInteger publicExponent;  
}
```

Observe que ambos os atributos são modelados utilizando `BigInteger`. Isso porque a especificação RSA declara que ambos os valores são inteiros muito grandes (especialmente *modulus*). Assim, `BigInteger` é a melhor classe para realizar essa modelagem.

Nosso próximo passo é alterar a assinatura do método para que seja possível receber essa mensagem. Porém, note que existem sérios problemas: o primeiro, os números que são utilizados na chave RSA são muito grandes, e não fica esteticamente agradável utilizá-los em uma requisição do tipo `GET`. Outro problema é que não é aconselhável trafegar dados sensíveis pelo método `GET`, como senhas ou chaves criptográficas, já que as informações fornecidas por `GET` trafegam como *query strings*, que os *browsers* armazenam em histórico — possibilitando a um usuário não autorizado a visualização destes dados. Assim, surge um conflito: estamos realizando uma busca de dados, mas não podemos utilizar `GET`.

Obviamente, existem maneiras de tratar este problema (transmitindo os dados da chave RSA como *headers* HTTP, por exemplo). No entanto, este cenário exhibe algo que pode acontecer com mais ou menos frequência em REST: nem sempre é possível atender todas as especificações de maneira eficiente. Por exemplo, uma busca complexa de informações (como uma busca por exemplos) poderia ser prejudicada por esse modelo `GET`. Desta forma, realizo aqui uma quebra de protocolo: altero o método HTTP para `POST`.

BUSCA DE INFORMAÇÕES UTILIZANDO POST

Não é a primeira vez que alguém faz esta modificação em uma API. De fato, a própria API do Twitter, para mencionar um exemplo, tem algo semelhante, visto em <https://dev.twitter.com/docs/api/1.1/post/statuses/filter>. Neste documento, é detalhado que se trata de uma busca de *status* por filtros — todavia, os filtros podem ser muito grandes e serem rejeitados por comprimento de URL.

Particularmente, eu acredito que a complexidade de buscas é ainda mais grave do que o comprimento da URL, pelo fato de que *query strings* são, naturalmente, desprovidas de elementos-pai ou qualquer coisa que sugira uma estrutura hierárquica.

Sendo assim, alteramos o método para ter o seguinte formato:

```
@POST
@Path("/{login}")
public Response find(@PathParam("login") String login,
    @HeaderParam("If-Modified-Since") Date modifiedSince,
    RSAPublicKey chaveCriptografica);
```

Tendo inserido essa definição do método na interface, resta implementar a busca, propriamente dita. Antes disso, porém, vou inserir um método utilitário na classe que representa a chave pública, chamado `encripta`. Esse método receberá como parâmetro um *array* de *bytes*, que conterá o dado a ser criptografado, e retornará uma *String* — ou seja, a senha já criptografada e codificada com o algoritmo `Base64`. Para isso, devo utilizar uma API já contida na *Virtual Machine*, chamada JCE — *Java Cryptography Extension*. Essa API já contém os mecanismos necessários para trabalharmos com criptografia em Java, de maneira que podemos aproveitá-la na nossa classe.

Para nos beneficiarmos dessa facilidade, utilizamos inicialmente a classe `java.security.spec.RSAPublicKeySpec`. Esta classe representa a especificação de uma chave pública RSA perante a *engine* do JCE. Por ser apenas uma especificação, precisamos materializar a chave, utilizando a classe `java.security.KeyFactory`, que contém um método chamado `generatePublic`, que irá criar uma instância condizente com a interface `java.security.PublicKey`.

A partir de então, basta alimentar o mecanismo de criptografia do JCE, representado pela classe `javax.crypto.Cipher`. Esta classe contém, principalmente, os métodos `init` (que irá realizar a alimentação do mecanismo de criptografia) e `doFinal` (que irá realizar o próprio procedimento de criptografia). Ao final, utilizamos a classe `org.apache.commons.codec.binary.Base64` (do Apache Commons Codec) para realizar a codificação do resultado em `Base64`.

```
package com.knight.usuarios.servicos.seguranca;

// imports omitidos

@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "RSA")
public class RSAPublica {

    private BigInteger modulus;

    private BigInteger publicExponent;

    public String encripta(byte[] bytes) throws ExcecaoCriptografia {
        try {
            PublicKey publicKey = criaChave();
            Cipher cipher = Cipher.getInstance("RSA");
            cipher.init(Cipher.ENCRYPT_MODE, publicKey);
            return Base64.encodeBase64String(cipher.doFinal(bytes));
        } catch (Exception e) {
            throw new ExcecaoCriptografia(e);
        }
    }

    protected PublicKey criaChave() throws InvalidKeySpecException,
        NoSuchAlgorithmException {
        RSAPublicKeySpec publicKeySpec = new RSAPublicKeySpec(modulus,
            publicExponent);
        return KeyFactory.getInstance("RSA").generatePublic(publicKeySpec);
    }
}
```

Note que o método `encripta` lança uma exceção, `ExcecaoCriptografia`. Ela foi criada para encapsular as exceções particulares do JCE (que são muitas) em uma única, facilitando o uso da API. Sua definição é a seguinte:

```
package com.knight.usuarios.servicos.seguranca;

public class ExcecaoCriptografia extends Exception {

    // Construtores herdados de Exception

}
```

6.9 A IMPLEMENTAÇÃO DO MÉTODO DE BUSCA

A seguir, realizamos a implementação do método de busca na classe `UsuariosService`. Para facilitar a busca, criamos uma *Named Query* na classe `Usuario`, de maneira a concentrar num único ponto esta busca específica. Trata-se de uma *query* simples, que filtra os usuários pelo *login*, como visto na listagem:

```
@NamedQueries(
    @NamedQuery(
        name = "usuario.encontrar.login",
        query = "select u from Usuario u where u.login = ?")
)
public class Usuario extends EntidadeModelo implements RESTEntity
```

A partir desse código, utilizamos o *Entity Manager* para realizar a busca usando a *named query* criada:

```
public Response find(String login, Date modifiedSince,
    RSAPublica chaveCriptografica) {
    Usuario usuario;
    try {
        usuario = em
            .createNamedQuery("usuario.encontrar.login", Usuario.class)
            .setParameter(1, login).getSingleResult();
    } catch (NoResultException e) {
        return Response.status(Status.NOT_FOUND).build();
    }

    // O restante do código...
}
```

Uma vez localizado o usuário, é importante se lembrar de utilizar o método `detach`, contido no *entity manager*, para “desligar” o usuário do contexto transa-

cional. Isso é porque, quando realizarmos a criptografia da senha, o próprio usuário recuperado será utilizado para devolver o resultado da consulta. Porém, se fizermos essa atualização da senha num usuário ainda conectado ao banco, as alterações que fizermos serão automaticamente persistidas no banco — o que fará com que requisições subsequentes falhem.

Realizado o procedimento, basta ajustar a senha do usuário como a senha já criptografada (através do método `encripta` da classe `RSAPublica`) e realizar os procedimentos já conhecidos para tratamento de *caches*. O código completo, portanto, é o seguinte:

```
public Response find(String login, Date modifiedSince,
    RSAPublica chaveCriptografica) {
    Usuario usuario;
    try {
        usuario = em
            .createNamedQuery("usuario.encontrar.login", Usuario.class)
            .setParameter(1, login).getSingleResult();
    } catch (NoResultException e) {
        return Response.status(Status.NOT_FOUND).build();
    }

    em.detach(usuario);

    if (modifiedSince != null) {
        if (usuario.getDataAtualizacao().after(modifiedSince)) {
            criptografarSenhaUsuario(usuario, chaveCriptografica);
            return Response.ok(usuario).build();
        }
        return Response.notModified().build();
    } else {
        criptografarSenhaUsuario(usuario, chaveCriptografica);
        return Response.ok(usuario).build();
    }
}

private void criptografarSenhaUsuario(Usuario usuario,
    RSAPublica chave) {
    try {
        usuario.setSenha(chave.encripta(usuario.getSenha())
```

```
        .getBytes()));  
    } catch (Exception e) {  
        throw new  
            WebApplicationException(Status.INTERNAL_SERVER_ERROR, e);  
    }  
}
```

6.10 REALIZE A COMUNICAÇÃO ENTRE OS DOIS SISTEMAS

Voltando ao sistema de estoques, devemos implementar na classe `CallbackSeguranca` o método `encontreUsuario`, que deverá realizar a comunicação com o serviço de usuários, em REST.

Para fazer isso, no entanto, ele deve ser capaz de prover uma chave pública para o serviço de usuários e ainda manter uma chave privada. Para realizar a geração destas chaves, podemos novamente recorrer ao JCE e à classe `java.security.KeyPairGenerator`. É possível informar a esta classe qual algoritmo será utilizado para a geração do par de chaves e, ainda, qual o tamanho. Para gerar essas chaves (assumindo 2048 bits de comprimento) e salvá-las em um arquivo, por exemplo, temos o seguinte código:

```
import java.io.*;  
import java.security.*;  
public class GeradorChaves {  
  
    public static void main(String[] args) throws Exception {  
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");  
        kpg.initialize(2048);  
  
        KeyPair kp = kpg.generateKeyPair();  
        Key publicKey = kp.getPublic();  
        Key privateKey = kp.getPrivate();  
  
        ObjectOutputStream oos = new ObjectOutputStream(  
            new FileOutputStream("public.key"));  
        oos.writeObject(publicKey);  
        oos.flush();  
        oos.close();  
  
        oos = new ObjectOutputStream(new FileOutputStream("private.key"));  
        oos.writeObject(privateKey);  
    }  
}
```

```

        oos.flush();
        oos.close();
    }
}

```

Uma vez gerados os arquivos `public.key` e `private.key`, temos as nossas chaves RSA. O próximo passo é construir um código que seja adequado para o consumo do serviço REST (lembre-se de que a classe deve ser compatível com aquilo que é esperado pelo serviço de usuários). Para facilitar o processo de carregar a chave em memória, construímos um método para carregar estaticamente a classe e também referenciá-la no *callback*. O código de carga da classe é o seguinte (assuma que o arquivo `public.key` está na raiz do diretório de fontes):

```

package com.knight.estoque.servicos.seguranca;

// imports omitidos

@XmlRootElement(name = "RSA")
@XmlAccessorType(XmlAccessType.FIELD)
public class ChaveRSA {

    private BigInteger modulus;

    private BigInteger publicExponent;

    public static ChaveRSA carregar() throws IOException,
        ClassNotFoundException {

        try (InputStream inputStream = ChaveRSA.class
            .getResourceAsStream("/public.key")) {
            ObjectInputStream ois = new ObjectInputStream(inputStream);
            RSAPublicKey rsaPublicKey = (RSAPublicKey) ois.readObject();
            ChaveRSA chaveRSA = new ChaveRSA();
            chaveRSA.modulus = rsaPublicKey.getModulus();
            chaveRSA.publicExponent = rsaPublicKey.getPublicExponent();
            return chaveRSA;
        }
    }
}

```

Esta classe será utilizada para trafegar os dados até o serviço corretamente. Na sequência, basta inserir o código para realizar a carga desta chave no *callback*:

```
public class CallbackSeguranca implements CallbackHandler {

    private static ChaveRSA chaveRSA;

    static {
        try {
            chaveRSA = ChaveRSA.carregar();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    //Restante da implementação...
}
```

O próximo passo é escrever a classe que irá receber os dados do serviço REST, ou seja, o usuário:

```
package com.knight.estoque.servicos.seguranca;

// imports omitidos

@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement
class Usuario {

    private String nome;
    private String login;
    private String senha;
    private Date dataAtualizacao;
}
```

Note a presença do campo `dataAtualizacao`. Este campo não é fornecido como parte da entidade, propriamente dita. No entanto, existe um *header* padronizado pelo HTTP chamado `Date`. Nós vamos preencher o campo `dataAtualizacao` com o conteúdo do *header* `Date`, de maneira que tenhamos um mecanismo para manter um *cache* de usuários.

Além disso, observe que a senha virá de forma encriptada. Podemos incluir um mecanismo, na própria classe `Usuario`, capaz de decriptar a senha. Novamente,

recorremos ao JCE, de maneira que tudo o que temos que fazer é carregar o arquivo contendo a chave privada e inicializar uma instância de `Cipher`, que ficará pronta para decriptar as senhas. Para fazer isso, basta utilizar o seguinte código:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement
class Usuario {

    private String nome;
    private String login;
    private String senha;
    private Date dataAtualizacao;

    private static Cipher cipher;

    static {
        try {
            InputStream keyStream = Thread.currentThread()
                .getContextClassLoader().getResourceAsStream("private.key");
            ObjectInputStream ois = new ObjectInputStream(keyStream);
            Key decodeKey = (Key) ois.readObject();
            ois.close();
            cipher = Cipher.getInstance("RSA");
            cipher.init(Cipher.DECRYPT_MODE, decodeKey);

        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    // getters

    public String getSenhaDecodificada() {
        try {
            return new String(cipher.doFinal(Base64.decodeBase64(senha.
                getBytes())));
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```


Finalmente, temos que escrever o cliente do serviço REST. Como isso já foi exibido no capítulo 5, não vou entrar em grandes detalhes da implementação. Basta que você se atente a alguns detalhes; o primeiro é a autenticação. Como ela é do tipo Basic, basta inserir o *header* `Authorization` na requisição com o valor `Basic` e o usuário e senha, separados por dois-pontos e codificados em `Base64`. Ou seja, podemos preparar a requisição da seguinte forma:

```
private static String ENDERECO_SERVICO_USUARIOS =
    "https://localhost:8443/soa-cap06-usuarios-0.0.1-SNAPSHOT/services/";

private static String USUARIO = "admin";

private static String SENHA = "admin";

private Usuario encontreUsuario(String login) throws IOException {
    Usuario usuario = null;
    try {
        ClientRequest request = new ClientRequest(ENDERECO_SERVICO_USUARIOS
            + "usuarios/{login}")
            .pathParameters(login)
            .header("Authorization", getAuth())
            .body(MediaType.APPLICATION_XML, chaveRSA)
            .accept(MediaType.APPLICATION_XML);

        // restante do método encontreUsuario
    }

    private String getAuth() {
        return "Basic "
            + Base64.encodeBytes((USUARIO + ":" + SENHA).getBytes());
    }
}
```

Saiba que o usuário e senha não deveriam ficar carregados em memória de maneira pura; eles estão colocados desta forma apenas para simplificar o código.

Na sequência da preparação do código, checamos o *cache* de usuários para saber se temos como incluir o *header* `If-Modified-Since` ou não. O nosso *cache* pode ser implementado com um mapa; no entanto, você não deve confiar no servidor para manter sempre a mesma instância desta classe. Assim, ele foi incluído de maneira estática na classe `CallbackSeguranca`. Como isso pode originar problemas de concorrência, utilizamos um `java.util.concurrent.ConcurrentHashMap`

para lidar melhor com essa questão. Assim, todo o código para tratamento de *cache* fica desta forma:

```
private static Map<String, Usuario> cache = new ConcurrentHashMap<>();

private Usuario encontreUsuario(String login) throws IOException {
    Usuario usuario = null;
    try {
        ClientRequest request = new ClientRequest(ENDERECO_SERVICO_USUARIOS
            + "usuarios/{login}")
            .pathParameters(login)
            .header("Authorization", getAuth())
            .body(MediaType.APPLICATION_XML, chaveRSA)
            .accept(MediaType.APPLICATION_XML);

        if (cache.containsKey(login)) {
            usuario = cache.get(login);
            request.header("If-Modified-Since",
                org.jboss.resteasy.util.DateUtil.
                    formatDate(usuario.getDataAtualizacao()));
        }

        ClientResponse<Usuario> response = request.post();
        if (response.getStatus() == Status.NOT_MODIFIED.getStatusCode()) {
            return usuario;
        }
        // restante do método encontreUsuario
    }
}
```

Resta apenas fazer o tratamento de dados em caso de o serviço informar que os dados não estão no *cache* ou do usuário não ter sido localizado. O código completo de busca de usuários fica assim:

```
private Usuario encontreUsuario(String login) throws IOException {
    Usuario usuario = null;
    try {

        ClientRequest request = new ClientRequest(ENDERECO_SERVICO_USUARIOS
            + "usuarios/{login}").pathParameters(login)
            .header("Authorization", getAuth())
            .body(MediaType.APPLICATION_XML, chaveRSA)
            .accept(MediaType.APPLICATION_XML);
```

```
if (cache.containsKey(login)) {
    usuario = cache.get(login);
    request.header("If-Modified-Since",
        org.jboss.resteasy.util.DateUtil.
            formatDate(usuario.getDataAtualizacao()));
}

ClientResponse<Usuario> response = request.post();
if (response.getStatus() == Status.NOT_MODIFIED.getStatusCode()) {
    return usuario;
}

if (response.getStatus() == Status.OK.getStatusCode()) {
    usuario = response.getEntity(Usuario.class);

    Date date = org.jboss.resteasy.util.DateUtil.
        parseDate(response.getHeaders().getFirst(
            "Date"));
    usuario.setDataAtualizacao(date);
    cache.put(login, usuario);
    return usuario;
}

if (response.getStatus() == Status.NOT_FOUND.getStatusCode()) {
    return null;
}

throw new Exception("Usuário não localizado");
} catch (Exception e) {
    throw new IOException(
        "Não foi possível recuperar as informações do usuário");
}
}
```

Finalmente, o código do nosso *callback* de usuários está pronto. Caso seja necessário acessar dados de segurança no serviço, podemos utilizar o mecanismo de injeção do *Application Server*. Para isso, basta injetar uma instância de `javax.xml.ws.WebServiceContext`, utilizando a anotação `javax.annotation.Resource`:

```
import org.apache.ws.security.WSUsernameTokenPrincipal;
//outros imports omitidos

public class AutoresService {

    @Resource
    private WebServiceContext context;

    @PersistenceContext
    private EntityManager em;

    public List<Autor> listarAutores() {
        WSUsernameTokenPrincipal principal =
            (WSUsernameTokenPrincipal) context.getUserPrincipal();

        System.out.println("O usuário " + principal.getName()
            + " está listando autores");

        return em.createQuery("select a from Autor a", Autor.class)
            .getResultList();
    }
}
```

O código do lado do servidor está, então, completo. Resta agora realizar os testes e escrever o cliente para nosso serviço.

6.11 TESTES COM SOAPUI

Finalmente, podemos realizar os testes a respeito do funcionamento da segurança da aplicação. Se você acessar o console do JBoss, verá uma seção chamada *Webservices*. Essa seção contém quais *web services* estão em operação no sistema. Se você clicar sobre `AutoresService`, verá a URL do WSDL já modificada:

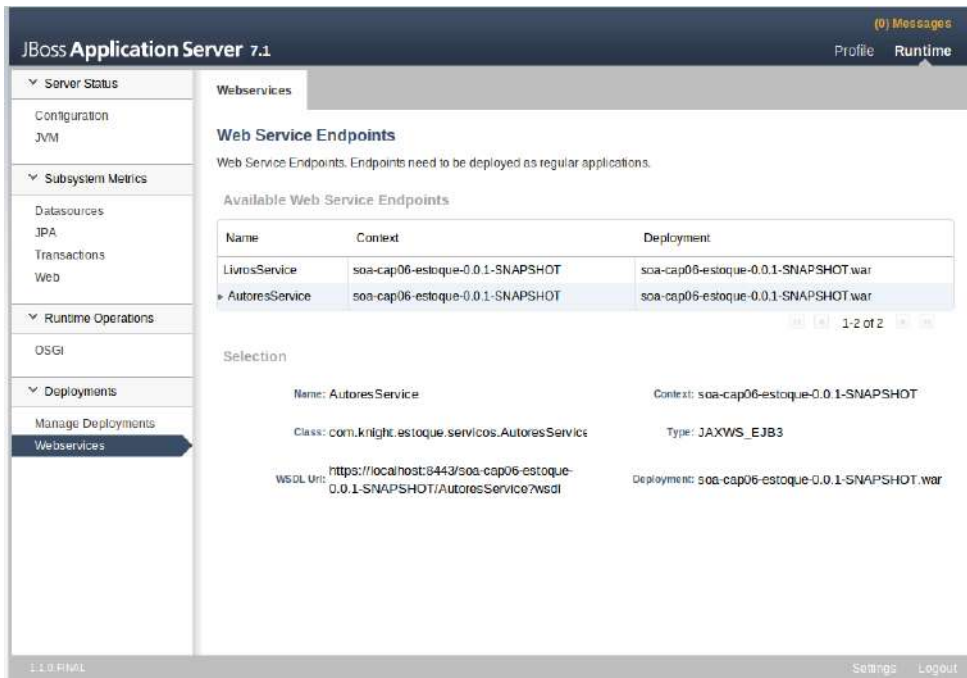


Figura 6.6: console do JBoss mostrando o endereço alterado do WSDL

De posse desse endereço, é possível acessá-lo pelo *browser* para checar se o contrato está corretamente exposto.

Uma vez checado, podemos prosseguir rumo aos testes com SoapUI. A criação do projeto, inicialmente, é igual à que fizemos anteriormente, ou seja, basta clicar com o botão direito do mouse sobre **Projects**, selecionar **New SoapUI Project** e, então, preencher o campo **Initial WSDL/WADL** com o endereço do WSDL. Feito isso, a estrutura inicial de projetos do SoapUI deve aparecer:



Figura 6.7: estrutura do projeto no SoapUI

Ao selecionar `Request 1`, uma caixa chamada `Request properties` deve aparecer pouco abaixo. Nesta caixa, existem duas seções, `Username` e `Password`. Estas duas caixas devem conter o usuário e senha, respectivamente, de um usuário existente no serviço de usuários.

Request Properties	
Property	Value
Name	Request 1
Description	
Message Size	229
Encoding	UTF-8
Endpoint	https://localhost:8...
Timeout	
Bind Address	
Follow Redirects	true
Username	alexandre
Password	*****
Domain	
Authentication Type	Global HTTP Settin...
WSS-Password Type	
WSS TimeToLive	
SSL Keystore	
Skip SOAP Action	false
Enable MTOM	false
Force MTOM	false
Inline Response A...	false

Figura 6.8: Caixa com entrada para usuário e senha

Ao realizar duplo clique sobre Request 1, a tela com a requisição “crua” deve aparecer. De acordo com o contrato, deve-se preencher tanto o usuário e a senha da requisição, quanto inserir um *timestamp*. Para preencher primeiro o usuário, basta clicar com o botão direito sobre a requisição e selecionar a opção Add WSS Username Token. Isso deve abrir uma caixa com o título Specify Password Type.

Nosso contrato especifica que deseja receber o *digest* da senha, então, selecione a opção PasswordDigest e clique em OK. Isso deverá alimentar a requisição para que a mesma contenha a estrutura necessária para passagem de usuário e senha.

Para inserir o *timestamp*, basta clicar com o botão direito sobre a requisição e clicar em Add WS-Timestamp. Ao fazer isso, o SoapUI abrirá uma caixa de diálogo com o título Specify Time-To-Live Value, ou seja, você deve fornecer como entrada o intervalo de validade do *timestamp*, em segundos. Deixe 60 e clique em OK.

Note que, caso você queira fazer outras requisições, deverá apagar o trecho que contém o *timestamp* e realizar esse procedimento novamente (já que a requisição tem prazo de validade).

Finalmente, basta clicar no botão com a seta verde (ao lado do endereço do serviço) para submeter a requisição. Se estiver tudo correto, a listagem de autores deve aparecer:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <wsse:Security soap:mustUnderstand="1" xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0"
      <wsu:Timestamp wsu:Id="TS-3">
        <wsu:Created>2012-12-09T20:58:20.649Z</wsu:Created>
        <wsu:Expires>2012-12-09T21:09:20.649Z</wsu:Expires>
      </wsu:Timestamp>
    </wsse:Security>
  </soap:Header>
  <soap:Body>
    <ns2:ListarAutoresResponse xmlns:ns2="http://servicos.estoque.knight.com/">
      <autor>
        <id>1</id>
        <nome>Paulo Silveira</nome>
      </autor>
      <autor>
        <id>2</id>
        <nome>Adriano Almeida</nome>
      </autor>
      <autor>
        <id>3</id>
        <nome>Vinicius Baggio Fuentes</nome>
      </autor>
    </ns2:ListarAutoresResponse>
  </soap:Body>
</soap:Envelope>
```

Figura 6.9: Resposta do serviço

6.12 CRIE O CLIENTE SEGURO

Para criar o cliente seguro do serviço, basta seguir os mesmos passos vistos no capítulo 1, com o comando `wsimport`. O que será diferente é a maneira como os cabeçalhos devem ser criados. Para criá-los (`UsernameToken` e `Timestamp`), realizaremos a interceptação da requisição, através de *Handlers* do JAX-WS. Estes *handlers* atuam de maneira a interceptar tanto a entrada como a saída da requisição, de maneira que temos a capacidade de utilizar o serviço normalmente, como fazíamos antes.

Para criar o *handler* que interceptará nossa requisição, basta criar uma classe que implemente a interface `javax.xml.ws.handler.soap.SOAPHandler`. Esta interface define os métodos `close`, `handleFault`, `handleMessage` e `getHeaders`. Destes, vamos nos concentrar no `handleMessage`, pois este método será executado antes de enviarmos a requisição.

A interface que implementamos, `javax.xml.ws.handler.soap.SOAPHandler`, possui um tipo genérico como parâmetro. Utilizamos a interface `javax.xml.ws.handler.soap.SOAPMessageContext` como tipo genérico para esta interface. Assim, o método `handleMessage` receberá como parâmetro uma instância desta interface:

```
package com.knight.estoque.servicos;

// imports omitidos

public class WSSecurityHandler
    implements SOAPHandler<SOAPMessageContext> {

    @Override
    public void close(MessageContext context) {
    }

    @Override
    public boolean handleFault(SOAPMessageContext context) {
        return true;
    }

    @Override
    public boolean handleMessage(SOAPMessageContext context) {
        return true;
    }
}
```



```

    }

    @Override
    public Set<QName> getHeaders() {
        return null;
    }
}

```

Quanto à criação dos *headers* de segurança, vamos utilizar o *framework* WSS4J para criar os cabeçalhos (que é o mesmo *framework* utilizado pelo Apache CXF para realizar as operações de segurança). Este *framework* conduz operações sobre SOAP no nível do XML, o que o torna ideal para ser utilizado dentro de um *handler*.

As classes que nos interessam do WSS4J são `org.apache.ws.security.message.WSSecHeader` (que representa o conjunto de cabeçalhos seguros), `org.apache.ws.security.message.WSSecUsernameToken` (que representa o *header* de usuário/senha) e `org.apache.ws.security.message.WSSecTimestamp` (que representa o *timestamp*).

Inicialmente, utilizamos `WSSecHeader` para “instalar” o mecanismo de segurança. Para isso, utilizamos a interface `org.w3c.dom.Document`, que representa o XML da mensagem. Esta informação pode ser obtida a partir da classe `javax.xml.soap.SOAPMessage`, que, por sua vez, é obtida a partir de `SOAPMessageContext`, passado como parâmetro para o método `handleMessage`. Para realizar a instalação, então, basta executar o método `insertSecurityHeader`, da classe `WSSecHeader`:

```

SOAPMessage message = context.getMessage();
WSSecHeader header = new WSSecHeader();

//obtém o Document da mensagem e insere o cabeçalho seguro
header.insertSecurityHeader(message.getSOAPBody()
    .getOwnerDocument());

```

Feito isso, passamos para o próximo passo: inserir informações de usuário e senha. Para isso, é preciso ajustar as informações em uma instância de `WSSecUsernameToken` e realizar a instalação na instância de `WSSecHeader` que foi criada. Basta executar o seguinte:

```
WSUsernameToken usernameToken = new WSUsernameToken();
usernameToken.setUserInfo(username, password);

//realiza o mesmo procedimento, de obter o Document da mensagem
usernameToken.prepare(message.getSOAPBody().getOwnerDocument());

//realiza a inserção do cabeçalho de usuário/senha
usernameToken.appendToHeader(header);
```

Opcionalmente, podemos utilizar esta classe para trabalhar com senhas codificadas em Base64, utilizando o método `setPasswordsAreEncoded`.

Finalmente, utilizamos a classe `WSTimestamp` para inserir o *header* de *timestamp*. A utilização desta classe é mais simples: basta executar o método `build`, passando como parâmetro o `Document` obtido da mensagem (como nas outras vezes) e a instância de `WSHeader`. Assim:

```
WSTimestamp timestamp = new WSTimestamp();
timestamp.build(message.getSOAPBody().getOwnerDocument(),
    header);
```

Feito isso, temos que prover um retorno para o método (que deve ser do tipo `boolean`). Este retorno é `true`, caso o processamento da requisição deva continuar, ou `false`, caso deva parar. Isso se tornará mais claro quando realizarmos a instalação do *handler* no sistema cliente.

Note que este método será executado tanto na saída de dados (ou seja, no envio de XML para o servidor) quando na entrada (ou seja, no recebimento de XML do servidor). Para executar este método apenas na saída de dados, utilizamos uma variável presente no contexto, que é referenciada a partir da constante `javax.xml.ws.handler.MessageContext.MESSAGE_OUTBOUND_PROPERTY`. Esta variável é do tipo `Boolean`, tendo o valor `true` caso o contexto seja de saída de dados e `false` caso seja de entrada.

Desta forma, o código completo dessa classe fica assim:

```
package com.knight.estoque.servicos;

import java.util.Set;
import javax.xml.namespace.QName;
import javax.xml.soap.*;
import javax.xml.ws.handler.*;
import javax.xml.ws.handler.soap.*;
```

```
import org.apache.ws.security.*;
import org.apache.ws.security.message.*;

public class WSSecurityHandler
    implements SOAPHandler<SOAPMessageContext> {

    private String username;
    private String password;

    private boolean encoded;

    public WSSecurityHandler(String username, String password) {
        this(username, password, false);
    }

    public WSSecurityHandler(String username, String password,
        boolean encoded) {
        this.username = username;
        this.password = password;
        this.encoded = encoded;
    }

    public void close(MessageContext context) {}

    public boolean handleFault(SOAPMessageContext context) {
        return true;
    }

    public boolean handleMessage(SOAPMessageContext context) {
        Boolean outbound = (Boolean) context
            .get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
        if (outbound) {
            try {
                SOAPMessage message = context.getMessage();
                WSSecHeader header = new WSSecHeader();
                header.insertSecurityHeader(message.getSOAPBody()
                    .getOwnerDocument());

                WSSecUsernameToken usernameToken = new WSSecUsernameToken();
                usernameToken.setUserInfo(username, password);
                usernameToken.setPasswordsAreEncoded(encoded);
            }
            catch (Exception e) {
                // Handle exception
            }
        }
    }
}
```

```

        usernameToken.prepare(message.getSOAPBody()
            .getOwnerDocument());
        usernameToken.appendToHeader(header);

        WSSecTimestamp timestamp = new WSSecTimestamp();
        timestamp.build(message.getSOAPBody().getOwnerDocument(),
            header);
    } catch (WSSecurityException | SOAPException e) {
        e.printStackTrace();
        return false;
    }
}
return true;
}

public Set<QName> getHeaders() {
    return null;
}
}

```

Agora, para instalar este *handler* no cliente, temos que realizar uma série de passos. A primeira, é instanciar o serviço de autores, como anteriormente:

```

AutoresService service = new AutoresServiceService()
    .getAutoresServicePort();

```

Com a engine JAX-WS, podemos realizar o *cast* do nosso *port* (no caso, *AutoresService*) para *javax.xml.ws.BindingProvider*. Esta interface irá conter uma série de métodos úteis para tratamento de baixo nível sobre o consumo do nosso *web service*. Um dos métodos dessa interface é *getHandlerChain*, que irá devolver uma lista de *handlers* para o serviço. Uma vez retornada esta lista, basta que adicionemos uma instância do nosso próprio *handler* e utilizemos o método *setHandlerChain*, desta forma:

```

AutoresService service = new AutoresServiceService()
    .getAutoresServicePort();

BindingProvider bindingProvider = (BindingProvider) service;

List<Handler> handlerChain = bindingProvider.getBinding()

```

```
.getHandlerChain();

handlerChain.add(new WSSecurityHandler("alexandre", "alexandre"));

bindingProvider.getBinding().setHandlerChain(handlerChain);
List<Autor> autores = service.listarAutores();

for (Autor autor : autores) {
    System.out.println(autor.getNome());
}
```

Para realizarmos o consumo do serviço, no entanto, ainda resta realizar um último passo. Como nosso serviço é exposto via HTTPS, precisamos incluir no *classpath* mecanismos que sejam capazes de tratar este aspecto. Para isso, realizamos a adição de dois JARs do Apache CXF ao *classpath*: `cxfrtfrontend-jaxws` e `cxfrt-transport-http`.

O GITHUB DO PROJETO

Para checar esse e outros aspectos do projeto, consulte o *github* que contém os códigos-fonte. O projeto do cliente, especificamente, está presente em <https://github.com/alesaudate/soa/tree/master/soa-capo6-estoque-cliente>.

Finalmente, ao executarmos o código, o Apache CXF será automaticamente instalado, provendo suítes para comunicação via HTTPS e outros aspectos. Desta forma, a primeira execução do cliente será lenta, ao passo que execuções subsequentes serão feitas de maneira mais rápida. Ao executar este código, obtemos uma série de *logs*, na saída padrão do console. Finalmente, então, obtemos os dados que estamos esperando:

Paulo Silveira
Adriano Almeida
Vinicius Baggio Fuentes

Desta forma, então, está concluído o cliente. Você pode efetuar novos testes fazendo a inclusão de novos usuários no sistema de usuários, e consultando este serviço utilizando o usuário recém-criado. Caso esteja tudo funcionando, você concluiu com sucesso a etapa de adição de segurança à sua aplicação orientada a serviços.

6.13 SUMÁRIO

Neste capítulo, você começou a desfrutar dos benefícios de uma aplicação orientada a serviços. Ao realizar a comunicação segura entre seus serviços, você está desfrutando de flexibilidade considerável, posto que a implementação do serviço de usuários não afeta em nada a implementação do serviço de tratamento de estoque da sua livreria.

O caminho até aqui foi árduo. Você pôde conferir, neste capítulo, várias das formas de segurança aplicáveis tanto a serviços REST como a serviços WS-*. Você pôde conferir detalhes de funcionamento de serviços REST com HTTP Basic e Digest, bem como algumas das formas de segurança presentes na especificação WS-Security.

Seus próximos passos, agora, estarão relacionados à manutenção de problemas mais ligados à natureza dessas aplicações, como a manutenção da escalabilidade, estabilidade e outros requisitos funcionais e não-funcionais.

CAPÍTULO 7

Design Patterns e SOA

“Não são as ervas más que sufocam a boa semente e sim a negligência do lavrador”
– Confúcio

Graças a você, a Knight.com está crescendo. Cada vez mais, novos clientes vão aparecendo, e sempre elevando o uso das APIs do seu sistema. Com isso, os primeiros problemas vão aparecendo: alguns sistemas saem do ar (causando a falha de outros), a manutenção dos serviços começa a se tornar um problema etc. Enquanto busca a solução, você se depara com um catálogo de *Design Patterns* para SOA. Você decide dar uma olhada.

7.1 INTEGRAÇÃO VERSUS SOA

Dos catálogos de *design patterns* SOA, o mais notório é *SOA Design Patterns*, de Thomas Erl [4]. No entanto, muitos dos padrões apresentados por ele não apresentam detalhes de implementação, mas sim, são de nível mais alto, como aqueles referentes à modelagem de formatos de dados, protocolos etc.

Isso porque SOA não diz respeito a detalhes de implementação. Neste ponto, muitos dos detalhes de como implementar uma arquitetura orientada a serviços se confundem com técnicas de pura integração de sistemas. Ou seja, faz com que SOA utilize técnicas de integração para ser “fisicalizada”. No entanto, várias das técnicas aplicáveis a integração de sistemas não são aplicáveis em SOA, o que faz com que ambas apenas compartilhem algumas características em comum, sem serem equivalentes.

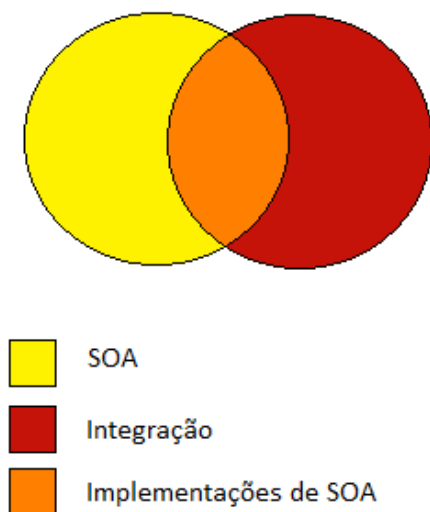


Figura 7.1: Relação de SOA e integração

Para nos mantermos focados na prática, mostrarei aqui *design patterns* ora relacionados a SOA, ora relacionados a integração de sistemas. Note que, pelo fato de serem assuntos que se misturam (conforme mostrado na figura 7.1), faz sentido utilizá-las em projetos SOA. Vamos apresentar apenas os *design patterns* mais relevantes, e deixo ao encargo do leitor pesquisar na bibliografia *design patterns* aplicáveis a cenários mais específicos.

7.2 O MODELO CANÔNICO

O Modelo Canônico é um dos *design patterns* mais importantes de SOA. Ele sintetiza o foco de SOA na reusabilidade e separação de responsabilidades, através de

uma técnica bastante simples — que, apesar disso, é ignorada em projetos puros de integração, sendo um dos pontos de divergência entre os dois modelos.

Para notar a existência e perceber a necessidade de um modelo canônico, tome o modelo de arquitetura que está sendo desenvolvido para a Knight.com: até agora, existe um sistema de gerenciamento de usuários e um sistema de gerenciamento de estoques. Caso este sistema precise de algo relacionado a usuários, ele não desenvolve o próprio, mas reutiliza o que já existe no sistema de usuários.

Para promover esta separação de responsabilidades, utiliza-se o Modelo Canônico. A técnica consiste, simplesmente, em manter XML Schemas separados por modelo de domínio para toda a empresa. Desta forma, suponha o seguinte XML Schema para o sistema de usuários:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://knight.com/usuarios/domain/v1"
  targetNamespace="http://knight.com/usuarios/domain/v1">

  <annotation>
    <documentation>
      Schema canônico para o sistema de usuários, versão 1.0
    </documentation>
  </annotation>

  <complexType name="usuario">
    <sequence>
      <element name="id" type="long" />
      <element name="nome" type="string" />
      <element name="login" type="string" />
      <element name="senha" type="string" />
    </sequence>
  </complexType>
</schema>
```

Este XML Schema é feito **antes** do desenvolvimento do sistema. Isto porque o padrão dá prioridade para a interação entre as aplicações envolvidas, ao invés de preferir a facilidade da geração de documentos de forma automatizada. Isso fará com que o desenvolvedor tenha pleno controle das interações necessárias entre os serviços, reduzindo custos de transformações de dados.

Note o `targetNamespace` deste *schema*, <http://knight.com/usuarios/domain/v1>. Não por acaso, ele recebe o seguinte formato: `http://<site da aplicação>`, sem

`www>/<nome da aplicação>/<a que se refere, na aplicação>/<versão, com o major number>`. Este formato não é obrigatório, mas é identificado empiricamente como sendo o mais adequado na maioria dos casos. Este tipo de *namespace* vai ajudar a diferenciar os modelos de domínio das aplicações, bem como suas versões.

Uma vez definido este *schema* (que é chamado de *schema* canônico, por pertencer à definição do modelo canônico), ele poderá ser reaproveitado por outras aplicações. Por exemplo, considere o XML Schema para o sistema de estoque:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://knight.com/estoque/domain/v1"
  xmlns:usuarios="http://knight.com/usuarios/domain/v1"
  targetNamespace="http://knight.com/estoque/domain/v1">

  <import
    namespace="http://knight.com/usuarios/domain/v1"
    schemaLocation="usuarios_v1_0.xsd" />

  <complexType name="autor">
    <sequence>
      <element name="id" type="long" minOccurs="0"/>
      <element name="dataNascimento" type="date" />
      <element name="nome" type="string" />
      <element name="usuario" type="usuarios:usuario" />
    </sequence>
  </complexType>
</schema>
```

Note a presença do atributo `usuario` na definição de `autor`. Esta é uma definição importada do sistema de usuários, daí a presença da *tag* `import`. Realizamos a definição do *namespace* de usuários logo no topo (repare a definição da *tag* `schema` — existe uma linha contendo “`xmlns:usuarios="http://knight.com/usuarios/domain/v1"`”). Para tratar estes dados corretamente, realizamos a importação do conteúdo, dando ao *schema* a localização de onde o *schema* de usuários está. Desta forma, não estamos fazendo qualquer redefinição de dados, mas reaproveitando o que já existe.

Para aproveitar estes dados, podemos definir as classes que utilizaremos diretamente através destes *schemas*, utilizando a ajuda de editores como o Eclipse ou por ferramentas de linha de comando presentes na JDK. Por exemplo, para realizarmos

a definição das classes utilizando o Eclipse, basta clicar com o botão direito em cima do *schema* desejado, apontar para *Generate* e clicar em *JAXB Classes...*, de acordo com a figura:



Figura 7.2: Gerando as classes JAXB a partir dos schemas

Feito isso, basta selecionar o projeto onde posicionar as classes geradas:

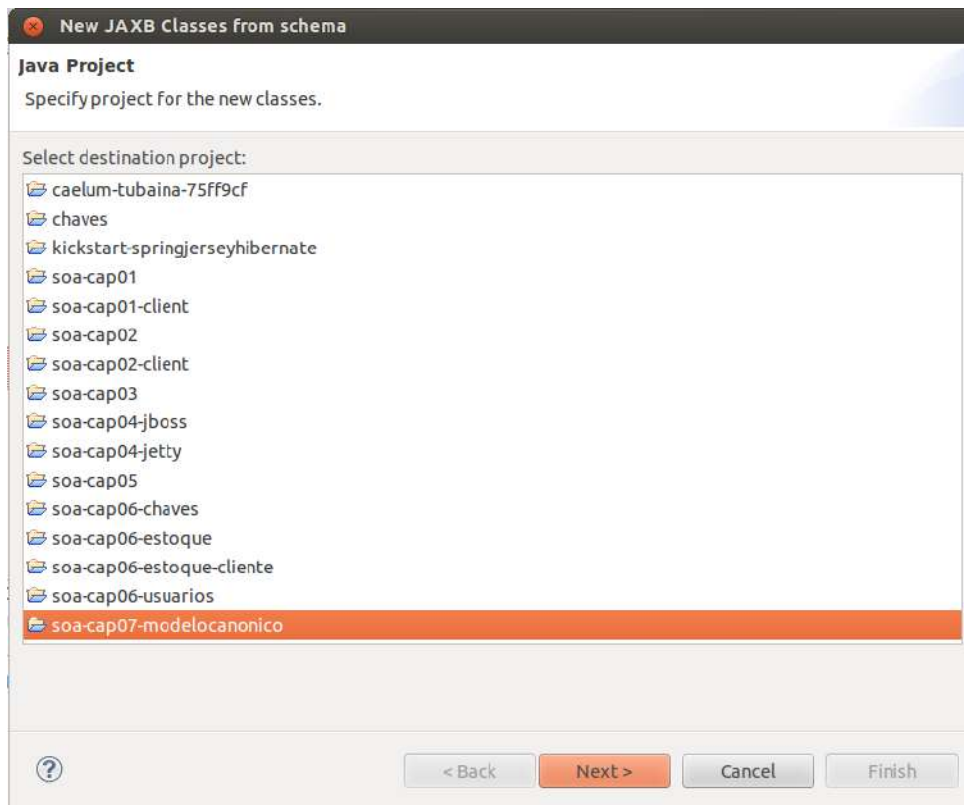


Figura 7.3: Selecione o projeto onde colocar as classes geradas pelo JAXB

Na sequência, ele abrirá uma janela para customização de detalhes da geração. Não se importe com nenhum dos detalhes, basta clicar em *Finish*:

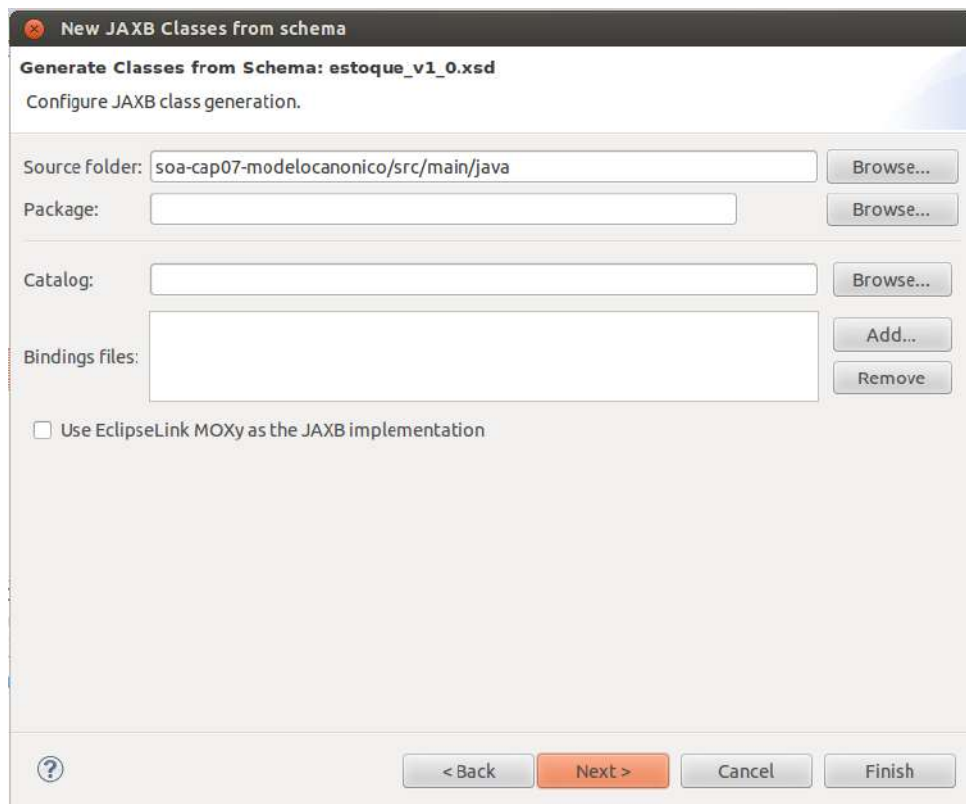


Figura 7.4: Customização de detalhes da geração de código

Dado isso, o JAXB emitirá um alerta dizendo que, caso alguma classe gerada entre em conflito com uma pré-existente, a essa última será então sobrescrita. Clique em Yes:

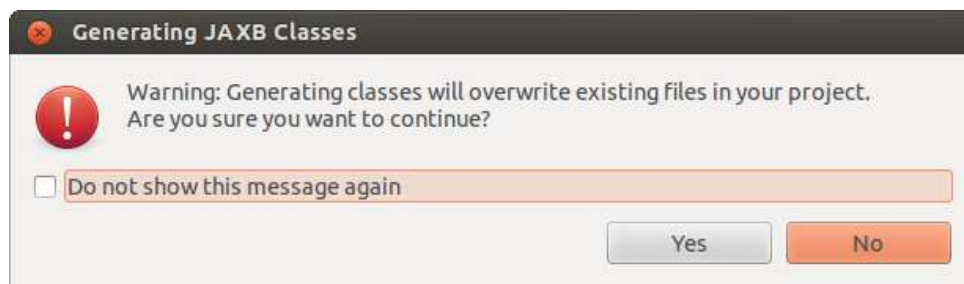


Figura 7.5: Alerta de sobrescrita de classes do JAXB

Finalmente, o JAXB realizará a compilação de todos os *schemas* envolvidos e criará as classes necessárias:

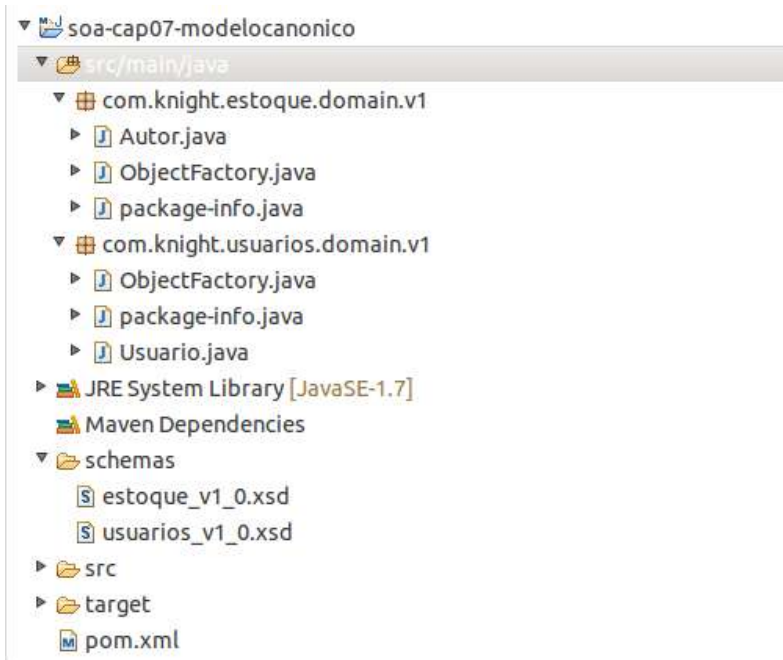


Figura 7.6: Estrutura de classes geradas pelo JAXB

Caso queira, também é possível atingir o mesmo resultado utilizando uma ferramenta da JDK chamada `xjc`. Para atingir o mesmo resultado obtido com o Eclipse, basta utilizar esta ferramenta fornecendo o argumento `-d` para especificar o destino

dos arquivos gerados:

```
$JDK_HOME/bin/xjc -d $WORKSPACE/$PROJETO/$PASTA_DE_FONTES  
$WORKSPACE/$PROJETO/schemas/estoque_v1_0.xsd
```

Por exemplo, em um Ubuntu, você pode efetuar algo como o seguinte comando:

```
/usr/lib/jvm/java-7-oracle/bin/xjc  
-d ~/workspace/soa/soa-cap07-modelocanonico/src/main/java/  
~/workspace/soa/soa-cap07-modelocanonico/schemas/estoque_v1_0.xsd
```

Note que uma das desvantagens deste modelo é que ele dificulta a adoção de *Domain-Driven Design*, já que as classes geradas podem ser sobrescritas (de acordo com a figura 7.2). No entanto, você pode achar uso para o padrão *Adapter* descrito pelo *Gang of Four* [1] — sendo esta técnica aconselhada por Eric Evans [5].

7.3 DESENVOLVIMENTO CONTRACT-FIRST

Uma extensão natural do desenvolvimento do modelo canônico é a técnica conhecida como desenvolvimento *contract-first*. Esta técnica consiste em desenvolver os WSDLs **antes** do código propriamente dito, dando ao desenvolvedor total controle sobre o tráfego de dados. Como você viu no capítulo 2, existem várias seções a serem tratadas em WSDLs, e lidar com elas não é exatamente fácil. No entanto, como você pode conferir no capítulo 6, pode ser necessário obter total domínio sobre os WSDLs para não deixar que eles te dominem.

Obviamente, também não é necessário desenvolver todo o conteúdo do WSDL manualmente — podemos utilizar a IDE para ajudar neste sentido. Para utilizar o Eclipse, por exemplo, basta clicar com o botão direito sobre a pasta onde deseja colocar o WSDL gerado, apontar para *New* e clicar em *Other...*:

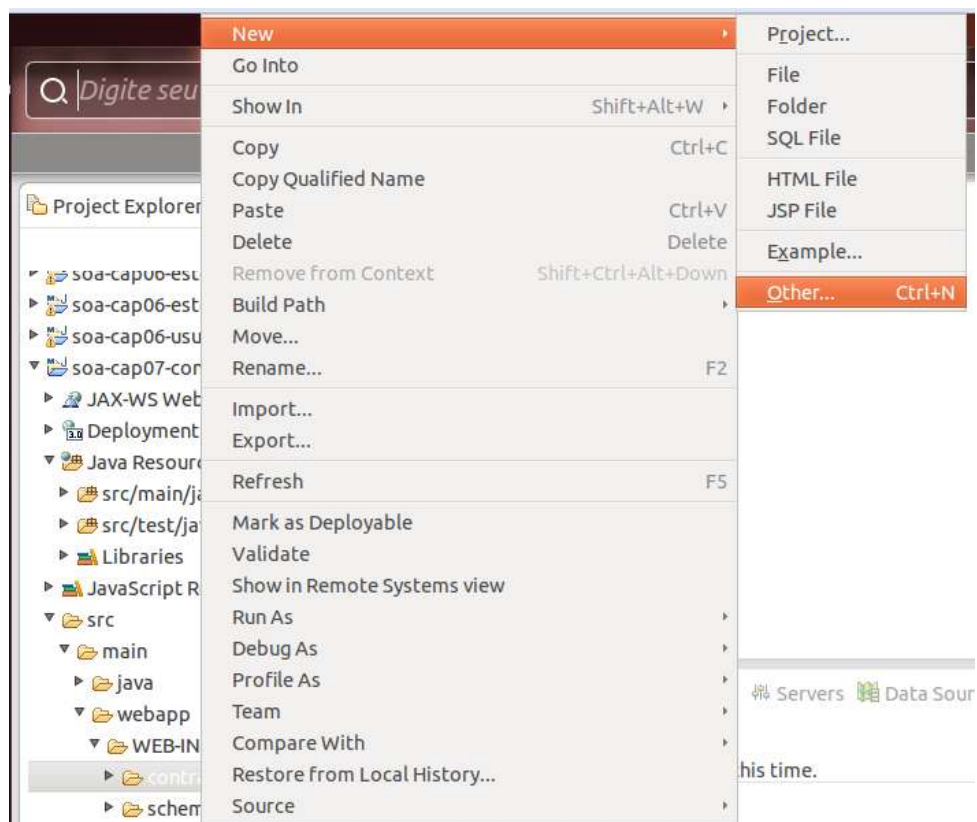


Figura 7.7: Menu de criação do WSDL pelo Eclipse

Feito isso, basta selecionar o *wizard* WSDL File e clicar em Next:

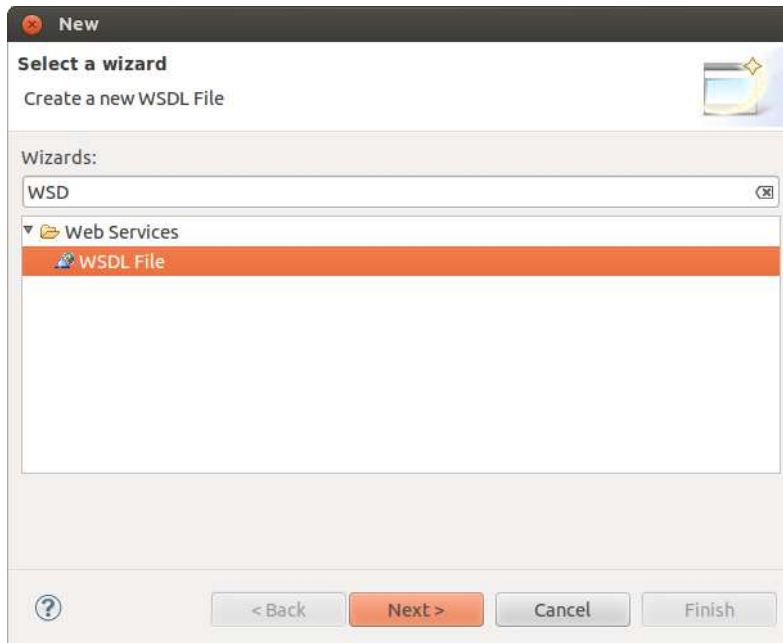


Figura 7.8: Seleção de wizard para WSDLs do Eclipse

Na sequência, dê um nome para o arquivo do novo WSDL e clique em `Next`:

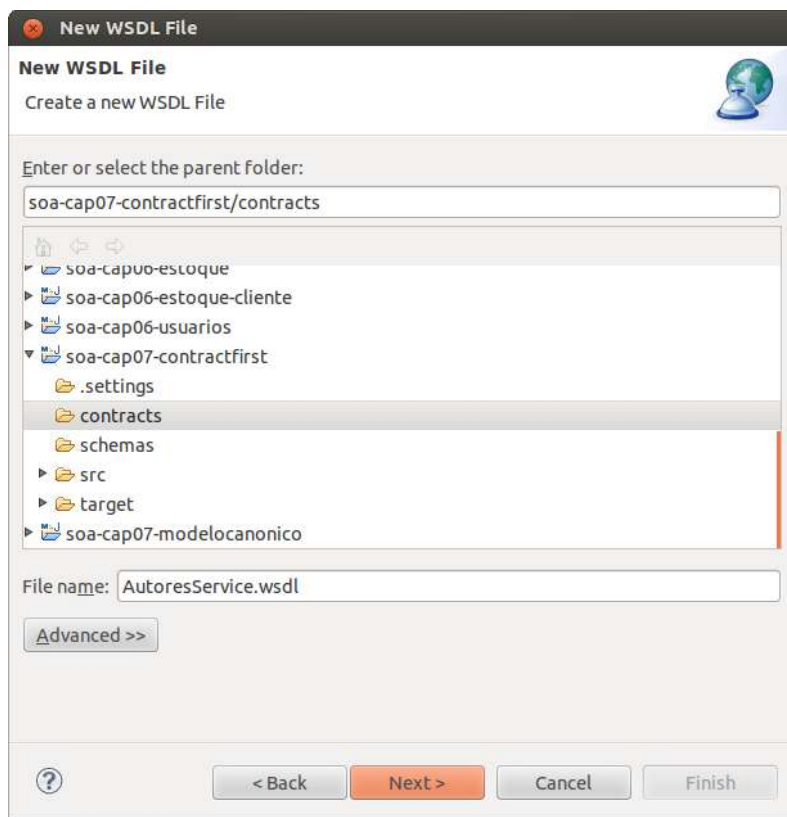


Figura 7.9: Dê um nome para o arquivo que irá conter seu WSDL

Finalmente, especifique as opções para criação do WSDL. Note, na figura, que está sendo criado o WSDL para o serviço de autores, com o *namespace* <http://knight.com/estoque/services/AutoresService/v1>. Quanto ao resto, deixe como está e clique em `Finish`:

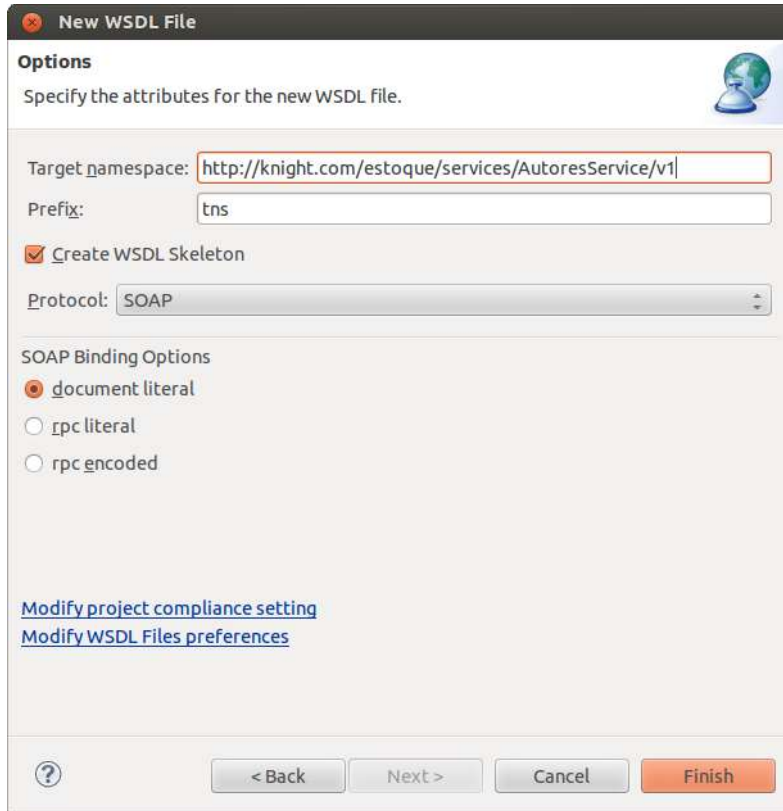


Figura 7.10: Opções de criação do WSDL

Ao final, o Eclipse gerou um *template* do WSDL para você. Agora, você deverá customizar o contrato manualmente. Lembre-se do que você leu no capítulo 2 e comece pela seção `types`. Nesta seção, crie primeiro os elementos que vão conter os parâmetros de entrada e saída do seu serviço, desta forma:

```
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://knight.com/estoque/services/AutoresService/v1"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  name="AutoresService"
  targetNamespace=
    "http://knight.com/estoque/services/AutoresService/v1">
  <wsdl:types>
    <xsd:schema targetNamespace=
```

```

        "http://knight.com/estoque/services/AutoresService/v1">
        <xsd:element name="listarAutores">
            <xsd:complexType />
        </xsd:element>
        <xsd:element name="listarAutoresResponse">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="out" type="xsd:string" />
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:schema>
</wsdl:types>
<!-- restante do WSDL -->
</wsdl:definitions>

```

Agora, resta você referenciar, efetivamente, os elementos do seu modelo canônico. Como a seção `types` contém, na verdade, um XML Schema, basta utilizar a `tag import`, da mesma maneira como vimos na definição do próprio `schema`. Além disso, você também deve criar um prefixo para o `namespace` importado, por exemplo, `estoque`:

```

<wsdl:types>
    <xsd:schema
targetNamespace="http://knight.com/estoque/services/AutoresService/v1"
    xmlns:estoque="http://knight.com/estoque/domain/v1">
        <xsd:import namespace="http://knight.com/estoque/domain/v1"
            schemaLocation="../schemas/estoque_v1_0.xsd" />
        <xsd:element name="listarAutores">
            <xsd:complexType />
        </xsd:element>

        <xsd:element name="listarAutoresResponse">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="autor" type="estoque:autor"
                        maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:schema>

```

```
</wsdl:types>
```

Na sequência, você deve modelar a seção `messages`. Novamente, não existe padronização para isso, mas o Eclipse nos dá uma dica de manter o nome da operação e colocar `Request` ou `Response` ao final do nome de cada `message`. Seguindo esta dica, obtemos o seguinte:

```
<wsdl:message name="listarAutoresRequest">
  <wsdl:part element="tns:listarAutores" name="parameters" />
</wsdl:message>
<wsdl:message name="listarAutoresResponse">
  <wsdl:part element="tns:listarAutoresResponse" name="parameters" />
</wsdl:message>
```

Feito isso, modelamos a seção `portType`. O Eclipse, inicialmente, gera a seguinte estrutura:

```
<wsdl:portType name="AutoresService">
  <wsdl:operation name="NewOperation">
    <wsdl:input message="tns:NewOperationRequest" />
    <wsdl:output message="tns:NewOperationResponse" />
  </wsdl:operation>
</wsdl:portType>
```

Tudo o que temos que fazer aqui é adaptar o nome da operação e das mensagens de entrada e saída. Desta forma, obtemos o seguinte:

```
<wsdl:portType name="AutoresService">
  <wsdl:operation name="listarAutores">
    <wsdl:input message="tns:listarAutoresRequest" />
    <wsdl:output message="tns:listarAutoresResponse" />
  </wsdl:operation>
</wsdl:portType>
```

A próxima seção é `binding`. Como ela faz referência a `portType`, basta alterar o nome da operação e `soapAction` para ter um valor mais condizente com a operação `listarAutores`. Desse modo, deixamos esta seção da seguinte forma:

```
<wsdl:binding name="AutoresServiceSOAP" type="tns:AutoresService">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="listarAutores">
```

```

    <soap:operation
      soapAction="AutoresService/ListarAutores" />
    <wsdl:input>
      <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>

```

Finalmente, precisamos modificar a seção `service`. A única informação desta seção que precisamos alterar é o endereço do serviço. Contudo, quando formos realizar o *deploy* deste serviço, é esperado que o servidor altere esta seção para o endereço efetivo. Portanto, podemos colocar qualquer valor como endereço do serviço. Para deixar esta intenção explícita, colocamos algum valor qualquer, como `ENDereco`:

```

<wsdl:service name="AutoresService">
  <wsdl:port binding="tns:AutoresServiceSOAP"
    name="AutoresServiceSOAP">
    <soap:address location="ENDereco" />
  </wsdl:port>
</wsdl:service>

```

Finalmente podemos gerar a implementação do serviço. Poderíamos utilizar o Eclipse para gerar a implementação; contudo, pode ser mais complicado gerar essa implementação pela IDE do que pela ferramenta de linha de comando. Então, vamos utilizar o `wsimport` para realizar essa geração.

O uso do `wsimport` é semelhante ao do `xjc`:

```

$JDK_HOME/bin/wsimport -s $WORKSPACE/$PROJETO/$PASTA_DE_FONTES
  $WORKSPACE/$PROJETO/contracts/AutoresService.wsdl

```

Ao parâmetro `-s` informamos o diretório onde os arquivos-fontes serão gerados. Um exemplo de comando seria:

```

/usr/lib/jvm/java-7-oracle/bin/wsimport
-s ~/workspace/soa/soa-cap07-contractfirst/src/main/java/
~/workspace/soa/soa-cap07-contractfirst/contracts/AutoresService.wsdl

```

Isso vai gerar a seguinte estrutura:

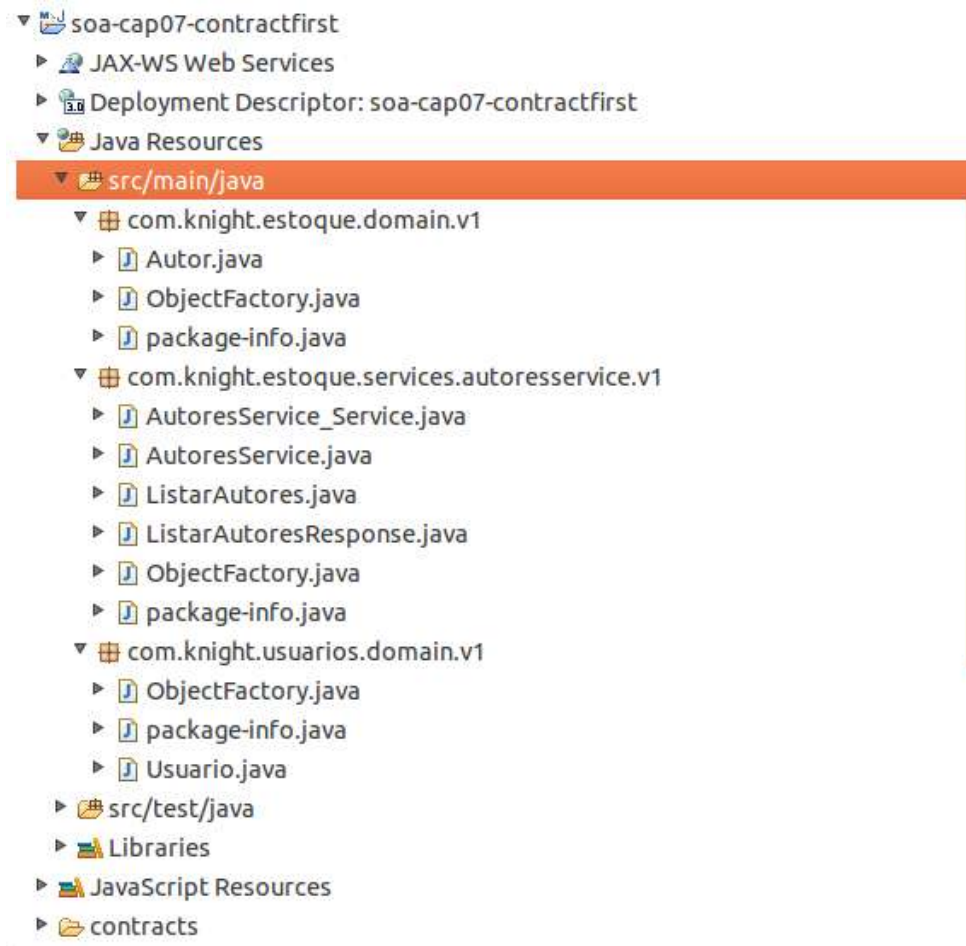


Figura 7.11: Estrutura gerada pelo wsimport

Note que o propósito do `wsimport` é, na verdade, gerar clientes para os serviços. Como a JDK não possui ferramentas para geração de serviços a partir destes contratos, utilizamos a interface gerada pelo `wsimport` para realizar a criação do serviço. A interface que nos interessa, no caso, é `AutoresService`, que possui o seguinte código:

```
package com.knight.estoque.services.autoresservice.v1;

// imports omitidos
```

```

@WebService(
    name = "AutoresService",
    targetNamespace =
        "http://knight.com/estoque/services/AutoresService/v1")
@XmlSeeAlso({
    com.knight.estoque.domain.v1.ObjectFactory.class,
    com.knight.estoque.services.autoresservice.v1.ObjectFactory.class,
    com.knight.usuarios.domain.v1.ObjectFactory.class
})
public interface AutoresService {

    @WebMethod(action = "AutoresService/ListarAutores")
    @WebResult(name = "autor", targetNamespace = "")
    @RequestWrapper(localName = "listarAutores",
        targetNamespace =
            "http://knight.com/estoque/services/AutoresService/v1",
        className =
            "com.knight.estoque.services.autoresservice.v1.ListarAutores")
    @ResponseWrapper(localName = "listarAutoresResponse",
        targetNamespace =
            "http://knight.com/estoque/services/AutoresService/v1",
        className =
            "com.knight.estoque.services.autoresservice.v1."
            + "ListarAutoresResponse")
    public List<Autor> listarAutores();

}

```

Agora precisamos implementar esta interface:

```

package com.knight.estoque.services.autoresservice.v1;

// imports omitidos

public class AutoresServiceImpl implements AutoresService {

    @WebMethod(action = "AutoresService/ListarAutores")
    @WebResult(name = "autor", targetNamespace = "")
    @RequestWrapper(localName = "listarAutores",
        targetNamespace =
            "http://knight.com/estoque/services/AutoresService/v1",
        className =

```



```

        "com.knight.estoque.services.autoresservice.v1.ListarAutores")
    @ResponseWrapper(localName = "listarAutoresResponse",
        targetNamespace =
            "http://knight.com/estoque/services/AutoresService/v1",
        className = "com.knight.estoque.services.autoresservice" +
            ".v1.ListarAutoresResponse")
    public List<Autor> listarAutores() {
        // TODO Auto-generated method stub
        return null;
    }
}

```

Repare que esta classe, agora, precisa da anotação `@WebService` para que possa ser devidamente instalada. No entanto, antes, é necessário mover os contratos e schemas para a pasta `WEB-INF` do projeto. Feito isso, basta incluir os seguintes atributos:

- `endpointInterface`: deve conter o nome completamente qualificado da interface que o serviço implementa (no caso, `com.knight.estoque.services.autoresservice.v1.AutoresService`)
- `serviceName`: corresponde ao valor do atributo `name` presente na *tag* `service`, no WSDL. No nosso caso, é `AutoresService`.
- `portName`: corresponde ao valor do atributo `name` presente na *tag* `port` (que é filha de `service`). No nosso caso, é `AutoresServiceSOAP`.
- `targetNamespace`: corresponde ao *namespace* do WSDL. No nosso caso, é `http://knight.com/estoque/services/AutoresService/v1`.
- `wsdlLocation`: corresponde ao caminho do WSDL, dentro do projeto. Movendo a pasta `contracts` e `schemas` para dentro da pasta `WEB-INF`, o valor no nosso caso é `WEB-INF/contracts/AutoresService.wsdl`.

Assim, a definição da classe fica:

```

@WebService(
    endpointInterface =
        "com.knight.estoque.services.autoresservice.v1.AutoresService",
    portName = "AutoresServiceSOAP",
    serviceName = "AutoresService",
    targetNamespace =

```

```
        "http://knight.com/estoque/services/AutoresService/v1",
        wsdlLocation = "WEB-INF/contracts/AutoresService.wsdl")
public class AutoresServiceImpl implements AutoresService {
    // implementação da classe
}
```

Ao realizar o *deploy* da classe, o seguinte deve aparecer no *console* do JBoss:

```
id=com.knight.estoque.services.autoresservice.v1.AutoresServiceImpl
address=http://localhost:8080/soa-cap07-contractfirst-0.0.1-SNAPSHOT
/AutoresService
implementor=com.knight.estoque.services.autoresservice.v1.
AutoresServiceImpl
invoker=org.jboss.wsf.stack.cxf.JBossWSInvoker
serviceName={http://knight.com/estoque/services/AutoresService/
v1}AutoresService
portName={http://knight.com/estoque/services/AutoresService/
v1}AutoresServiceSOAP
wsdlLocation=null
mtomEnabled=false
```

Quando este conteúdo aparecer, significa que o *deploy* foi realizado corretamente. Os dados do serviço implantado podem ser obtidos a partir de <http://localhost:8080/soa-cap07-contractfirst-0.0.1-SNAPSHOT/AutoresService?wsdl> (endereço inferido a partir das informações da implantação).

7.4 SERVIÇOS ASSÍNCRONOS COM WS-ADDRESSING

Muitas vezes, pode ser necessário invocar uma operação cujo resultado não pode ser oferecido imediatamente. Estes são o caso de relatórios emitidos pelo sistema. Quando um volume de informações muito grande deve ser levantado, geralmente é mais interessante realizar a operação de maneira assíncrona, ou seja, o cliente faz a requisição e recebe imediatamente uma resposta dizendo que a requisição será processada. O servidor, então, inicia o processamento e quando terminar, invoca o solicitante.

A especificação responsável por prover facilidades relacionadas ao endereçamento de mensagens é a WS-Addressing. Com esta especificação, é possível redirecionar mensagens, e é a partir desta que podemos informar ao servidor qual o endereço para resposta de uma determinada operação. Assim como a WS-Security,

a WS-Addressing é ativada utilizando WS-Policy como apoio. No entanto, as alterações realizadas pela WS-Addressing vão permear todo o WSDL.

A seguir, você verá as alterações necessárias para criar uma operação assíncrona para solicitação de relatórios de criação de autores no sistema de estoques da Knight.com.

Namespaces

Os *namespaces* a serem referenciados no WSDL são os seguintes: <http://www.w3.org/2007/05/addressing/metadata> e <http://www.w3.org/2006/05/addressing/wSDL>, além do *namespace* de WS-Policy, <http://www.w3.org/ns/ws-policy>. A definição dos *namespaces* no contrato do serviço de autores fica assim:

```
<wSDL:definitions xmlns:soap="http://schemas.xmlsoap.org/wSDL/soap/"
  xmlns:tns="http://knight.com/estoque/services/AutoresService/v1"
  xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
  xmlns:wsaw="http://www.w3.org/2006/05/addressing/wSDL"
  xmlns:wsp="http://www.w3.org/ns/ws-policy"
  name="AutoresService"
  targetNamespace=
    "http://knight.com/estoque/services/AutoresService/v1">
```

As mensagens de entrada e saída

Por estarmos tratando de um serviço assíncrono, o serviço que receberá a resposta da requisição não necessariamente será o mesmo que enviou a mesma. No entanto, a mensagem de resposta trafegada pelo serviço processador da requisição deve ser **exatamente igual** à mensagem que o serviço que a receberá espera. Neste ponto, torna-se bastante útil a abordagem **contract-first**, pois temos a habilidade de controlar isto de maneiras mais fáceis do que pelas anotações JAXB/JAX-WS.

Vamos supor a adição de mais uma operação no serviço de autores, `solicitarRelacaoDeAutores`. Esta operação terá WS-Addressing habilitado, mas não necessariamente deverá responder sempre de modo assíncrono, apenas quando explicitamente solicitado pelo cliente.

As definições da requisição nova serão exatamente iguais às normais, com a adição de elementos WS-Addressing. Estes elementos irão permear as seções `portType` e `binding`, bem como requerer a criação de uma seção contendo a

política WS-Addressing.

A seção `portType`

A seção `portType` deverá ser alterada para incluir informações de *actions*. Estas *actions* irão dizer ao mecanismo WS-Addressing qual o método para as quais as mensagens devem ser entregues, e devem ser definidas nas seções `input` e `output` do `portType`, desta forma:

```
<wsdl:portType name="AutoresService">
  <!-- definição de listagem de autores, que permanece inalterada -->
  <wsdl:operation name="solicitarRelacaoDeAutores">
    <wsdl:input message="tns:solicitarRelacaoDeAutoresRequest"
      wsam:Action="AutoresService/solicitarRelacaoDeAutores"/>

    <wsdl:output message="tns:solicitarRelacaoDeAutoresResponse"
      wsam:Action=
        "AutoresService/solicitarRelacaoDeAutoresResponse"  />
  </wsdl:operation>
</wsdl:portType>
</wsdl:portType>
```

As *actions* podem ter qualquer valor — você apenas deve certificar-se de que elas são únicas.

A seção `binding`

A seção `binding` é a mais afetada pela inclusão de WS-Addressing. Como queremos incluir esta especificação apenas para uma única operação, incluímos as informações para uso de WS-Addressing e da política de uso da mesma apenas na definição da operação `solicitarRelacaoDeAutores`, desta forma:

```
<wsdl:binding name="AutoresServiceSOAP" type="tns:AutoresService">
  <!-- Definição da operação listarAutores, inalterada -->
  <wsdl:operation name="solicitarRelacaoDeAutores">
    <soap:operation soapAction="" />
    <wsaw:UsingAddressing />
    <wsp:PolicyReference URI=
      "#WSAddressingAutoresServiceSoapBinding_WSAM_Addresssing_Policy" />
    <wsdl:input>
      <soap:body use="literal" />
  </wsdl:operation>
</wsdl:binding>
```

```
</wsdl:input>
<wsdl:output>
  <soap:body use="literal" />
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
```

Note que a definição do atributo `soapAction`, aqui, **deve ser mantida vazia**. Isto porque pode haver conflitos entre esta informação e as definições de WS-Addressing. Observe, também, que o elemento `UsingAddressing` contém o atributo `required`, com valor igual a `false`. Este é um dos mecanismos necessários para desobrigar o cliente a passar informações relativas a WS-Addressing. Caso você queira obrigá-lo, apenas remova esta declaração e a declaração de opcional na política, a ser mostrada a seguir.

A definição da política WS-Addressing

Assim como em WS-Security, WS-Addressing também requer a definição de uma política para uso. No entanto, esta é bem mais simples: basta declarar o elemento `Addressing` na política:

```
<wsp:Policy
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-
200401-wss-wssecurity-utility-1.0.xsd" wsu:Id=
  "WSAddressingAutoresServiceSoapBinding_WSAM_Addresssing_Policy">
  <wsam:Addressing wsp:Optional="true">
    <wsp:Policy />
  </wsam:Addressing>
</wsp:Policy>
```

Perceba, aqui, o elemento `Optional`, com valor `true`. Este é um mecanismo utilizado para desobrigar o cliente a transmitir informações de WS-Addressing (ou seja, transformando o serviço em síncrono/assíncrono, dependendo da requisição do cliente). Caso você queira obrigar os clientes a sempre transmitirem as informações, basta retirar esta declaração.

O receptor da mensagem

O receptor das mensagens será um *web service* simples, com duas alterações: o método que receberá o resultado da requisição (ou seja, o *callback*) não deve posuir retorno (ou seja, a declaração *output*) e a mensagem de entrada deve ser igual à

mensagem de saída do serviço que origina a mensagem. Assim, como a requisição terá como resultado uma lista de autores, a entrada do *callback* também será uma listagem de autores. Segue abaixo a definição do grande WSDL do *callback*:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://knight.com/estoque/services/AutoresService/v1"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
  xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
  xmlns:wsp="http://www.w3.org/ns/ws-policy"
  name="AutoresService"
  targetNamespace=
    "http://knight.com/estoque/services/AutoresService/v1">
  <wsdl:types>
    <xsd:schema targetNamespace="http://knight.com/estoque/services
      /AutoresService/v1"
      xmlns:estoque="http://knight.com/estoque/domain/v1" >
      <xsd:import namespace="http://knight.com/estoque/domain/v1"
        schemaLocation="../schemas/estoque_v1_0.xsd" />
      <xsd:element name="solicitarRelacaoDeAutoresResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="autor"
              type="estoque:autor"
              maxOccurs="unbounded" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="solicitarRelacaoDeAutoresResponse">
    <wsdl:part element="tns:solicitarRelacaoDeAutoresResponse"
      name="parameters" />
  </wsdl:message>
  <wsdl:portType name="AutoresServiceCallback">
    <wsdl:operation name="solicitarRelacaoDeAutoresCallback">
      <wsdl:input message="tns:solicitarRelacaoDeAutoresResponse" />
    </wsdl:operation>
  </wsdl:portType>
```

```

<wsdl:binding name="AutoresServiceCallbackSOAP"
    type="tns:AutoresServiceCallback">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="solicitarRelacaoDeAutoresCallback">
        <soap:operation soapAction="" />
        <wsdl:input>
            <soap:body use="literal" />
        </wsdl:input>
    </wsdl:operation>
</wsdl:binding>
<wsdl:service name="AutoresServiceCallback">
    <wsdl:port binding="tns:AutoresServiceCallbackSOAP"
        name="AutoresServiceCallbackSOAP">
        <soap:address location="ENDereco" />
    </wsdl:port>
</wsdl:service>
</wsdl:definitions>

```

A geração deste serviço, de acordo com a técnica mencionada na seleção 7.2, produz a seguinte interface:

```

@WebService(name = "AutoresServiceCallback",
    targetNamespace =
        "http://knight.com/estoque/services/AutoresService/v1")
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
@XmlSeeAlso({
    com.knight.estoque.domain.v1.ObjectFactory.class,
    com.knight.estoque.services.autoresservice.v1.ObjectFactory.class,
    com.knight.usuarios.domain.v1.ObjectFactory.class
})
public interface AutoresServiceCallback {

    @WebMethod
    @Oneway
    public void solicitarRelacaoDeAutoresCallback(
        @WebParam(name = "solicitarRelacaoDeAutoresResponse",
            targetNamespace =
                "http://knight.com/estoque/services/AutoresService/v1",
            partName = "parameters")
        SolicitarRelacaoDeAutoresResponse parameters);

```

```
}
```

Note que a classe `SolicitarRelacaoDeAutoresResponse` foi gerada porque o nome da operação e do parâmetro para a operação são incompatíveis. De qualquer forma, esta classe contém uma lista de autores; basta executar o método `getAutor`.

Testando com SoapUI

Para realizar os testes com o SoapUI, basta importar o projeto normalmente. Primeiro, ele habilitará as requisições para ambas as operações normalmente. Se você fizer a requisição para `solicitarRelacaoDeAutores` sem qualquer modificação, obterá uma resposta síncrona. Para habilitar a requisição assíncrona, você deve incluir no cabeçalho as informações de WS-Addressing. O conjunto mínimo que você deve passar é `Action` e `MessageID`; para habilitar a requisição assíncrona, basta passar como parâmetro o elemento `ReplyTo`.

A `Action` deverá ter o valor do que está na implementação do serviço - no nosso caso, `AutoresService/solicitarRelacaoDeAutores`. A `MessageID`, por sua vez, é o ID da mensagem. Trata-se de uma implementação do padrão de integração *Correlation Identifier* (<http://eapatterns.com/CorrelationIdentifier.html>). `ReplyTo`, por sua vez, irá conter o endereço de retorno da requisição.

“O PADRÃO CORRELATION IDENTIFIER”

O uso de um *correlation identifier* é importante para que o remetente da mensagem mantenha um rastreamento das mensagens que enviou. Num cenário em que várias mensagens são enviadas e recebidas de volta, o *correlation identifier* é o mecanismo que o cliente utiliza para saber do que se trata a mensagem. A implementação, em si, é bem simples: trata-se apenas de um ID (como se fosse o ID de uma linha em um banco de dados), que referencia cada mensagem de maneira única.

A requisição do SoapUI deverá ter o seguinte formato:

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:v1="http://knight.com/estoque/services/AutoresService/v1">
```



```

<soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
  <wsa:Action>AutoresService/solicitarRelacaoDeAutores</wsa:Action>
  <wsa:MessageID>
    4585ee4b-5e25-4e82-a91f-0f79bb330b6a
  </wsa:MessageID>
  <wsa:ReplyTo>
    <wsa:Address>
      <!-- Endereço do serviço de callback -->
    </wsa:Address>
  </wsa:ReplyTo>
</soapenv:Header>
<soapenv:Body>
  <v1:solicitarRelacaoDeAutores>
    <desde>2012-12-12</desde>
  </v1:solicitarRelacaoDeAutores>
</soapenv:Body>
</soapenv:Envelope>

```

Após o envio da requisição, a seguinte resposta deverá ser exibida no console do SoapUI:

```

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <MessageID xmlns="http://www.w3.org/2005/08/addressing">
      urn:uuid:b7a9385d-a1f3-4600-910d-6767002a7da4
    </MessageID>
    <To xmlns="http://www.w3.org/2005/08/addressing">
      http://www.w3.org/2005/08/addressing/anonymous
    </To>
    <ReplyTo xmlns="http://www.w3.org/2005/08/addressing">
      <Address>http://www.w3.org/2005/08/addressing/none</Address>
    </ReplyTo>
    <RelatesTo xmlns="http://www.w3.org/2005/08/addressing">
      http://www.w3.org/2005/08/addressing/unspecified
    </RelatesTo>
  </soap:Header>
  <soap:Body/>
</soap:Envelope>

```

Isso significa que a requisição foi aceita. Caso o serviço de *callback* esteja implementado corretamente, algo como o seguinte deverá aparecer no console do JBoss:

```
17:24:40,061 INFO [stdout] (default-workqueue-3) Callback imprimindo:
17:24:40,062 INFO [stdout] (default-workqueue-3) Alexandre Saudate
17:24:40,063 INFO [stdout] (default-workqueue-3) Adriano Almeida
17:24:40,063 INFO [stdout] (default-workqueue-3) Paulo Silveira
```

Implementando um cliente Java

Finalmente, o último passo é criar o cliente que fará essa requisição. Infelizmente, este é um dos passos mais complexos, já que, até o momento, este autor não localizou nenhuma biblioteca que faça estas adaptações na requisição de forma satisfatória. Portanto, a requisição deverá ser adaptada de forma apenas semiautomatizada, utilizando *handlers* (da mesma forma como foi feita no tratamento de segurança).

O primeiro passo é gerar o cliente normalmente. Depois, faremos uso de uma *feature* (ou seja, uma classe que estende `javax.xml.ws.WebServiceFeature`) que faz o reconhecimento de questões mais triviais relacionadas a WS-Addressing. Esta *feature* é `javax.xml.ws.soap.AddressingFeature`, e deve ser passada como parâmetro durante a obtenção da representação do serviço, assim:

```
service = new AutoresService_Service().
    getAutoresServiceSOAP(new AddressingFeature());
```

Depois, faremos a adaptação com o *handler*. Ele deve endereçar duas questões principais: inserir o endereço do serviço para resposta assíncrona e cuidar para que os clientes assíncronos não tenham uma exceção (como o elemento `Body` vem vazio na requisição assíncrona, o *parser* JAX-WS tenta fazer o *parse* e, não conseguindo, lança uma exceção). Esta adaptação é feita através de uma mistura da API para tratamento de mensagens SOAP e da API de tratamento de XML - DOM.

Para endereçar estas questões, tudo o que faremos será obter o elemento `Address` (que já deverá ter sido colocado pela *engine* JAX-WS) e obter o primeiro elemento do corpo da requisição, acrescentando `Response` ao final. Isto é feito através do código:

```
private void trataRequisicao(SOAPMessageContext context)
    throws SOAPException {
    SOAPMessage soapMessage = context.getMessage();
    NodeList nodeList = soapMessage.getSOAPHeader().getElementsByTagName(
        "Address");
    Node node = nodeList.item(0);
```

```

node.setTextContent(enderecoResposta);

Node requestNode = soapMessage.getSOAPBody().getFirstChild();
String namespace = requestNode.getNamespaceURI();
String nodeName = requestNode.getLocalName() + "Response";

QName qName = new QName(namespace, nodeName);
this.responseQName = qName;
}

```

Note que o elemento da requisição foi atribuído a uma variável da instância do *handler*. Quando esta requisição retornar, o seguinte será executado:

```

private void trataResposta(SOAPMessageContext context)
    throws SOAPException {
    SOAPMessage soapMessage = context.getMessage();
    soapMessage.getSOAPBody().addChildElement(this.responseQName);
}

```

Finalmente, ao enviar a requisição, basta que o cliente assíncrono informe o endereço para resposta através da instância do *handler*. O código do cliente fica assim:

```

private static boolean ehAssincrono = true;
private String enderecoResposta =
    "http://localhost:8080/soa-cap07-wsaddressing-servidor-0.0.1-SNAPSHOT"
    + "/AutoresServiceCallback";

public static void main(String[] args) {
    AutoresService service = null;

    if (ehAssincrono) {
        service = new AutoresService_Service()
            .getAutoresServiceSOAP(new AddressingFeature());
        List<Handler> handlerChain = ((BindingProvider) service)
            .getBinding().getHandlerChain();

        handlerChain.add(new AddressingHandler(enderecoResposta));
        ((BindingProvider) service).getBinding()
            .setHandlerChain(handlerChain);
    } else {

```

```
        service = new AutoresService_Service().getAutoresServiceSOAP();
    }

    List<Autor> autores = service.solicitarRelacaoDeAutores(null);

    for (Autor autor : autores) {
        System.out.println(autor.getNome());
    }
}
```

Dessa forma, ao submeter esta requisição, ajustando o valor de `ehAssincrono` para `true`, nada será impresso pelo cliente — a resposta deverá aparecer no console do JBoss. Se o valor for `false`, o valor dos autores do sistema deverá aparecer no console do cliente. Seu serviço foi habilitado para tratamento de requisições assíncronas.

7.5 SUMÁRIO

Neste capítulo, você viu alguns dos padrões de projeto mais comuns em SOA. Você descobriu alguns dos benefícios do desenvolvimento orientado por contratos (ou seja, a técnica conhecida como `contract-first`), assim como os benefícios de se ter um modelo padronizado para tratamento de suas entidades de negócio — o famoso **Modelo Canônico**.

Finalmente, você conheceu uma técnica para tratamento assíncrono de requisições, através da especificação `WS-Addressing`, e viu como ela pode te ajudar ao realizar tratamento de grandes volumes de dados.

No entanto, muito ainda é desconhecido. Resta muito a ser visto — como englobar os principais padrões de integração, como coordenar corretamente a execução de vários serviços, como tratar as especificações `WS-*` de maneira mais eficiente etc. Para isso, você deverá conhecer duas das ferramentas mais famosas do mundo SOA: *ESB (Enterprise Service Bus)* e *BPEL (Business Process Execution Language)*.

CAPÍTULO 8

Flexibilizando sua aplicação com um ESB

“O futuro começa com uma simples iniciativa”

– Albert Einstein

Agora que você já conhece os mecanismos para expor suas aplicações de maneira orientada a serviços, uma dúvida vem à mente: como trabalhar com ambientes que não são compatíveis com o que já foi criado por você? Por exemplo, como interagir com um sistema de CRM proprietário, que expõe *web services* no próprio formato? Como tornar sua aplicação flexível o suficiente para incluir funcionalidades parecidas com as antigas, sem que haja necessidade de modificar os clientes para que eles enxerguem estas novas funcionalidades?

Estas e outras questões são endereçadas pelo ESB (*Enterprise Service Bus*). Trata-se da implementação de um padrão de projeto SOA (catalogado por Thomas Erl [4]) cujo propósito é abstrair as implementações de serviços dos clientes, conferindo a estes, certo grau de desacoplamento em relação à implementação do serviço. Isto é

feito através da capacidade do ESB de expor uma **interface de serviço** (ou seja, um contrato qualquer) e transformar a requisição do cliente para esta interface em algo que seja compatível com a implementação real do serviço.

Com estas transformações, o ESB consegue implementar todos (ou quase todos) os padrões de projeto de integração existentes (sendo o principal catálogo o livro de Gregor Hohpe e Bobby Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* [9], também disponível em <http://eapatterns.com/>), conferindo grande flexibilidade às aplicações orientadas a serviços.

Neste capítulo, você aprenderá a instalar e utilizar o Oracle Service Bus (OSB), um dos ESBs mais poderosos da atualidade.

8.1 COMO INSTALAR O ORACLE WEBLOGIC E O OEPE

O OSB é, na verdade, uma aplicação que roda sobre o *Application Server* da Oracle, o *WebLogic*. Portanto, antes de realizar a instalação do OSB, você deve instalar primeiro o *WebLogic*. Além disso, para interagir com o Oracle Service Bus, é recomendável que você instale, em conjunto com o *WebLogic*, o Oracle Enterprise Pack for Eclipse — OEPE. Trata-se de uma versão do Eclipse com vários *plugins* já habilitados para que você possa trabalhar corretamente com o *WebLogic* e outros produtos da Oracle, como o OSB.

Para realizar o download do *WebLogic* do OSB, você deve ir até a página de *download* do OSB: <http://www.oracle.com/technetwork/middleware/service-bus/downloads/index.html>. Lá, você encontrará diversas opções de instalação para o OSB e seus pré-requisitos. Para todos os procedimentos a seguir, serão utilizados os componentes da versão 11.1.1.6.0, em JVM 32-bits para Windows.

ASPECTOS LEGAIS DA INSTALAÇÃO DO OSB

Para instalar o OSB, você precisa ter usuário e senha na *Oracle Technology Network* (OTN) e aceitar os termos da licença. Tenha em mente de que a licença para desenvolvedores é gratuita, mas para quaisquer outros usos, não. Em caso de dúvidas, entre em contato com a Oracle.

A instalação do *WebLogic* e do OEPE é bem simples. Ao realizar o *download* do mesmo, basta dar um duplo-clique sobre o instalador, o que, após um breve período de carga, vai abrir sua tela de boas vindas:

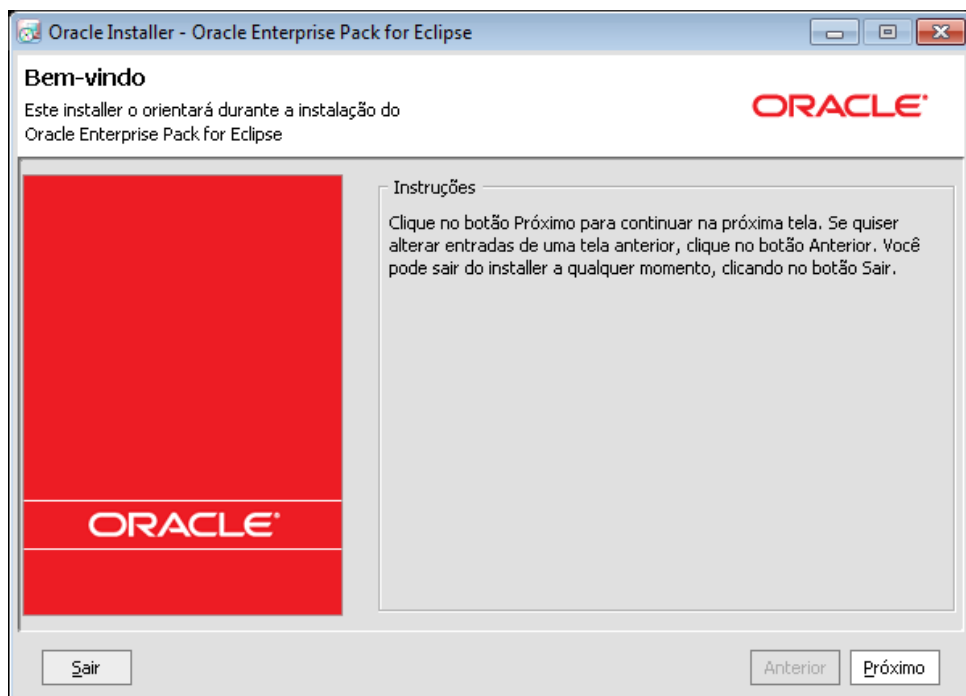


Figura 8.1: Tela de boas vindas da instalação do OEPE

Na sequência, será solicitada a seleção da pasta onde o *Middleware Home* estará localizado. Caso você já possua algum produto Oracle instalado, recomendo a criação de um *Middleware Home* diferente dos já existentes. Caso contrário, basta deixar como está:

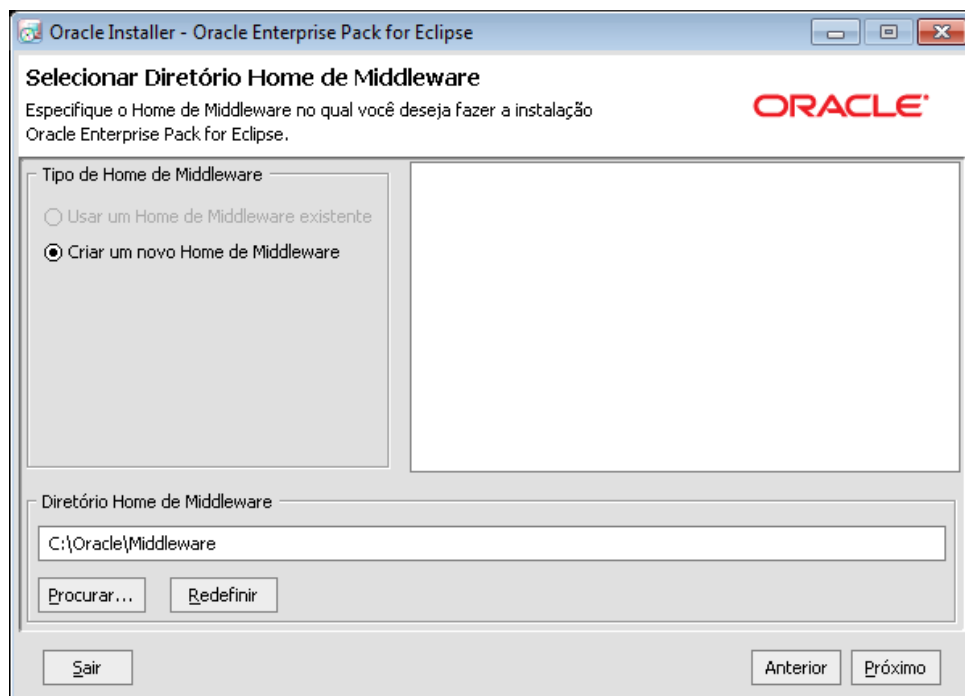


Figura 8.2: Tela de criação do Middleware Home

Na próxima tela, será solicitado um endereço de *e-mail* para que você possa receber mensagens sobre atualizações de segurança. Como desenvolvedor, não é necessário fornecer um endereço, e você pode desmarcar a seleção da caixa “Gostaria de receber atualizações de segurança através do My Oracle Support” (você deverá, além disso, marcar “Sim” na janela *pop-up* que irá aparecer na sequência):

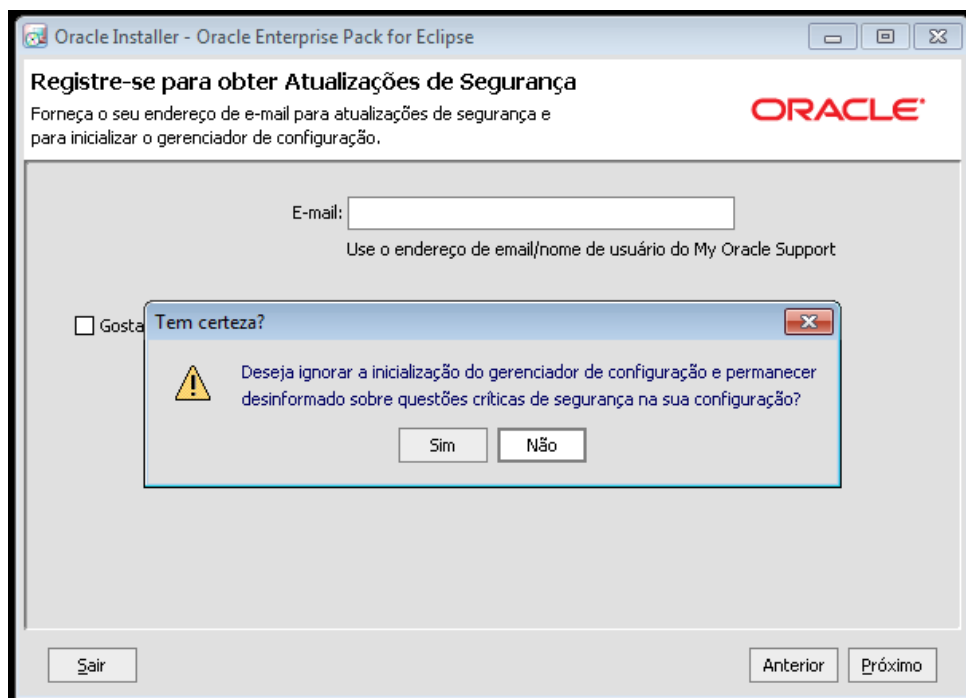


Figura 8.3: Tela de solicitação de recebimento de atualizações de segurança

A seguir, aparecerá uma tela onde você pode selecionar qual o tipo de instalação desejada — típica ou personalizada. Basta deixar marcada como típica e clicar em **Próximo**:

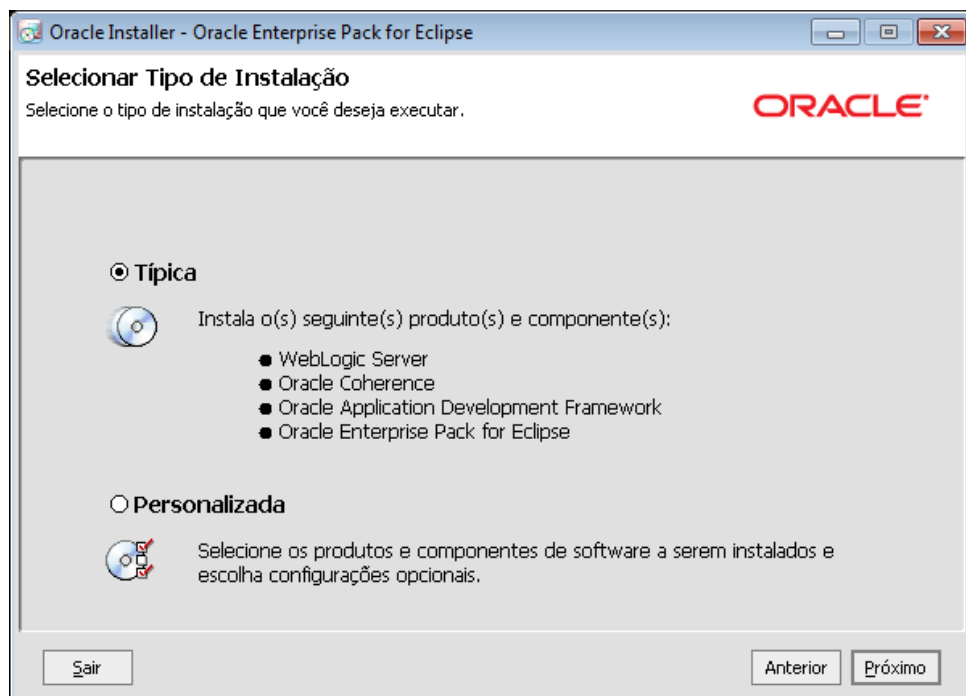


Figura 8.4: Tela de seleção de tipo de instalação

A próxima tela solicitará os diretórios nos quais se deseja instalar os produtos selecionados (WebLogic, Coherence e OEPE). Basta deixar como está e clicar em Próximo:

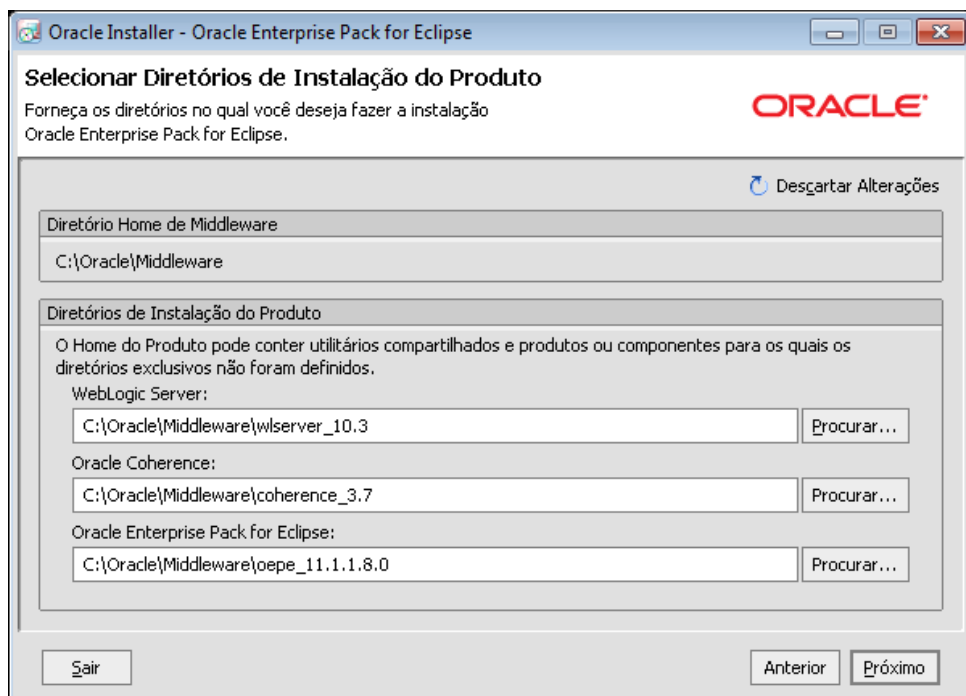


Figura 8.5: Tela de seleção de diretórios

As próximas telas solicitarão os locais para posicionamento dos atalhos (no menu Iniciar do Windows) e apresentarão um resumo do que será instalado. Basta clicar em **Próximo** nas duas telas para realizar a instalação. Ao final, a tela de conclusão deve ser apresentada, com uma caixa para solicitação de execução de *quickstart*. Desmarque esta caixa e clique em **Concluído**:

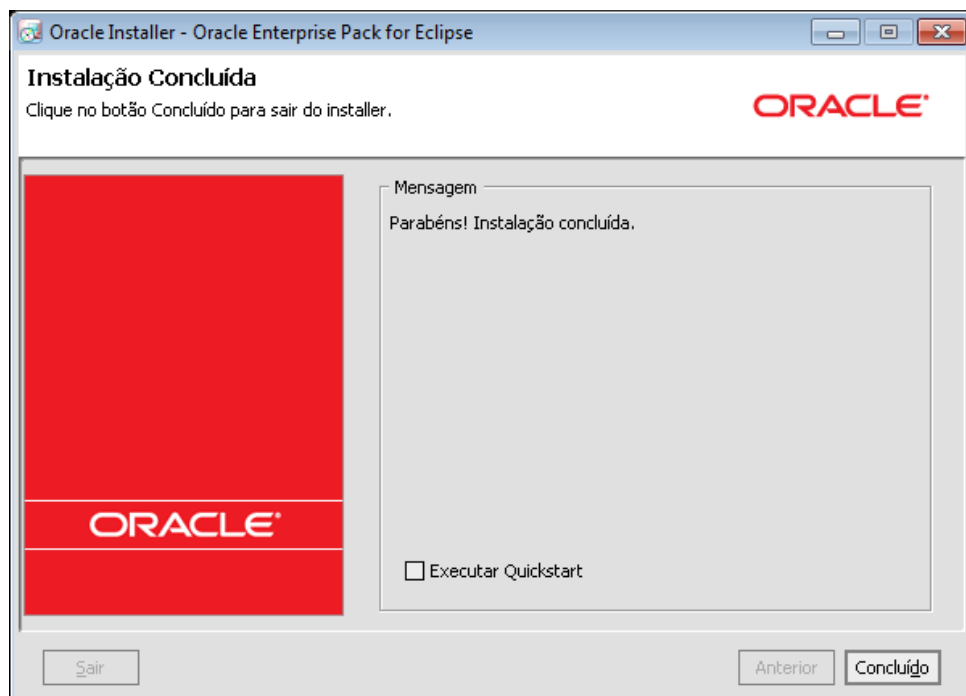


Figura 8.6: Conclusão da instalação

8.2 A INSTALAÇÃO DO OSB

Uma vez realizado este procedimento, o OEPE e o *WebLogic* estão instalados. O próximo passo é instalar o OSB. Realizado o seu *download*, descompacte-o em uma pasta de sua preferência e, a partir do aplicativo de linha de comando, execute a instalação fornecendo o caminho da sua JDK, através do argumento `-jreLoc` (atenção: instale sua JDK em um diretório que não contenha espaços no nome):

```
Disk1\setup.exe -jreLoc C:\Java\jdk1.7.0_09
```

A primeira tela será de boas-vindas. A segunda, será relativa às instalações de *software*. Novamente, como desenvolvedor, você não deve se preocupar com este detalhe, e pode ignorar estas atualizações:



Figura 8.7: Tela de solicitação de atualizações do OSB

Na sequência, será solicitado o caminho do *Middleware Home*. Caso nada esteja marcado, forneça o caminho do *Middleware Home* criado na instalação do OEPE e clique em **Próximo**:



Figura 8.8: Seleção do Middleware Home

A seguir, será questionado qual o tipo da instalação (típica ou personalizada) e uma checagem dos pré-requisitos para utilização do OSB será realizada. Caso esta verificação tenha sucesso, o seguinte deverá ser apresentado:



Figura 8.9: Verificação dos pré-requisitos

VERIFICAÇÃO DE PRÉ-REQUISITOS

Esta verificação de pré-requisitos é trivial em plataformas Windows. Em outras plataformas, costuma ser mais exigente — devido à preexistência de certas aplicações do Sistema Operacional, espaço *swap* etc. Caso você deseje instalar o OSB em plataforma Linux, observe com atenção os pré-requisitos para isto.

As próximas telas questionarão a localização do *WebLogic* e do OEPE, bem como apresentarão o resumo do que será instalado. Ao clicar em **Próximo** em ambas, a instalação do OSB terá início. Se tudo correr bem, a seguinte tela deverá ser apresentada:



Figura 8.10: Finalização da instalação do OSB

Ao clicar em **Próximo**, uma tela de finalização da instalação será mostrada. Basta clicar em **Finalizar**.

8.3 CONFIGURAÇÃO DO OSB

Agora, o próximo passo é configurar o OSB para funcionamento. Isto é feito através da criação de um novo domínio do *WebLogic*, habilitado para execução do OSB. Antes de fazer isso, no entanto, é necessário ter um banco de dados instalado (preferencialmente, Oracle) e executar uma ferramenta de configuração chamada *Repository Creation Utility* — RCU. Esta ferramenta pode ser obtida na página de *downloads* do Oracle SOA Suite - <http://www.oracle.com/technetwork/middleware/soasuite/downloads/index.html>.

Ao realizar o *download* do RCU, descompacte-o em uma pasta de sua preferência

e, na sequência, acesse a pasta `BIN` e execute o arquivo `rcu.bat` (em ambiente Linux, apenas `rcu`). Após a tela de boas vindas, uma tela de seleção de criação ou remoção de esquemas aparecerá. Deixe `Criar` selecionado e clique em `Próximo`. A próxima tela questionará a respeito das informações do banco. Assumindo um banco na própria máquina, do tipo `Oracle XE`, você deverá manter configurações semelhantes às seguintes:



Figura 8.11: Tela de entrada de configurações de banco de dados

Note que é necessário fornecer um usuário com privilégios `DBA` ou `SYSDBA`. Para um banco de dados Oracle XE, por exemplo, este usuário é `SYS`.

Na sequência, o RCU irá iniciar os testes para verificar a compatibilidade do banco de dados. Note que, para um banco de dados Oracle XE, um alerta de compatibilidade é emitido.



Figura 8.12: Alerta de compatibilidade com o Oracle XE

Este alerta ocorre porque o Oracle XE não é totalmente compatível para uma instalação de produção, mas é aceitável para uma máquina de desenvolvimento.

O próxima etapa é selecionar o tipo de preparação do banco de dados que deverá ser realizado. Para o OSB, você deve marcar a opção `Infraestrutura SOA`:

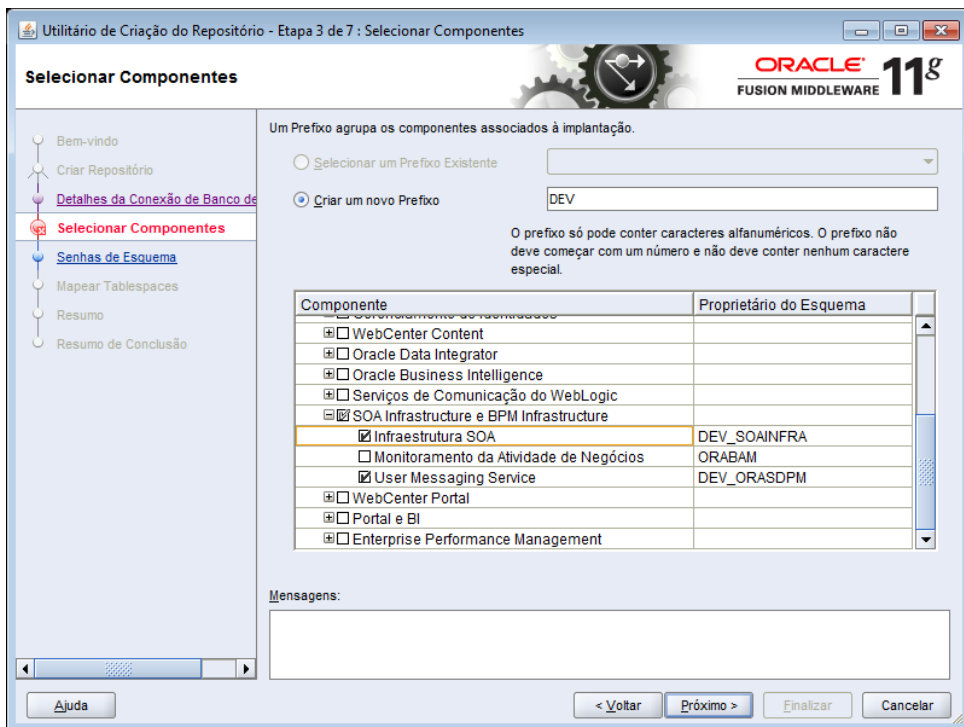


Figura 8.13: Seleção de configuração de banco de dados

Em seguida, podemos realizar a definição das senhas dos esquemas (no Oracle XE, é criado um usuário por esquema). Sugiro manter a mesma senha para todos os esquemas, conforme a imagem:



Figura 8.14: Definição das senhas dos esquemas

Finalmente, basta seguir as instruções até o final para terminar a definição do formato do banco de dados.

Agora, você deve configurar o OSB. Para isso, basta executar o programa `config.cmd`, localizado na pasta `common\bin` do diretório do WebLogic. Assumindo que você manteve os diretórios padronizados até agora, este programa estará localizado na pasta `C:\Oracle\Middleware\wlserver_10.3\common\bin`. Feito isso, o configurador irá solicitar a criação de um novo domínio. Ao clicar em **Próximo**, a seguinte tela deverá aparecer:

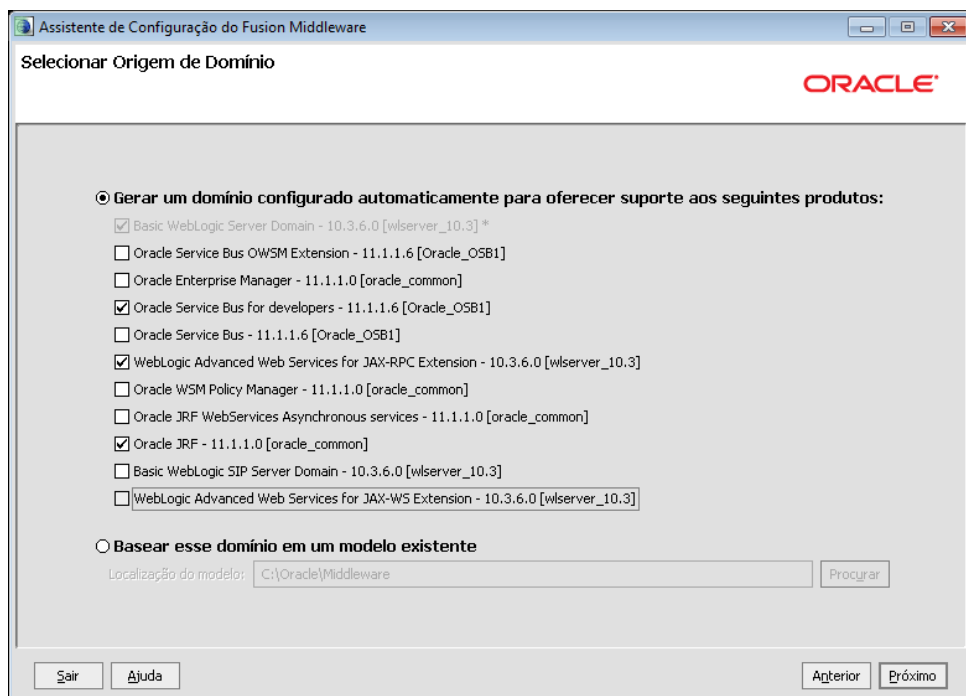


Figura 8.15: Criação do domínio no WebLogic

Para uma máquina de desenvolvimento, basta assinalar a opção `Oracle Service Bus for developers` (as dependências deste serão automaticamente assinaladas).

A seguir, você deve fornecer um nome para o domínio. Sugiro criar um domínio com nome `osb_domain`:

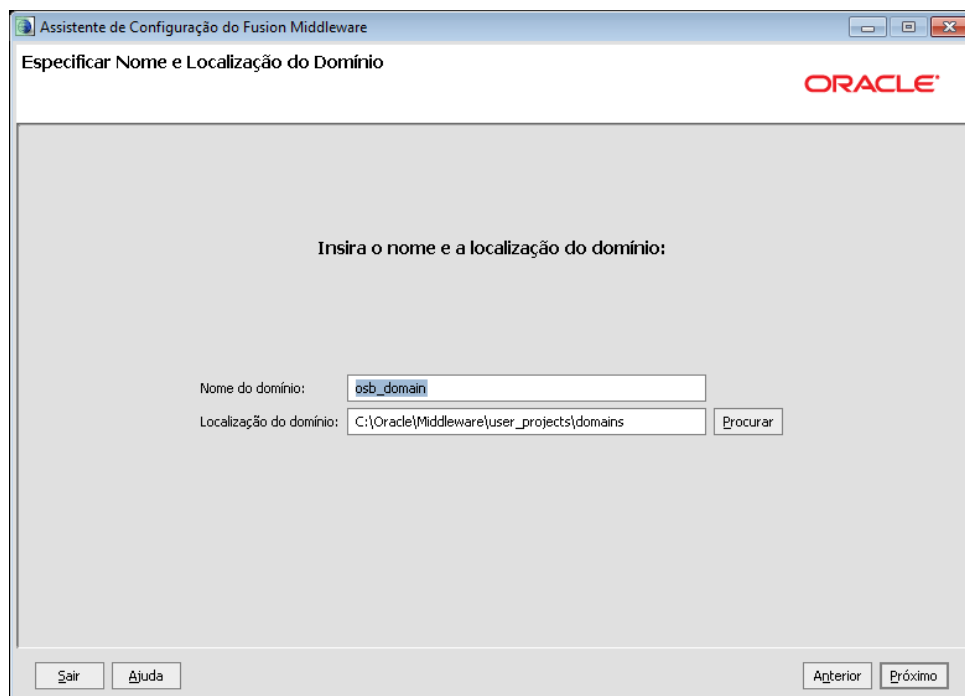


Figura 8.16: Nomeando o domínio

Na sequência, você deve criar uma senha para o usuário `weblogic`. Esta senha deve ser alfanumérica, ou seja, conter pelo menos um número e uma letra.

O próximo passo é selecionar o modo de inicialização do `WebLogic` e a `JDK` de operação. Novamente, para máquinas de desenvolvimento, é recomendado deixar as opções padrão, isto é, manter em modo de desenvolvimento e com a `JDK Sun`:

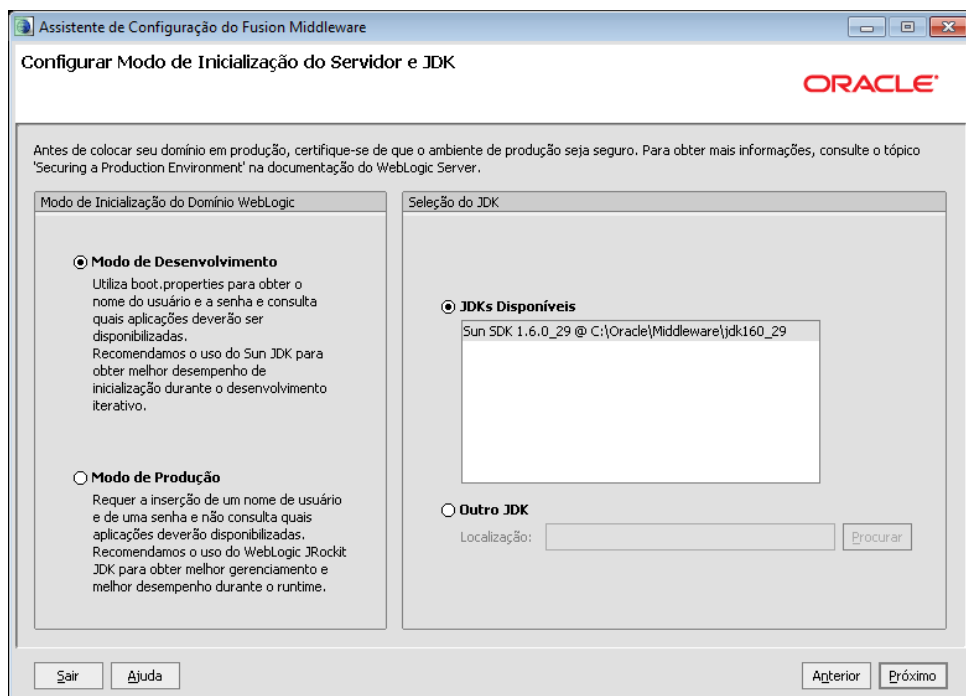


Figura 8.17: Seleção do modo de inicialização e JDK

A próxima tela pede as informações de banco de dados. O esquema solicitado é DEV_SOAINFRA, ou seja, você deve fornecer as informações que sejam compatíveis com aquelas utilizadas na definição deste esquema:

Assistente de Configuração do Fusion Middleware

Configurar Esquema do Componente JDBC

ORACLE

Observação: Altere somente os campos de entrada a seguir que deseja modificar e os valores serão aplicados a todas as linhas selecionadas.

Fornecedor: Oracle DBMS/Serviço: XE

Driver: *Oracle's Driver (Thin) for Instance connections; Versions: 9.0.1 and | Nome do Host: localhost

Proprietário do Esquema: DEV_SOAINFRA Porta: 1521

Senha de Esquema: *****

RAC configuration for component schemas:

☐ Convert to GridLink ☐ Convert to RAC multi data source ☐ Don't convert

	Esquema do Componente	DBMS/Serviço	Nome do Host	Porta	Proprietário do Esq...	Senha de Esquema
<input checked="" type="checkbox"/>	OSB JMS Reporting Provider	XE	localhost	1521	DEV_SOAINFRA	*****

Sair Ajuda Anterior Próximo

Figura 8.18: Formulário de informações do banco de dados

A tela seguinte realizará um teste de conectividade com o banco de dados (é importante que seu banco esteja plenamente operacional neste momento). Caso o teste tenha sucesso, você deverá ver uma tela como a que segue:

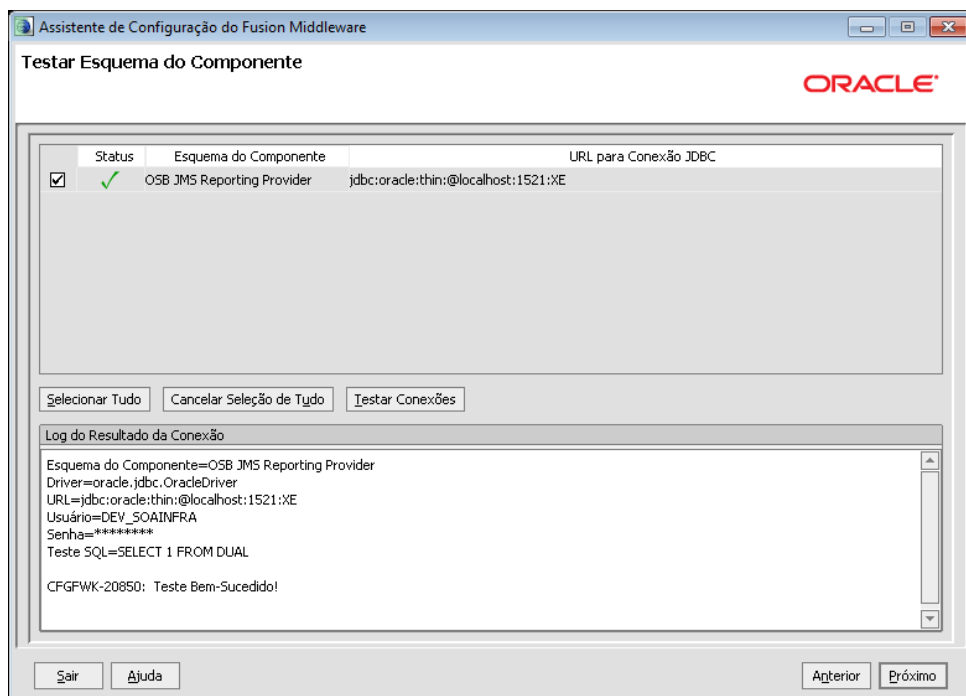


Figura 8.19: Teste de conectividade do banco de dados

Finalmente, basta seguir o instalador até o final. Caso tudo tenha sucesso, você deverá ver uma tela como a próxima:

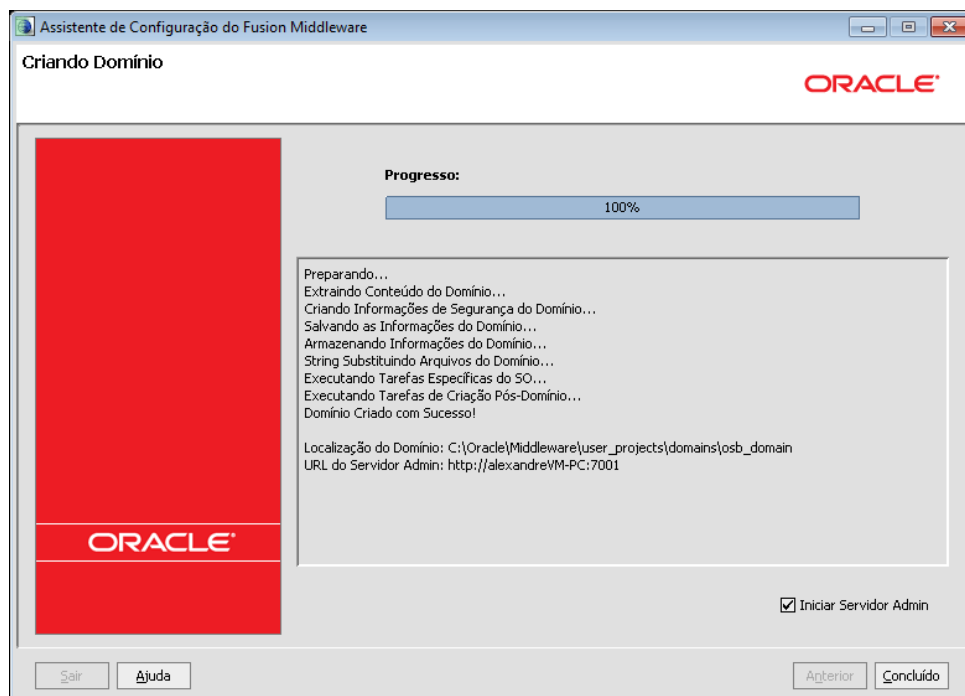
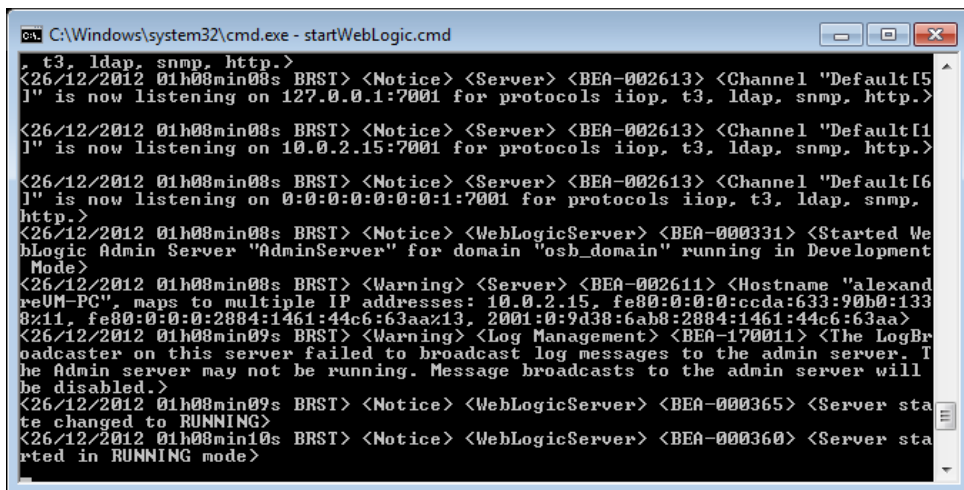


Figura 8.20: Tela de finalização da criação do domínio

Se você deixar a caixa `Iniciar Servidor Admin` marcada, a inicialização do servidor será realizada. Quando você realizar a inicialização posteriormente, basta acessar o domínio do OSB e executar o *script* `startWebLogic.bat`. Se você tiver seguido todos os passos passados até aqui à risca, este diretório é `C:\Oracle\Middleware\user_projects\domains\osb_domain`.

Ao final da inicialização, você deve enxergar uma tela semelhante à seguinte:



```
C:\Windows\system32\cmd.exe - startWebLogic.cmd
t3, ldap, snmp, http.>
<26/12/2012 01h08min08s BRST> <Notice> <Server> <BEA-002613> <Channel "Default[5
1]" is now listening on 127.0.0.1:7001 for protocols iiop, t3, ldap, snmp, http.>
<26/12/2012 01h08min08s BRST> <Notice> <Server> <BEA-002613> <Channel "Default[1
1]" is now listening on 10.0.2.15:7001 for protocols iiop, t3, ldap, snmp, http.>
<26/12/2012 01h08min08s BRST> <Notice> <Server> <BEA-002613> <Channel "Default[6
1]" is now listening on 0:0:0:0:0:0:1:7001 for protocols iiop, t3, ldap, snmp,
http.>
<26/12/2012 01h08min08s BRST> <Notice> <WebLogicServer> <BEA-000331> <Started We
bLogic Admin Server "AdminServer" for domain "osb_domain" running in Development
Mode>
<26/12/2012 01h08min08s BRST> <Warning> <Server> <BEA-002611> <Hostname "alexand
reUM-PC", maps to multiple IP addresses: 10.0.2.15, fe80:0:0:0:ccda:633:90b0:133
8x11, fe80:0:0:0:2884:1461:44c6:63aa:13, 2001:0:9d38:6ab8:2884:1461:44c6:63aa>
<26/12/2012 01h08min09s BRST> <Warning> <Log Management> <BEA-170011> <The LogBr
oadcaster on this server failed to broadcast log messages to the admin server. Th
e Admin server may not be running. Message broadcasts to the admin server will
be disabled.>
<26/12/2012 01h08min09s BRST> <Notice> <WebLogicServer> <BEA-000365> <Server sta
te changed to RUNNING>
<26/12/2012 01h08min10s BRST> <Notice> <WebLogicServer> <BEA-000360> <Server sta
rted in RUNNING mode>
```

Figura 8.21: Inicialização do servidor OSB

Note a presença da palavra **RUNNING**. Este é o indicador de que seu servidor WebLogic está operante. Para visualizar o console do OSB, basta acessar a URL <http://localhost:7001/sbconsole> e preencher o formulário de autenticação com o usuário `weblogic` e a senha fornecida durante a instalação.

8.4 CONCEITOS DO OSB

O OSB centraliza em si os artefatos utilizados pela comunicação, como WSDL's e XSD Schemas. A comunicação feita com cada serviço é realizada através de *Business Services* (ou “Serviços de Negócio”, na versão em português). A definição dos *Business Services* carrega diversas configurações além da especificação do *web service* em si, como o sistema de segurança utilizado, qual a codificação da comunicação, mecanismos para balanceamento de carga etc.

Já os mecanismos de roteamento, transformação de dados, enriquecimento de mensagens e outras técnicas são incorporados pelos *Proxy Services* (ou *Serviços de Proxy*, na versão em português). Um *Proxy Service* recebe a requisição e pode efetuar diversos tipos de filtragem, transformação e enriquecimento até finalmente entregá-la a um (ou mais) *Business Services*. Além disso, um *Proxy Service* pode ser criado diretamente a partir de um *Business Service* sem complicações. Vale lembrar: o ponto de acesso aos serviços deve ser sempre feito via *Proxy Service*, e nunca via

Business Service.

8.5 CRIE UMA ROTA NO OSB

Para os próximos passos, você deve inicializar o OEPE. Caso ele não esteja disponível no menu Iniciar do Windows, basta acessar a *Middleware Home* e localizar uma pasta chamada `oepe_<versão>`. Nesta pasta, está contido o programa `eclipse.exe`, que contém o pacote do OEPE.

Uma vez aberto, é necessário verificar se o `WebLogic` que contém o OSB está presente. Caso não esteja, basta adicioná-lo da mesma forma como você adicionaria outro servidor qualquer.

Em seguida, você precisa criar dois projetos. O primeiro, para conter as especificações da configuração geral do seu projeto (que recebe o nome de `Oracle Service Bus Configuration Project`). O segundo, para conter artefatos específicos (que recebe o nome de `Oracle Service Bus Project`). Para demonstrar o uso destes, criei um `Oracle Service Bus Configuration Project` com o nome de `KnightELibrary` e um `Oracle Service Bus Project` com o nome de `Estoque`. Este último deve ser criado sempre associado a um `Oracle Service Bus Configuration Project`.

CRIANDO OS PROJETOS DO ORACLE SERVICE BUS

A criação dos projetos `Oracle Service Bus` no OEPE é trivial; basta proceder da mesma maneira como você faria em outros projetos. Tenha em mente, no entanto, que melhores resultados serão atingidos ao ajustar a perspectiva para `Oracle Service Bus`.

Uma vez criados os projetos, crie duas pastas sob o projeto `Estoque`: `artefatosServicos` e `artefatosOSB`. A primeira conterá o WSDL de `AutoresService` e os `XML Schemas`. A segunda conterá o *Business Service* que irá referenciar o serviço de autores e o *Proxy Service* que fornecerá o acesso a este.

O projeto que contém os artefatos e uma implementação via `Endpoint.publish` está disponível no github, sob nome <https://github.com/alesaudate/soa/tree/master/soa-capo8>. Lá, basta copiar o WSDL e os `Schemas` para a pasta `artefatosServicos`.

Feito isso, clique com o botão direito sobre a pasta `artefatosOSB` e aponte para `New -> Business Service`. Isso abrirá a caixa de diálogo para a criação de um novo *Business Service*, que você pode dar o nome `BSAutoresService`:

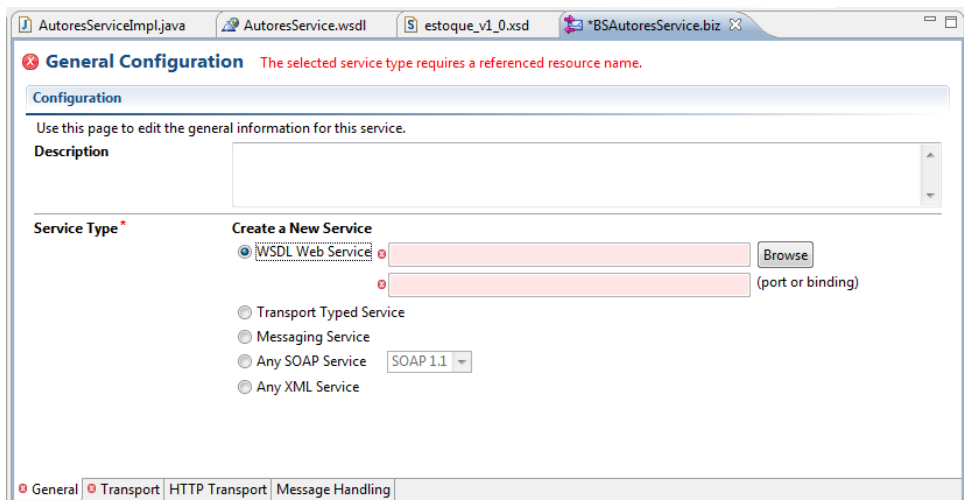


Figura 8.22: Tela para criação do Business Service de Autores

Selecione a opção `WSDL Web Service` e clique em `Browse`. Feito isso, selecione o arquivo `AutoresService.wsdl` e o expanda para utilizar o *binding* `AutoresServiceSOAP`. Clique em `OK`.

A seguir, você deve inserir o *endpoint* no qual o seu serviço irá responder. Para isso, mude a seleção para a aba `Transport` e, na seção `Endpoint URI`, coloque o endereço `http://localhost:9090/autoresService`, clique em `Add` e depois salve. Seu *Business Service* está pronto.

O próximo passo é criar o *Proxy Service* que fará a conexão com o *Business Service*. Para isso, clique novamente sobre o projeto `Estoque` e crie um *Proxy Service*, dessa vez, chamado `PSAutoresService`. Da mesma forma como você fez em relação ao *Business Service*, crie o *Proxy Service* baseado em um WSDL — o mesmo WSDL utilizado no *Business Service*. Feito isso, você deve mudar para a aba `Messaging Flow`, onde você deve configurar as opções de roteamento utilizadas pelo OSB:

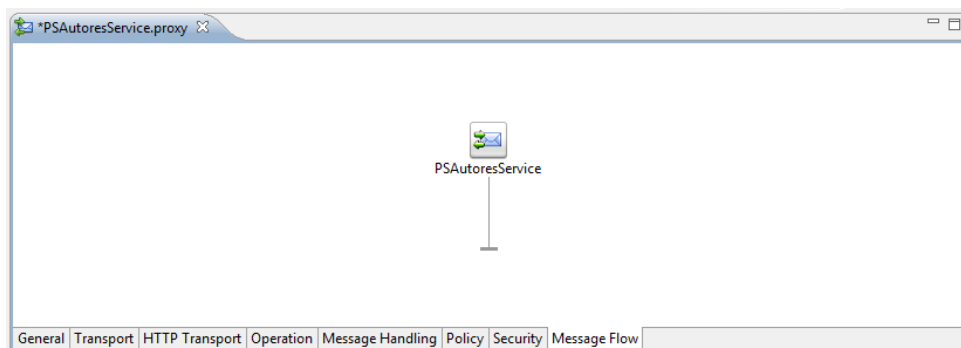


Figura 8.23: Tela para configuração do Proxy Service no OEPE

Agora, clique com o botão direito sobre o ícone do envelope e aponte para `Insert Into -> Route`. A seguir, clique com o botão direito sobre `Insert Into -> Communication -> Routing`. Isso criará uma caixa dentro da caixa `RouteNode1`. Clique nesta caixa interna e selecione a aba `Properties`, mais abaixo. No campo `Service`, clique em `Browse` e selecione o *Business Service* que você criou há pouco.

Neste momento, sua configuração está pronta. Para testá-la, clique com o botão direito sobre a configuração, aponte para `Export -> Oracle Service Bus - Resources to Server`. Preserve todas as configurações como estiverem. Caso esteja tudo configurado corretamente, você deve ser capaz de ver o projeto no console do OSB, disponível em <http://localhost:7001/sbconsole>.

Para realizar o teste, basta expandir a aba `Resource Browser` e, a seguir, clicar em `Serviços de Proxy`. O *Proxy Service* `PSAutoreService` deverá aparecer na listagem. Basta clicar sobre o pequeno inseto (*bug*) à direita do mesmo para ser direcionado para a tela de teste do serviço.

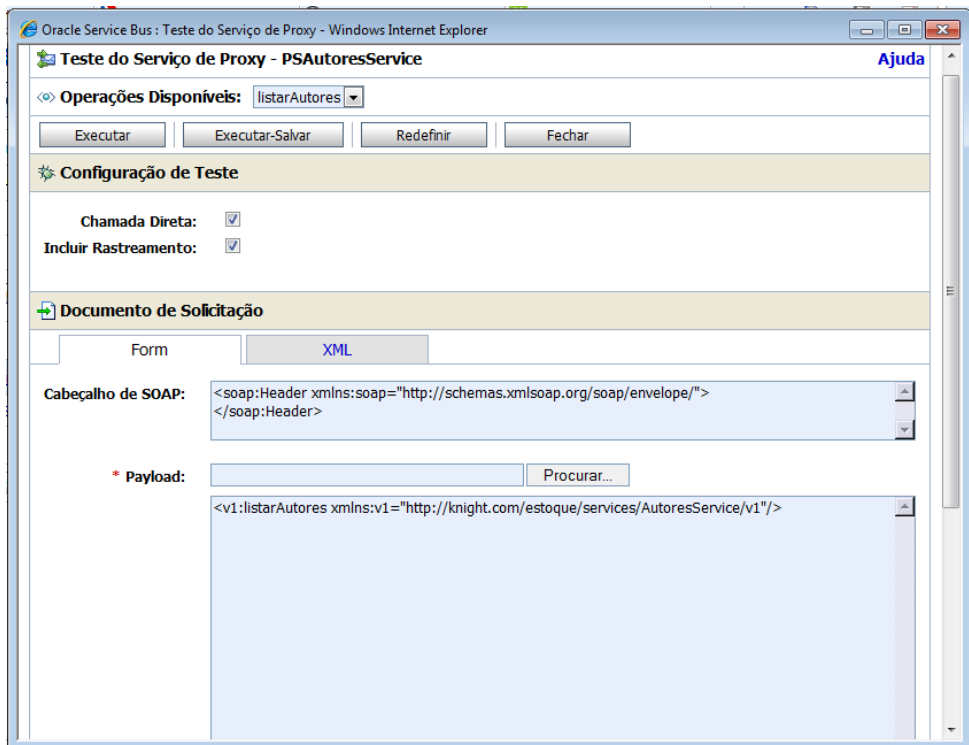


Figura 8.24: Tela de teste do proxy service

Finalmente, basta selecionar a operação desejada e modificar o *payload* (ou seja, o corpo da requisição) à vontade. Para enviar a chamada, basta clicar em **Executar**. Caso tudo esteja bem, algo como o seguinte deve ser apresentado na tela **Documento de Resposta** (lembre-se de realizar o teste com o serviço de autores inicializado):

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  <soapenv:Header/>
  <S:Body xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
    <ns2:listarAutoresResponse
      xmlns:ns2=
        "http://knight.com/estoque/services/AutoresService/v1">
      <autor>
        <nome>Alexandre</nome>
      </autor>
    </ns2:listarAutoresResponse>
```

```
</S:Body>  
</soapenv:Envelope>
```

Caso você deseje criar o cliente Java, basta utilizar WSDL disponibilizado pelo OSB em <http://localhost:7001/Estoque/artefatosOSB/PSAutoresService?wsdl>.

8.6 SUMÁRIO

Neste capítulo, você aprendeu a instalar, configurar e utilizar o Oracle Enterprise Pack for Eclipse e o Oracle Service Bus. Você aprendeu que esse tipo de ferramenta é utilizada para rotear mensagens, transformá-las e outros aspectos de *patterns* de integração.

Resta ainda ver como reunir vários serviços em uma coisa só, ou seja, expor vários serviços de maneira coordenada, de forma que sua implementação SOA seja ainda mais flexível.

CAPÍTULO 9

Coordene serviços com BPEL

“Não está ocioso apenas aquele que não faz nada, mas também aquele que poderia fazer algo melhor.”

– Sócrates

Você já tem as respostas para as tradicionais questões que podem surgir durante a implementação de uma arquitetura orientada a serviços. O último desafio que se faz presente é: como criar, de maneira adequada, composições de serviços? Em outras palavras, como endereçar desafios como transacionalidade e heterogeneidade na comunicação entre vários serviços de uma maneira simples, adaptável a mudanças?

Obviamente, existe mais de um modo de realizar comunicação entre vários serviços. Você vai conhecer os mecanismos mais comuns e como fazer isso de uma forma inteligente e rápida.

9.1 CONHEÇA ORQUESTRAÇÃO E COREOGRAFIA

As composições de serviços são comumente descritas como **orquestrações** ou **coreografias**. Uma coreografia é uma composição de serviços descentralizada. Na coreografia, cada participante (*web service*) deve saber especificamente quais ações executar. Já na orquestração existe um controlador centralizado, dizendo a cada *web service* a maneira de prosseguir a troca de dados.

Na prática, isto se traduz entre realizar estas trocas de informações com código puro (ou seja, expôr um *web service* da maneira tradicional, com código) ou fazer isto utilizando uma ferramenta exposta num servidor próprio para isso.

Ambas as abordagens possuem vantagens e desvantagens. Uma das vantagens da forma descentralizada é que esta é mais escalável, posto que você pode fornecer mais máquinas para atender as requisições. Além disso, o controle dos formatos de dados não está necessariamente preso a XML — por exemplo, se você tiver vários EJBs expostos como *web services* e quiser coordenar requisições entre eles, você pode expôr um terceiro EJB como *web service* e fazer a comunicação entre seus serviços de maneira direta, com código Java.

Isso facilita o estabelecimento de transações entre eles, de maneira que é desnecessário utilizar mecanismos mais complexos para isso. No entanto, isso se torna uma desvantagem do ponto de vista do acoplamento — se você quiser modificar algum desses serviços, os outros possivelmente sofreram impactos.

Já na orquestração, ao manter tudo de maneira centralizada, toda a comunicação é feita obrigatoriamente através da interface pública destes serviços, isto é, como um cliente comum do *web service*. Isso estimula o desenvolvimento voltado ao modo como os clientes vão utilizar o serviços. Além disso, também facilita a manutenção dos serviços em estado consistente, já que um nó centralizado consegue detectar falhas nos outros e colocar todos em estado consistente.

Entre as duas formas, a que tem sido mais adotada pelo mercado é a orquestração. Ela é implementada através do BPEL (*Business Process Execution Language*), que é uma linguagem especificamente criada com o propósito de coordenar a comunicação entre vários serviços. Dentre as implementações de BPEL, uma das versões mais utilizadas é o Oracle BPEL, que você irá conhecer neste capítulo.

9.2 INSTALE O ORACLE SOA SUITE

O Oracle BPEL é formado de um conjunto de aplicações, que são parte de um produto chamado `SOA Suite`. Este produto é instalado de maneira semelhante ao

Oracle Service Bus, ou seja, basta executar o RCU e instalar o SOA Suite sobre o WebLogic.

Existem várias maneiras de instalar o SOA Suite levando em conta nossa instalação prévia do OSB. Como estamos falando de uma máquina de desenvolvimento, você aprenderá a instalar o OSB e o SOA Suite juntos.

Sua primeira tarefa deve ser criar os bancos de dados de trabalho do SOA Suite. Para isso, você deve executar o RCU novamente — desta vez, marcando a opção SOA Infrastructure e BPM Infrastructure e criando um prefixo novo para o conjunto, como mostra a figura:



Figura 9.1: Criando as tabelas necessárias para o funcionamento do SOA Suite

A seguir, você deve instalar o SOA Suite, propriamente dito. A instalação é semelhante à do OSB: basta realizar o *download* dos arquivos referentes ao mesmo e, uma vez concluídos, descompactá-los em uma pasta de sua preferência. A seguir, inicie a aplicação de execução de linhas de comando do Windows (o `cmd`), navegue até a pasta `ofm_soa_generic_11.1.1.6.0_disk1_1of2/Disk1` e execute

a aplicação `setup.exe`.

Tome o cuidado de fornecer o parâmetro `-jreLoc`, com a localização de sua JDK/JRE (tome o cuidado, também, de tê-la instalada em uma pasta sem espaços no nome).

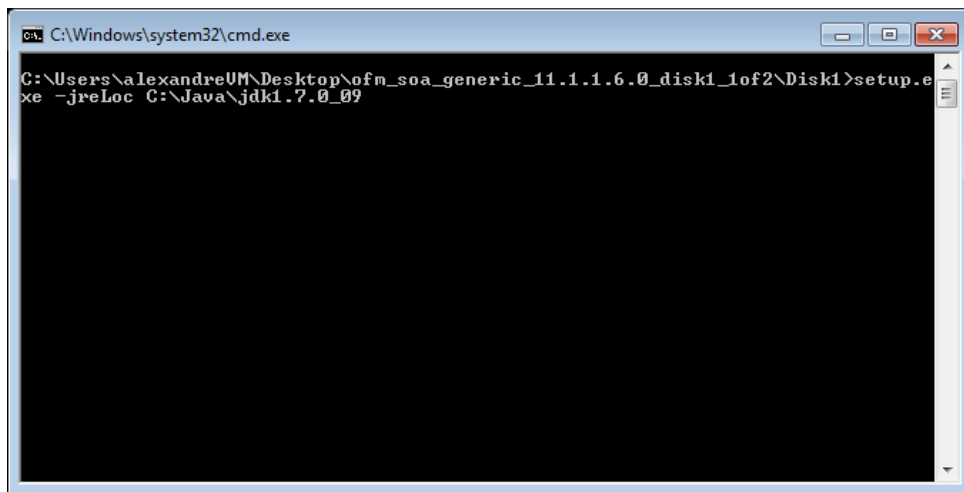


Figura 9.2: Inicializando a instalação do SOA Suite

A instalação do `SOA Suite` é bem simples, diferenciando pouco da instalação do `OSB`. Uma das diferenças será o questionamento de qual servidor de aplicação deverá ser utilizado (no nosso caso, o `WebLogic`):



Figura 9.3: Questionamento sobre qual Application Server utilizar

Outra diferença é que, em certo ponto da instalação (ou seja, com a transferência de arquivos já inicializada), o instalador deverá solicitar a localização dos discos 4 e 5. Quando isso acontecer, basta navegar até as pastas `ofm_soa_generic_11.1.1.6.0_disk1_2of2/Disk4` e `ofm_soa_generic_11.1.1.6.0_disk1_2of2/Disk5`, respectivamente:

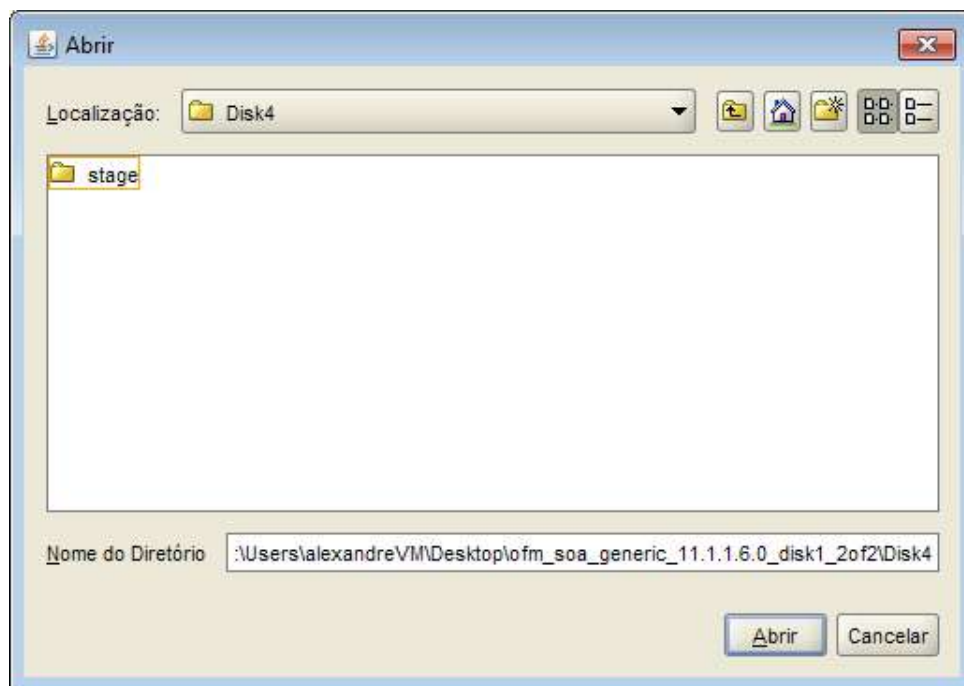


Figura 9.4: Selecionando o disco de instalação correto

Uma vez finalizado o procedimento para instalação, você deve realizar a configuração. Isso é feito utilizando o mesmo aplicativo que utilizamos para configurar o OSB, ou seja, o Configuration Wizard do WebLogic. O Configuration Wizard pode ser encontrado, mais uma vez, em `MIDDLEWARE_HOME\wlserver_10.3\common\bin\config.cmd`. A diferença entre nossa configuração anterior e a atual é que, ao invés de criarmos um novo domínio, vamos estender o já existente, que contém o OSB. Para fazer isso, selecionamos a opção **Ampliar um domínio WebLogic existente** na tela inicial do Configuration Wizard

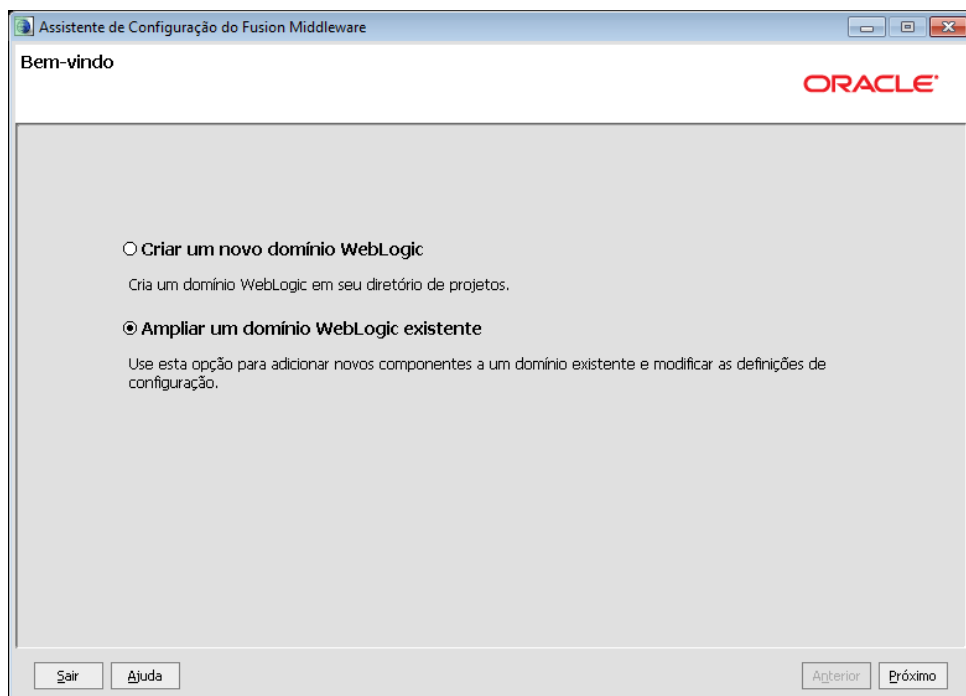


Figura 9.5: Utilizando o Configuration Wizard para ampliar um domínio do WebLogic

Uma vez selecionada esta opção, o Configuration Wizard irá questionar a respeito do domínio a ser ampliado. Basta selecionar a pasta onde o domínio está:

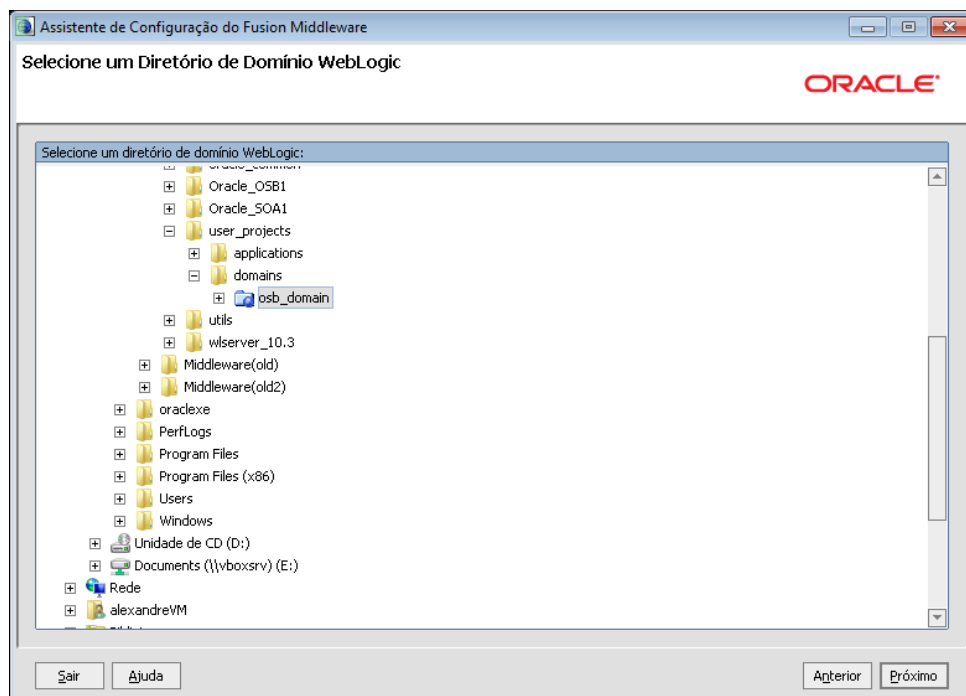


Figura 9.6: Selecionando o domínio do WebLogic

Selecionado o domínio, o Configuration Wizard irá exibir o que já existe no domínio (na forma de botões esmaecidos) e oferecerá as opções de ampliação. Basta marcar as opções Oracle SOA Suite for developers e Oracle Enterprise Manager (os pré-requisitos para a instalação de ambos serão automaticamente selecionados também):

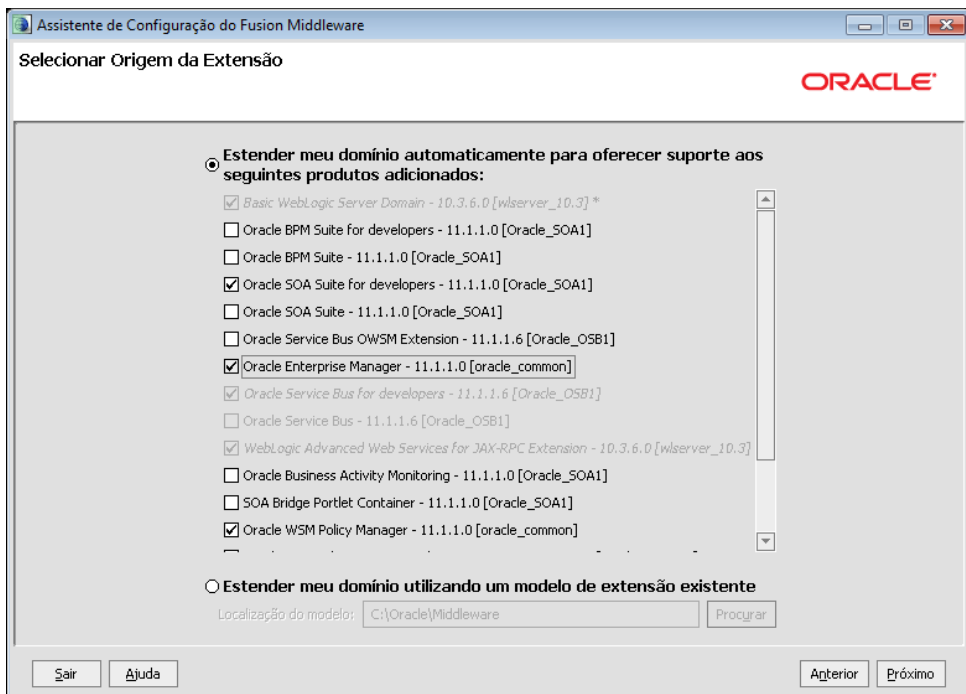


Figura 9.7: Configurando o SOA Suite no domínio do OSB

Ao prosseguir, o Configuration Wizard questionará a respeito de onde colocar as aplicações. Basta deixar a caixa de seleção como está:

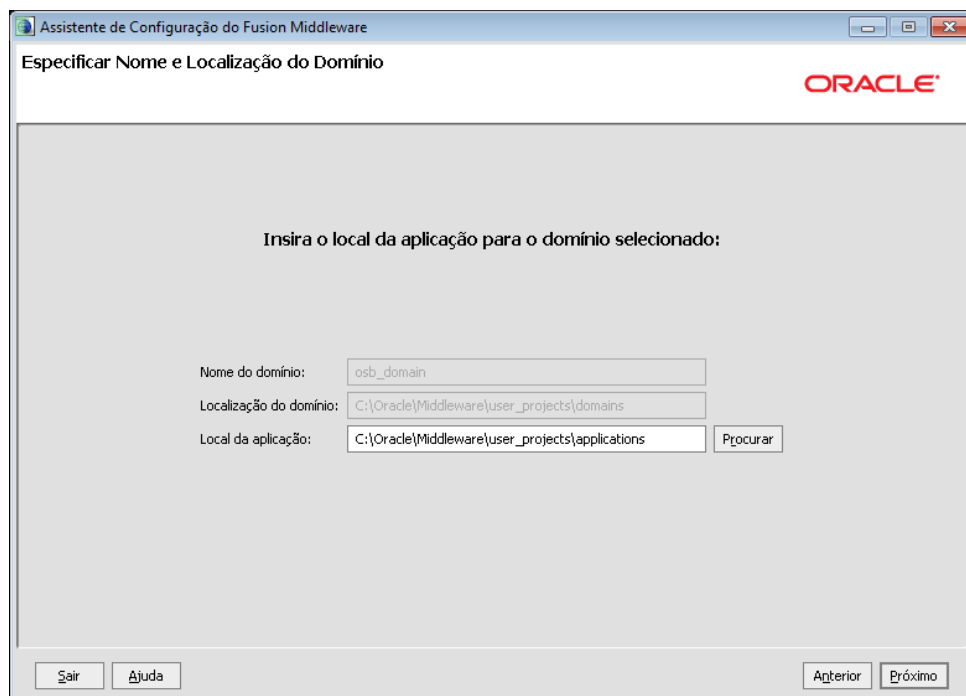


Figura 9.8: Apontando a pasta onde as aplicações do SOA Suite deverão ficar

Depois, o Configuration Wizard questionará a respeito das informações de banco de dados. Uma sutileza estará presente: as informações de proprietários de esquemas (ou *schema owners*, na versão em inglês) estará configurada com o prefixo `DEV`. Você deve selecionar uma a uma das configurações do SOA Suite para reconfigurar com o prefixo `DEV1` (tome cuidado para não reconfigurar também o componente OSB JMS Reporting Provider, que deve ser deixado como está). Uma vez realizado este procedimento, você pode marcar todos os componentes para configurar as informações de bancos de dados de uma vez só. A figura mostra como sua configuração deve ficar:

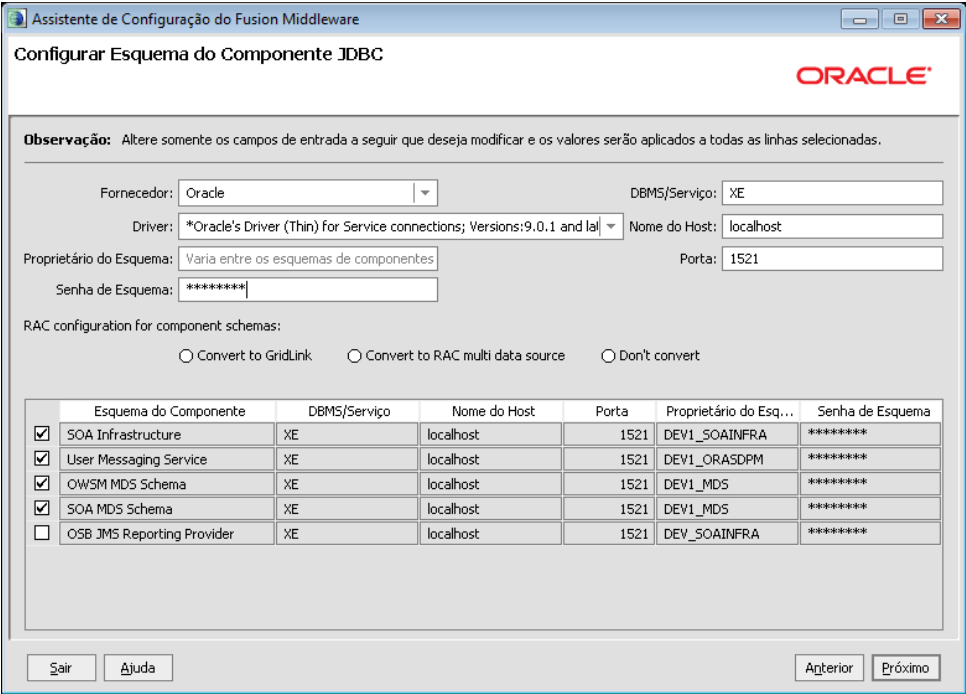


Figura 9.9: Configurando os esquemas de bancos de dados do SOA Suite

Aperte o botão `Selecionar Tudo` e, em seguida, `Testar Conexões`. O `Configuration Wizard` realizará o teste de todas as configurações de bancos de dados e, caso tudo esteja bem, o `status` de todos deve ficar verde:

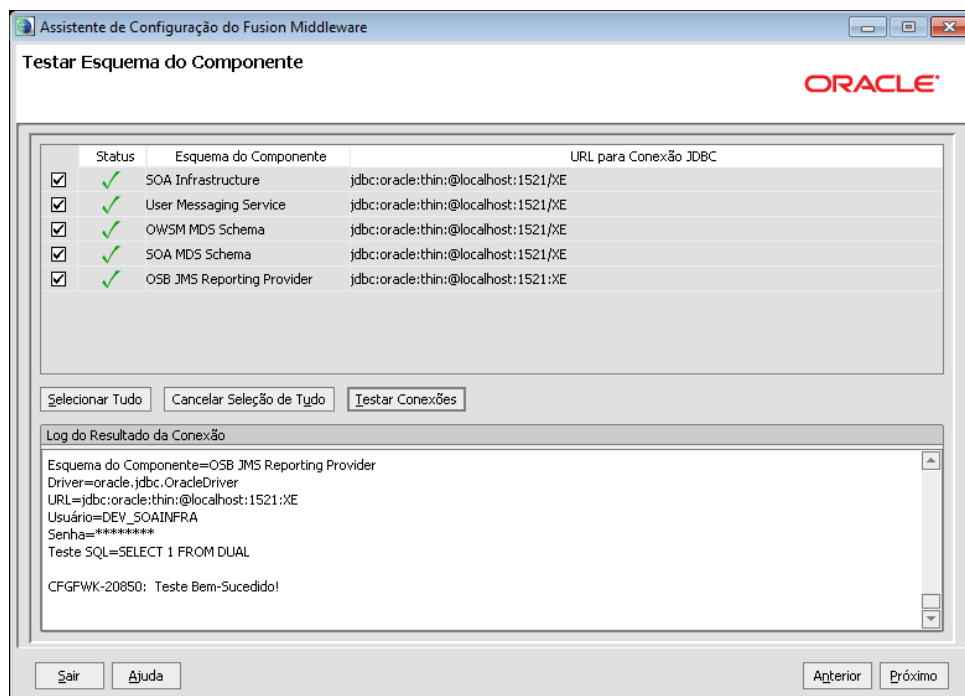


Figura 9.10: Status da configuração de componentes do SOA Suite

A partir deste ponto, basta prosseguir até o final. Seu `SOA Suite` está pronto.

9.3 INSTALE O JDEVELOPER

A IDE utilizada para desenvolvimento com o `Oracle SOA Suite` é o `JDeveloper`. Na página de *download* do próprio `SOA Suite`, você irá encontrar o arquivo do `JDeveloper`. Uma vez feito o *download*, execute o arquivo (este pode ser um `.exe`, se o *download* foi feito para Windows, ou `.jar` para plataforma genérica).

Após a extração do pacote, o procedimento para instalação será iniciado. Um dos pontos principais aos quais você deve estar atento é a seleção do `Middleware Home` para o `JDeveloper`. Note que este deve ser diferente daquele já reconhecido (ou seja, algo como `C:\Oracle\Middleware`), porque o `JDeveloper` já vem com um `WebLogic` próprio, o que pode causar a sobrescrita do que já está presente no seu `Middleware Home` antigo. Em suma, basta selecionar um novo (por exemplo,

C:\Oracle\IDE):

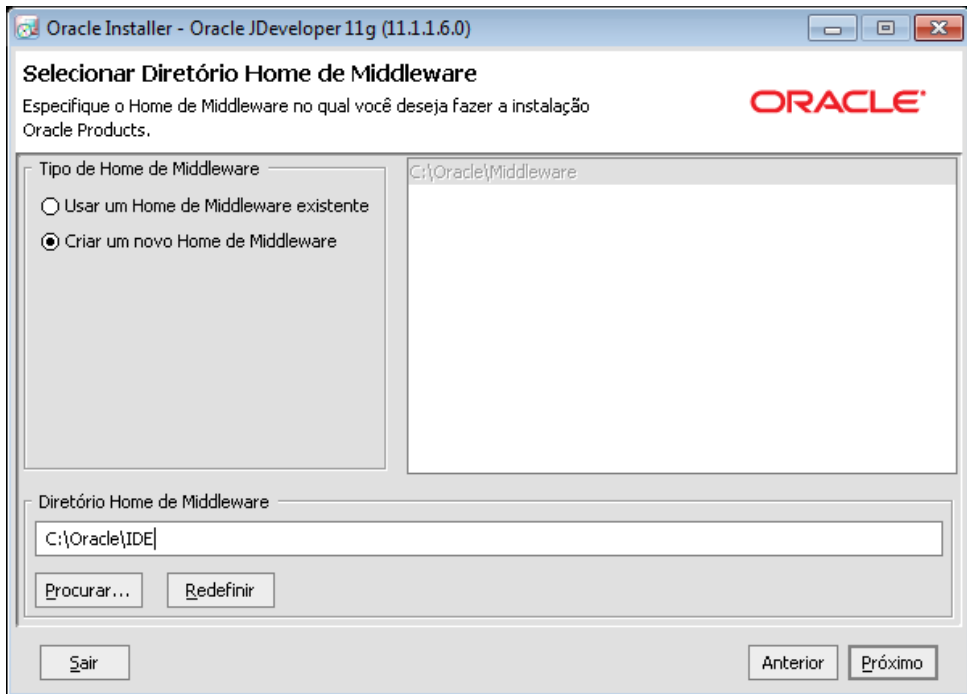


Figura 9.11: Criando um novo Middleware Home para o JDeveloper

A partir daqui, a instalação deve ser bastante intuitiva; basta seguir o procedimento até o final.

Sob Windows, um atalho para o JDeveloper deve ter sido criado na sua área de trabalho ou no menu Iniciar. Caso isso não tenha acontecido, basta ir até a pasta de instalação do JDeveloper e executar o arquivo `jdeveloper\jdeveloper.exe`. Inicializado o JDeveloper, uma caixa para seleção de *roles* deve aparecer. Caso não queira pensar nisso agora, basta deixar como está:

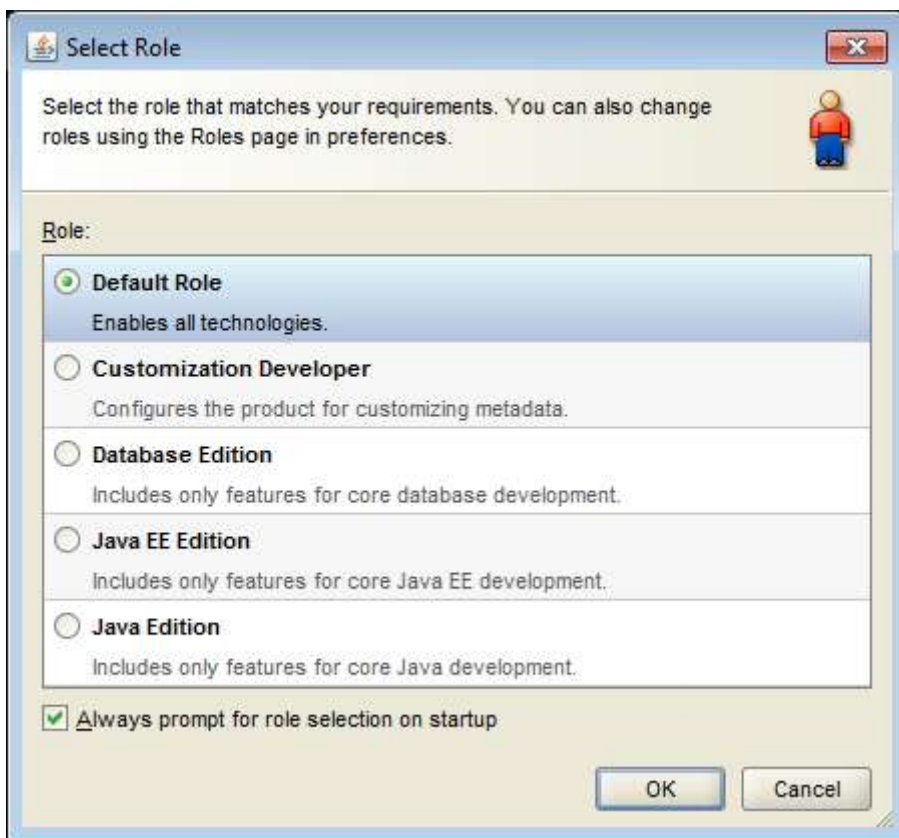


Figura 9.12: Seleção de roles para a execução do JDeveloper

A segunda seleção que deverá aparecer será a quais arquivos associar o JDeveloper. Sugiro marcar as duas primeiras e deixar as outras duas desmarcadas:

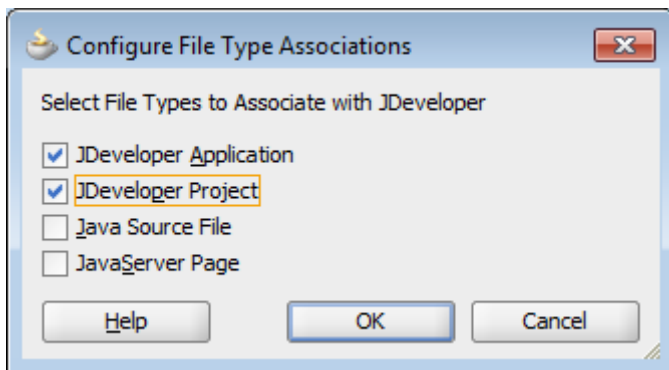


Figura 9.13: Seleção de tipos de arquivos aos quais associar o JDeveloper

O próximo passo é instalar o *plugin* para que seja possível trabalhar com o SOA Suite. Para isso, basta ir até o menu **Help** e selecionar **Check for Updates...**, conforme a imagem:

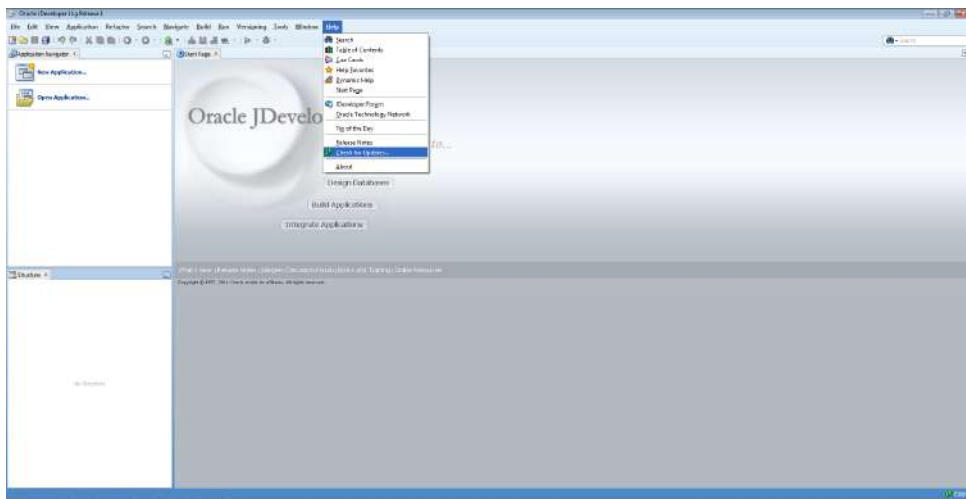


Figura 9.14: Acione o menu de instalação de plugins do JDeveloper

Feito isso, a tela de boas-vindas à checagem de atualizações é exibida. Aperte **Next** e a tela de seleção de fontes de atualizações é exibida. Marque as fontes **Oracle Fusion Middleware Products** e **Official Oracle Extensions and Updates**:

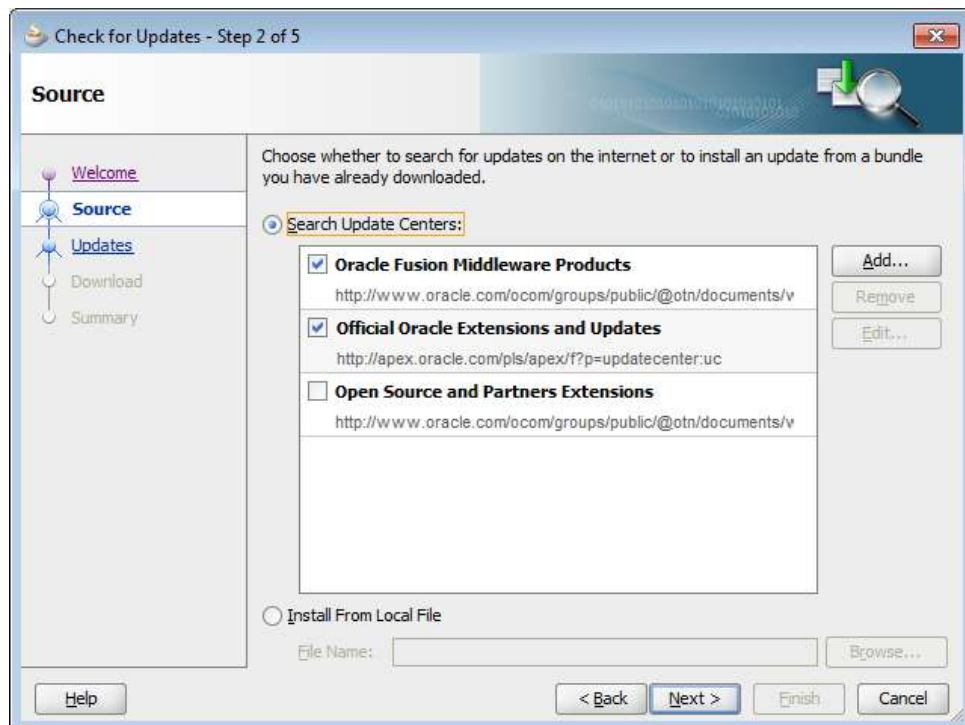


Figura 9.15: Acione o menu de instalação de plugins do JDeveloper

Uma vez feita a checagem por *plugins*, marque o *plugin* Oracle SOA Composite Editor e prossiga. Ao final, o JDeveloper irá solicitar reinicialização. Ao término da reinicialização, a instalação e configuração do JDeveloper está concluída.

9.4 INTRODUÇÃO A BPEL

Está disponível no github (em <https://github.com/alesaudate/soa/tree/master/KnightServices>) um projeto do JDeveloper chamado KnightServices. Este projeto contém um arquivo chamado KnightServices.jws. Ao dar um duplo-clique sobre ele, ele será aberto com o JDeveloper, permitindo a você visualizar sua estrutura:

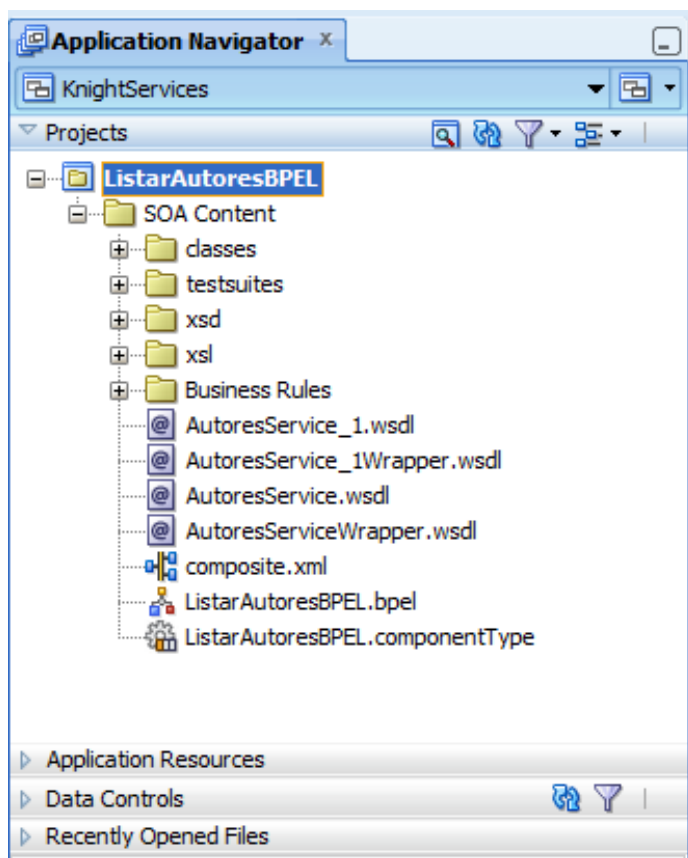


Figura 9.16: Estrutura do projeto BPEL

Ao realizar duplo-clique sobre o arquivo `ListarAutoresBPEL.bpel`, a seguinte estrutura deverá se abrir:

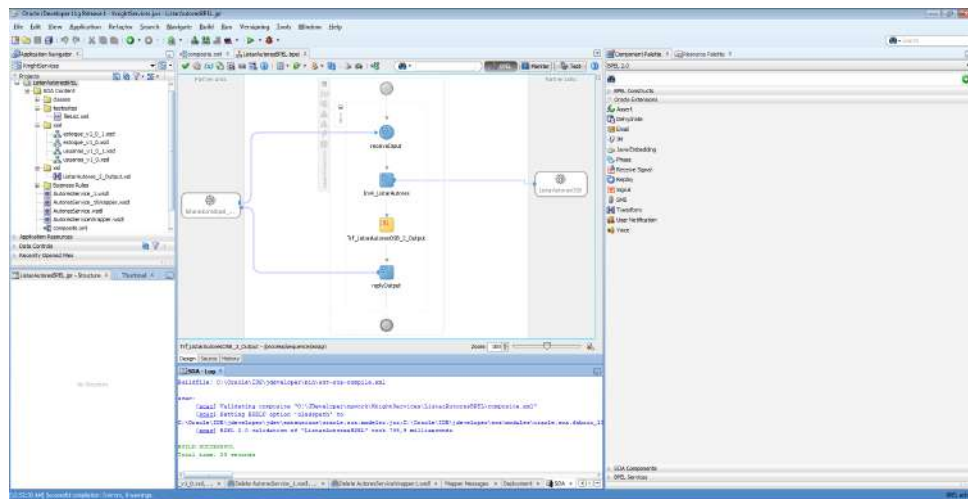


Figura 9.17: Visualização do fluxo BPEL

Este fluxo irá realizar a listagem de autores, delegando para o *web service* exposto no OSB a função de listagem de autores, mapeando os resultados e retornando-os para o cliente. Para realizar a implantação deste fluxo no servidor, clique com o botão direito sobre o projeto, aponte para **Deploy** e clique em **ListarAutoresBPEL...**:

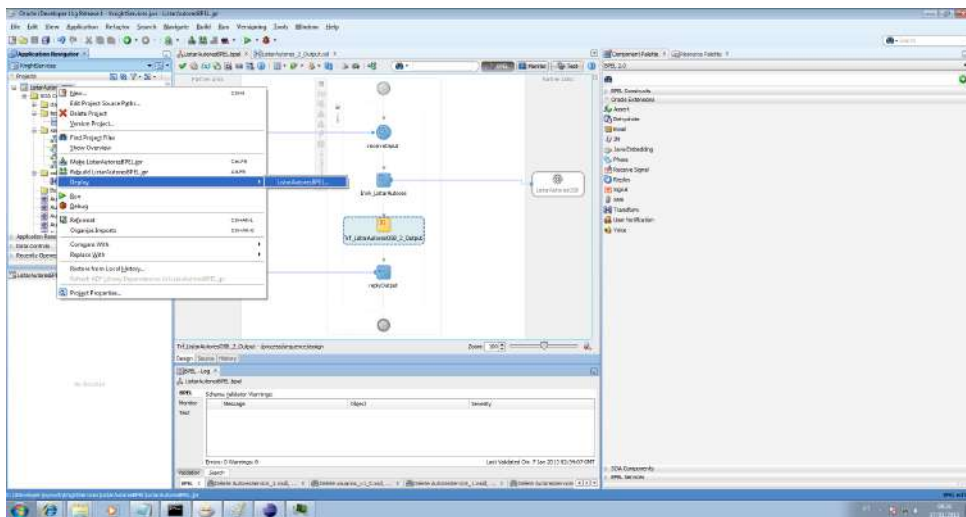


Figura 9.18: Iniciando o deploy do BPEL

Sua primeira implantação vai requerer a definição do *Application Server* onde o BPEL será instalado. Uma caixa de seleção questionará a respeito do tipo de implantação: direta no *Application Server* ou para um SAR — Service ARchive (o que, na prática, significa um arquivo JAR que pode ser implantado em qualquer servidor). Marque a opção *Deploy to Application Server* e clique em *Next*.

A seguir, uma caixa de opções de implantação será exibida. Marque a opção *Overwrite any existing composites with the same revision ID*. e clique em *Next*, de acordo com a imagem:

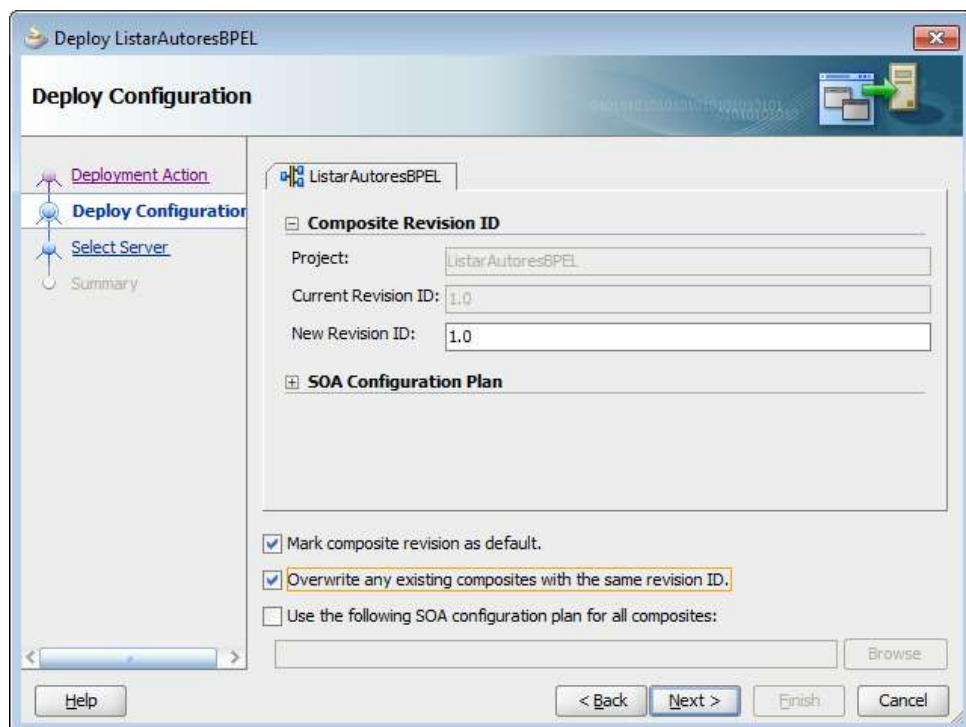


Figura 9.19: Opções de implantação do BPEL

Depois disso, uma listagem de *Application Servers* será exibida, em branco. Para adicionar um *Application Server*, clique no símbolo de mais verde, no topo à direita:

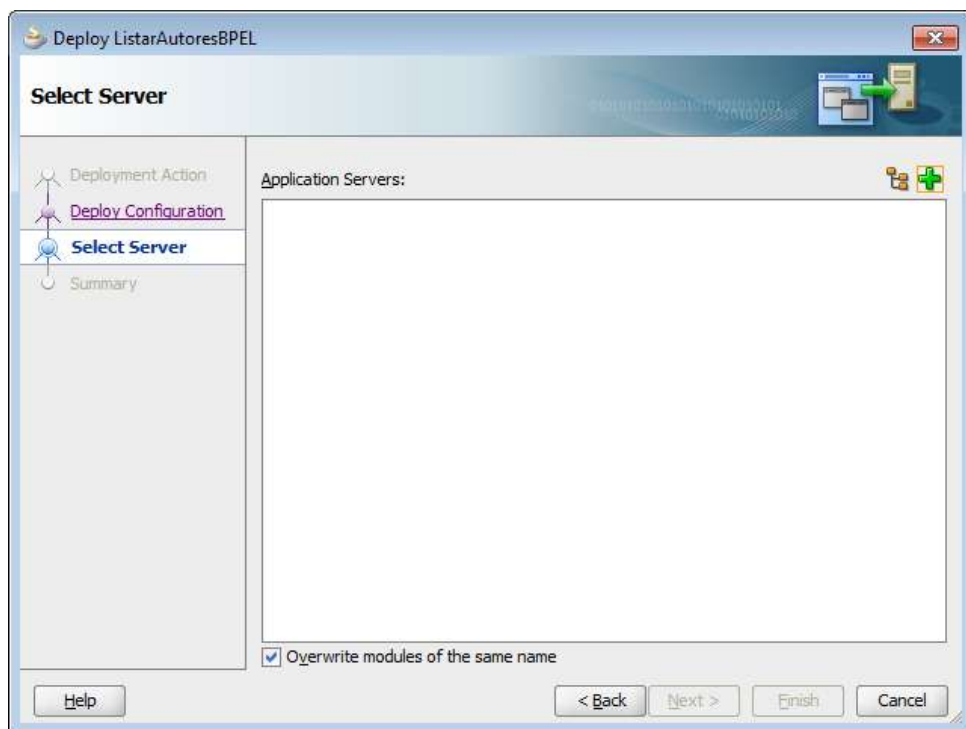


Figura 9.20: Definição de novo application server no JDeveloper

Na caixa que aparecerá, dê um nome para a conexão (sugiro, no caso, localhost). Clique em **Next**. Na próxima caixa, você deverá fornecer o *login* e senha do usuário `weblogic`:

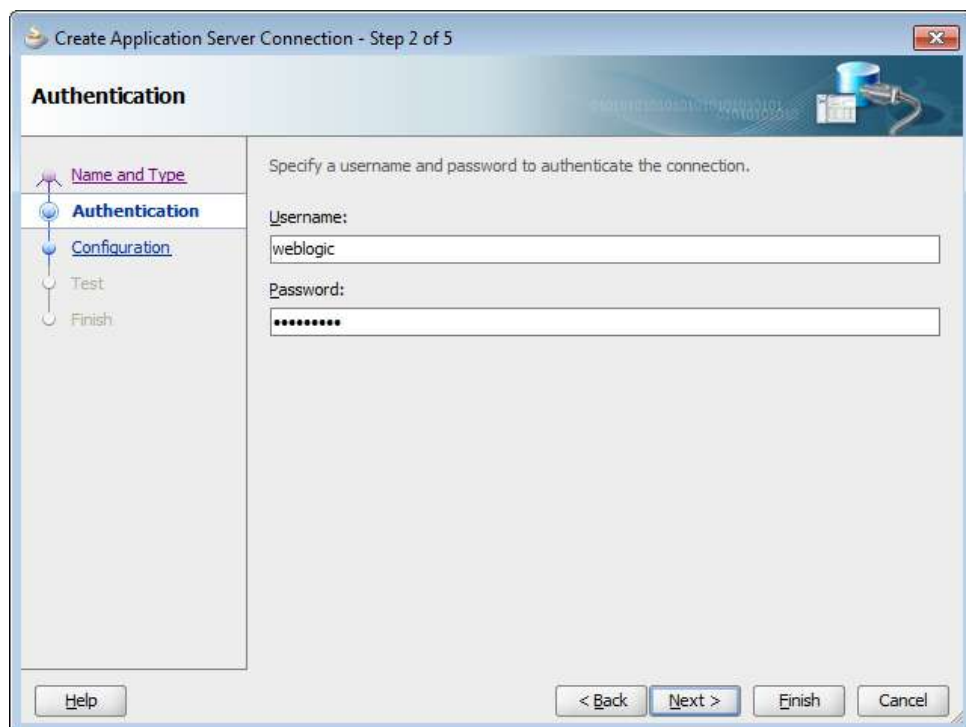


Figura 9.21: Fornecimento do login e senha do Application Server

Em seguida, você deverá fornecer detalhes sobre a instalação do SOA Suite, ou seja, você deverá fornecer o endereço IP, a porta e o domínio do WebLogic. Para estes três valores, você deverá fornecer `localhost`, `7001` e `osb_domain`, respectivamente:

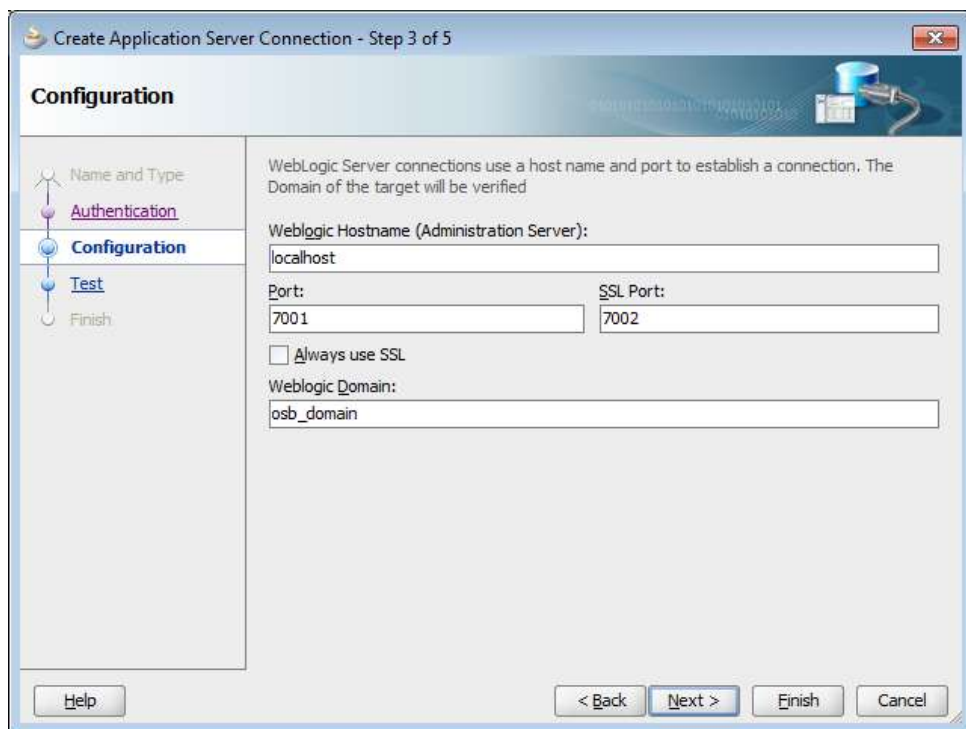


Figura 9.22: Detalhes do Application Server

Na próxima tela, aparecerá uma tela para teste da configuração, que será ativada ao clicar no botão **Test Connection**. Se a configuração estiver correta (e o SOA Suite estiver em execução), ela deverá ficar assim:

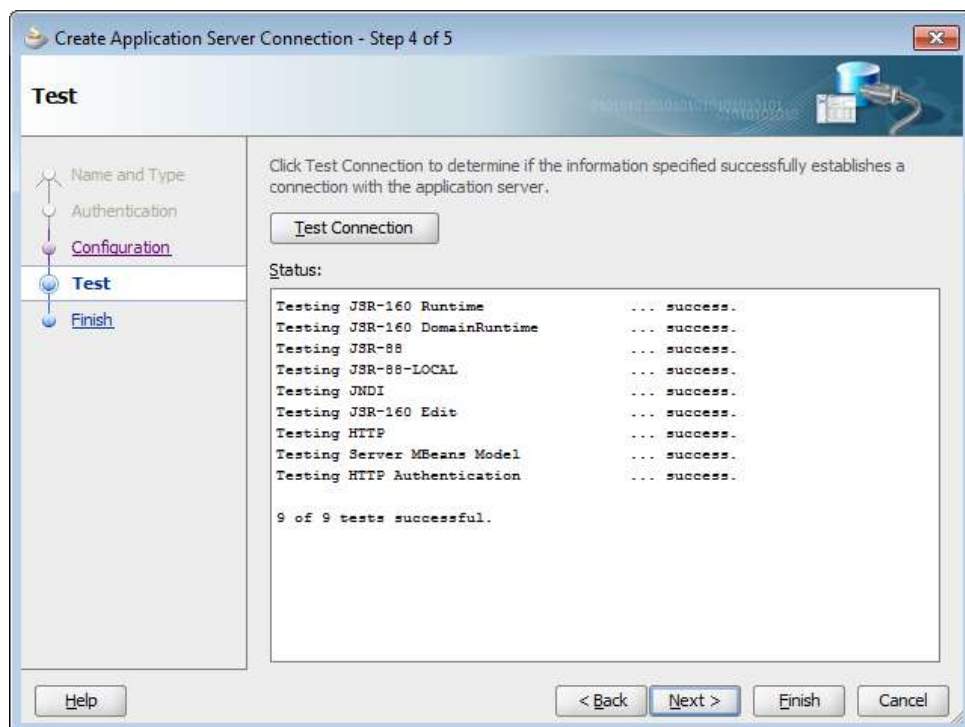


Figura 9.23: Teste da configuração

Na sequência, ao clicar em **Finish**, você deverá voltar à tela de seleção de *Application Servers*. Selecione o recém-criado e clique em **Next**. A sua infraestrutura deve ser exibida da seguinte forma:

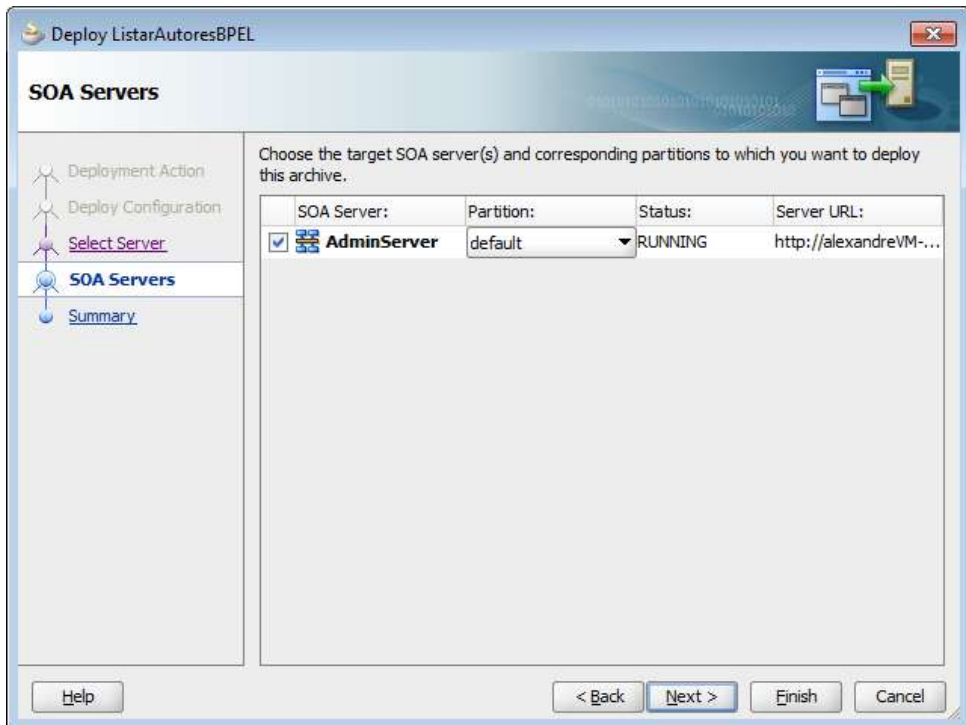


Figura 9.24: Exibição da infraestrutura SOA Suite

Neste momento, você pode clicar em `Finish`. Seu código vai ser compilado e, caso passe deste estágio, será implantado no servidor. O *status* destes passos pode ser conferido no console do `JDeveloper`, em várias abas presentes na parte inferior do mesmo.

Para realizar testes, você pode navegar até o `Enterprise Manager` (uma aplicação para gerenciamento do `SOA Suite` e outros componentes), disponível em <http://localhost:7001/em> (você precisará fornecer o *login* e senha do usuário `weblogic` para acessar a aplicação). Uma vez logado, existe um painel à esquerda listando as composições — basta expandir o menu `SOA`. Expandindo completamente as opções, você deve obter o console de gerenciamento da sua composição BPEL.

Ao clicar no botão `Testar`, você tem duas opções: pode realizar os testes pelo próprio *browser*, pelo formulário que estará presente, ou pode obter o WSDL do fluxo, disponível nesta mesma página.

Neste momento, seu fluxo está pronto. Você pode fazer modificações no fluxo a

partir do JDeveloper, e refazer a implantação sempre que quiser, repetindo os mesmos passos citados.

9.5 SUMÁRIO

Neste capítulo, você visualizou uma das principais ferramentas do universo SOA, BPEL. Você viu como instalar e configurar o `Oracle SOA Suite`, uma das principais ferramentas deste segmento. Você também pôde conferir como instalar e configurar o `JDeveloper`, a IDE padrão para desenvolvimento com o SOA Suite.

Por último, você teve um vislumbre do `Enterprise Manager`, a ferramenta de gerenciamento do `WebLogic`. Esta ferramenta permite, entre outras coisas, visualizar e gerenciar seus fluxos BPEL implantados no SOA Suite.

CAPÍTULO 10

Conclusão

“Nossas dúvidas são traidoras e nos fazem perder o bem que poderíamos conquistar, se não fosse o medo de tentar.”

– William Shakespeare

Você viu, até aqui, toda a base para se construir um projeto SOA adequadamente. Nos primeiros capítulos, você pôde vislumbrar como construir e utilizar *web services* — a base para toda arquitetura orientada a serviços moderna.

Obviamente, não para por aí. Muitas questões foram respondidas; você conheceu, nos quatro capítulos iniciais, a *engine* do JAX-WS e como trabalhar com os chamados *web services* clássicos, ou seja, aqueles baseados em SOAP e WS-*. No capítulo 5, você conheceu REST, uma abordagem diferente para o problema da transmissão de dados entre aplicações.

A partir do capítulo 6, você foi apresentado a problemas novos, e começou a visualizar o uso real de SOA: você passou a **reutilizar** serviços, e não apenas construí-los.

Do capítulo 7 em diante, você foi apresentado a técnicas de desenvolvimento utilizadas em projetos reais — como e para que utilizar o **modelo canônico**, como desenvolver serviços baseados em seus WSDLs (*contract-first*), como coordenar chamadas com BPEL, etc.

Há muito ainda por vir. Como meu amigo Felipe Oliveira mencionou no prefácio, existem diversas técnicas no mundo da orientação a serviços que podem e devem ser utilizadas em projetos reais, e que dariam vida à muitos livros sobre os assuntos. Porém, você deve considerar este um ponto de início para seus estudos de SOA, e não um ponto final. Um segundo livro está sendo escrito por mim, para continuar o mesmo tema. Neste próximo livro, espero dar continuidade aos tópicos apresentados neste, mostrando conceitos ainda mais avançados.

Para continuar seus estudos, existem diversas fontes. Eu tenho mantido alguns artigos em revistas especializadas do ramo de desenvolvimento, onde escrevo sobre temas relacionados a SOA. Além disso, também participo de fóruns — no momento, tanto o GUJ (<http://www.guj.com.br/>) como o SOA Cloud (<http://soacloud.com.br/>). Além disso, o InfoQ (<http://www.infoq.com/br>) também tem sessões especializadas nos segmentos de SOA e *cloud computing*, e é uma fonte de estudos fortemente recomendada.

Como mencionado na introdução, você também está convidado a fazer parte do grupo de discussão próprio deste livro, em <https://groups.google.com/forum/?fromgroups=#!forum/soa-aplicado>. Lá, você pode entrar em contato com outros leitores do livro, tirar dúvidas, fazer sugestões etc.

O mercado de SOA mantém-se muito promissor para todos, sejam analistas, desenvolvedores, arquitetos ou outros papéis. Espero que, com este livro, você se sinta capaz de lidar com os desafios presentes neste universo e, também, sinta-se capaz de prosseguir com sua própria pesquisa.

Bons estudos!

-- Alexandre Saudate

Referências Bibliográficas

- [1] Ralph Johnson John Vlissides Erich Gamma, Richard Helm. Design patterns: Elements of reusable object-oriented software. 1994.
- [2] Thomas Erl. Service-oriented architecture: Concepts, technology design. 2005.
- [3] Thomas Erl. Soa principles of service design. 2008.
- [4] Thomas Erl. Soa design patterns. 2009.
- [5] Eric Evans. Domain-driven design: Tackling complexity in the heart of software. 2003.
- [6] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. 2000.
- [7] Martin Fowler. Anemic domain model. 2003.
- [8] Martin Fowler. Patterns of enterprise application architecture. 2003.
- [9] Bobby Woolf Gregor Hohpe. Enterprise integration patterns: Designing, building, and deploying messaging solutions. 2003.
- [10] Sam Ruby Leonard Richardson. Restful web services. 2007.