Universidade de Brasília
Centro de Informática
Unidade de Serviços de Sistemas de Informação - SSI

Guia Rápido para Administração do Serviço LDAP v3 no Barramento ERLANGMS

Everton de Vargas Agilar

Manual de Arquitetura

Resumo

Este guia sintetiza os conhecimentos que são necessários para trabalhar com a nova arquitetura Erlangms no ambiente Java. Este trabalho é resultado dos esforços realizados no Mestrado em Computação Aplicada pelo Analista Everton de Vargas Agilar do CPD/UnB.

A abordagem proposta por esta arquitetura impõem o uso de um barramento de serviço desenvolvido na linguagem Erlang e um SDK (Software Development Kit) na linguagem que será utilizada para a implementação dos Web-Services, neste caso, o SDK ems-java para a linguagem Java.

De forma muito resumida, a arquitetura ErlangMS tem o intuíto de facilitar a criação e a integração de sistemas através de uma abordagem orientada a serviços no estilo arquitetural REST (Representational State Transfer).

Palavras-chave: Arquitetura Orientada a Serviços, LDAP, ERLANGMS.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este guia sintetiza os conhecimentos necessários para trabalhar com o serviço LDAP v3 no barramento de serviços ERLANGMS.

O serviço LDAP v3 refere-se a implementação protocolo Lightweight Directory Access Protocol, ou LDAP, no barramento de serviços. O LDAP é um protocolo de aplicação aberto, livre de fornecedor e padrão de indústria para ser utilizado para oferecer um "logon único"onde uma senha para um usuário é compartilhada entre muitos serviços.

O barramento de serviços ERLANGMS foi desenvolvido por Everton de Vargas Agilar no Mestrado Profissional em Computação Aplicada da UnB com o intuíto de facilitar a criação e a integração de sistemas através de uma abordagem orientada a serviços no estilo arquitetural REST (Representational State Transfer). No entanto, outros protocolos foram adicionados para suprir tal necessidade, entre eles, um serviço de autenticação de usuários através do protoloco LDAP.

No restante desta introdução, serão apresentados os principais elementos de tempo de execução do barramento de serviços ERLANGMS, para que o leitor possa ter uma visão geral do funcionamento do software antes de proseguir com a instalação e configuração.

1.1 Componentes em Tempo de Execução

Os principais componentes da arquitetura do barramento de serviços ERLANGMS de tempo de execução estão listados a seguir com uma breve descrição sobre o seu objetivo sem entrar em muitos detalhes:

1.1.1 Barramento de serviços (ems-bus)

É o software do servidor. Sua função é interligar os clientes (tipicamente os Front-ends) aos serviços que contém as regras de negócios da organização (como o logon do usuário).

Quando alguém faz uma requisição para um serviço, é o software do barramento que intermedia o envio e o recebimento das mensagens, sendo que o cliente só precisa saber o endereço do barramento e o protocolo utilizado. A Figura 1.1 mostra o esquema típico de roteamento de mensagens HTTP/REST. Este esquem é em tudo semelhante para o protocolo LDAP também.

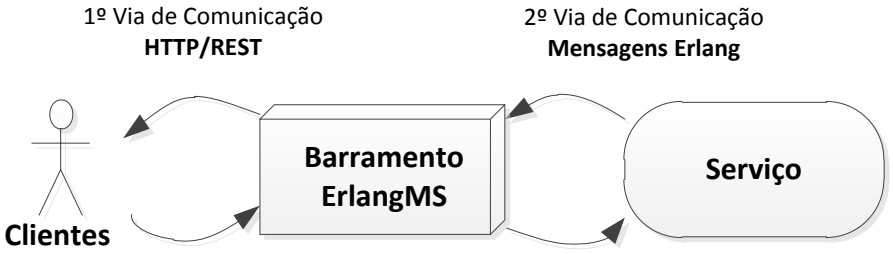

Figura 1.1: Esquema do roteamento das mensagens da arquitetura.

1.1.2 Catálogo de serviços

Representa o componente chave da arquitetura do barramento de serviços pois permite dar visibilidade aos serviços disponibilizados na organização. É no catálogo que os serviços são descritos e registrados. O exemplo a seguir expõem um serviço no catálogo para o serviço LDAP v3 utilizado na integração do software Redmine ao banco de dados de usuários SCA da UnB.

```
{
  "name": "ems_ldap_server",
  "comment": "LDAP service for client integration to the SCA database",
  "owner": "emsbus",
  "version": "1.0.0",
  "service" : "ems_ldap_server:start",
  "url": "/emsbus/ems_ldap_server",
  "type": "KERNEL",
  "lang" : "erlang",
  "tcp_listen_address" : ["0.0.0.0"],
  "tcp_allowed_address" : ["*.*.*.*"],
  "tcp_port": 2389,
  "datasource" : {
    "type" : "sqlserver",
    "connection" : "string conection",
    "primary_key" : "PesCodigoPessoa",
```

```
    "timeout" : 3000,
    "max_pool_size" : 10
  },
  "ldap_admin" : "cn=admin,dc=unb,dc=br",
  "ldap_password_admin" : "xxxxxxxxx",
}
```

Código 1.1: Exemplo de um serviço no catálogo de serviços.

1.1.3 Módulo Back-end

È um projeto Java Web típico do CPD/UnB mas desenvolvido com um padrão de design que será apresentado mais adiante neste guia e que contém a implementação dos serviços. No CPD, estes projetos são publicados em servidores de aplicação JBoss/Wildfly.

1.1.4 Módulo Front-end

Consiste na interface da aplicação bem como a parte que o usuário vê. O front-end é o responsável por coletar os dados de entrada do usuário e realizar as chamadas para os serviços por meio do barramento de serviços.

1.1.5 Servidor de Aplicação JBoss/Wildfly

O servidor de aplicação JBoss/Wildfly é onde são publicados (deployment) os projetos Java Web. Podem existir várias instâncias desses servidores para aumentar a escalabilidade dos serviços, sendo que o barramento despacha as requisições dos clientes utilizando um simples algoritmo round-robin.

1.1.6 Erlang Port Mapper Daemon

É um serviço que executa em segundo plano em cada nó de um cluster e que age como um servidor de nome. É importante salientar que tanto o barramento quanto os servidores de aplicação JBoss/Wildfly são vistos como nós na arquitetura. O cliente não faz parte do cluster pois é apenas um consumidor.

Capítulo 2

Instalando o Barramento de Serviços

Este capítulo apresenta uma revisão dos principais conceitos relacionados ao tema deste trabalho, envolvendo arquiteturas orientadas a serviços com foco em modernização de sistemas legados, estando estruturado da seguinte forma: a Seção 2.1 dá uma visão geral de *Service Oriented Architecture* (SOA). A Subseção 2.1.1 descreve os componentes de um ambiente SOA. A Subseção 2.1.2 investiga duas opções de tecnologia para implementação de *Web Services*. A Subseção 2.1.3 apresenta as justificativas para o uso do estilo arquitetural REST. A Subseção 2.1.4 apresenta o conjunto de restrições de REST que devem ser compreendidos para este trabalho. Por fim, na Seção 2.2, estuda-se duas abordagens de modelagem de domínio do negócio de um software.

2.1 Service Oriented Architecture (SOA)

O termo SOA foi usado pela primeira vez em 1996, quando Roy Schulte e Yeffim V. Natiz (ambos do Gartner) definiram-na como "um estilo de computação de múltiplas camadas que ajudam as organizações a compartilharem as lógicas e os dados de negócios entre os sistemas computacionais" [**?**].

De forma simplificada, SOA compreende uma arquitetura corporativa onde os serviços são criados, reutilizados e compartilhados entre os vários sistemas de uma organização em um sistema distribuído de modo que as funcionalidades possam estar disponíveis a todos os interessados que estejam autorizados a usá-las [**?**, **?**].

Serviços são componentes de software que encapsulam conceitos de negócios de alto nível, composto basicamente por três elementos: a interface, o contrato e a implementação do serviço. A interface define como o fornecedor do serviço viabiliza as requisições dos clientes; o contrato do serviço descreve o serviço (funcionalidade, parâmetros, restrições entre outros atributos); e a implementação é o código do serviço em si [**?**].

Nesse contexto, como observado em [**?**], essa arquitetura corporativa tem sido amplamente utilizada nas organizações em projetos envolvendo a modernização dos sistemas legados, onde os serviços representam os ativos principais com interfaces bem definidas, que podem ser decompostas em módulos interoperáveis possuindo algumas características importantes descritas a seguir [**?**, **?**]:

Valor agregado. Refere-se à capacidade dos serviços de fornecerem valor agregado ao negócio da organização, a qual não recomenda-se que funcionalidades de baixo nível sejam expostas como serviços. Para exemplificar, não faz sentido o CPD disponibilizar serviços como bibliotecas de funções (como tratamento de texto, data e hora, etc) já que não vai agregar valor ao negócio (e possivelmente ocasionar *overhead* na rede [**?**]).

Visibilidade. Capacidade dos serviços de serem encontrados pelos interessados e que suas interfaces sejam bem compreendidas pelo invocador. Neste trabalho, a visibilidade será implementada por meio de um catálogo de serviços que poderá ser consultado em um portal.

Autocontido. Os serviços não devem depender de informações do contexto de outros serviços e também não devem armazenar estados entre as requisições de serviços.

Baixo acoplamento. É um design de arquitetura dos sistemas distribuídos que determina que diferentes partes e funcionalidades de um sistema sejam independentes umas das outras. Assim, alterações em uma determinada parte do sistema não trará consequências para o resto do sistema, trazendo benefícios como escalabilidade, flexibilidade e tolerância a falhas.

Com base nas características apresentadas, pode-se inferir que em um ambiente SOA, os sistemas críticos (possivelmente grandes e complexos) deveriam ser substituídos por sistemas mais simples a partir da composição dos serviços disponíveis em um ambiente distribuído. Essa estratégia representa uma possibilidade para a modernização do Sistema de Informações e Gestão Acadêmica (SIGRA) da Universidade de Brasília.

Dado os benefícios percebidos na literatura, SOA tem potencial para prover mais flexibilidade na modernização dos sistemas legados, haja vista os benefícios com reuso e compartilhamento das funcionalidades que podem ser obtidos. Contudo, no contexto da UnB, o uso de uma abordagem orientada a serviço ainda precisa ser investigada, pois como afirmam os autores [**?**, **?**, **?**, **?**], há alguns *trade-offs* a considerar, entre eles, a maior complexidade dos sistemas distribuídos; a preocupação com a segurança e acesso aos serviços, o *overhead* ocasionado na rede com a troca de mensagens; e a necessidade das equipes de TI dominarem algumas tecnologias talvez não habituadas.

2.1.1 Componentes de um Ambiente SOA

Um ambiente SOA abrange a interação de três componentes: o provedor do serviço (*Service Provider*), o consumidor (*Service Consumer*) e o catálogo de serviços (*Service Broker*) [**?**, **?**]. A interação entre esses elementos é conhecida como "*find-bind-execute paradigm*" [**?**], que significa paradigma "procura-consolida-executa". De forma resumida, os provedores (que são os fornecedores dos serviços) devem consolidar as informações sobre os serviços no catálogo de serviços (um repositório central para os serviços) com a descrição desses serviços. Com base nas informações armazenadas nesse catálogo, os consumidores (que são os clientes) podem identificar e solicitar a execução dos serviços requeridos junto ao provedor, conforme ilustra a Figura 2.1.

(a) Provedor do Serviço. Na perspectiva do negócio, corresponde ao dono do serviço ou o fornecedor do serviço. Em uma perspectiva arquitetural, é a plataforma onde os *hosts* (usuários dos serviços) acessam os serviços oferecidos. O provedor de serviços disponibiliza acesso aos serviços e a especificação desses serviços são publicadas no registro de serviços. Essa publicação permite que os clientes localizem os serviços e requisitem sua execução ao provedor.

(b) Consumidor do Serviço. São os clientes que requisitam serviços. Note que, o consumidor do serviço por ser uma pessoa, uma aplicação ou mesmo outro serviço;

(c) Catálogo de Serviços. É a localização central dos serviços (como um repositório) onde o provedor pode publicar tais serviços e o consumidor pode encontrá-los.

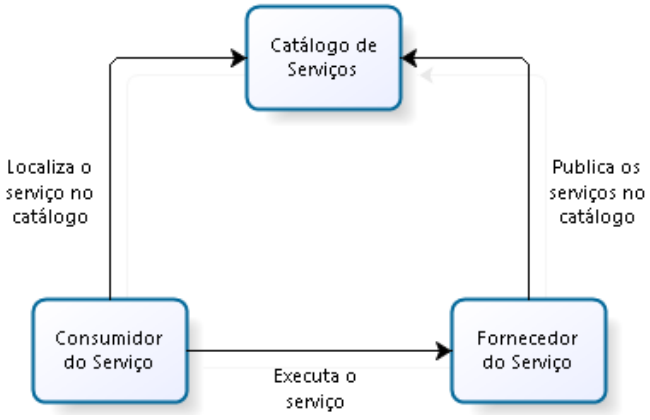

Figura 2.1: Relacionamentos entre os elementos de uma arquitetura SOA.

2.1.2 Web Services SOAP e REST

Destacam-se atualmente duas tecnologias para a implementação de um ambiente SOA, os *Web Services* SOAP e REST. Ambos são muito utilizados (e até mesmo combinados) para invocação de serviços de negócios em um sistema distribuído [**?**, **?**]. O transporte de dados desses serviços é realizado tipicamente pelos protocolos *Hypertext Transfer Protocol* (HTTP) ou *Hypertext Transfer Protocol Secure* (HTTPS) para conexões seguras.

Os *Web Services* SOAP, acrônimo de *Simple Object Access Protocol* representam um conjunto de tecnologias e padrões da indústria definidas pela *World Wide Web Consortium* (W3C) com um suporte tecnológico bastante maduro por parte dos fornecedores de tecnologia que as suportam [**?**]. Este tipo de *Web Service* é baseado na linguagem *eXtensible Markup Language* (XML) para a especificação da estrutura e o formato das mensagens, impondo restrições no formato das mensagens (funcionando como um contrato de serviço). A linguagem XML consiste basicamente em uma linguagem de marcação extensível, quer permite especificar as mensagens de forma simples e legível por meio de *tags* personalizadas [**?**]. Por exemplo, em SOAP, tanto a descrição dos serviços como a requisição e a resposta de uma invocação a um serviço são mensagens no formato XML. Usa-se a linguagem WSDL para descrever a estrutura das mensagens SOAP e as ações possíveis em um *endpoint* (URL onde o serviço pode ser acessado pela aplicação). Assim, o WSDL nada mais é do que um documento em formato XML descrevendo o serviço oferecido, como acessá-lo, e quais as operações e os métodos disponíveis na especificação do serviço [**?**].

Os *Web Services* REST, acrônimo de *Representational State Transfer* representam outro tipo de serviço que está sendo adotado pelas organizações (substituindo ou complementando os *Web Services* SOAP). De acordo com Roy Fielding [**?**], REST é um estilo arquitetural baseado no protocolo de hipermídia HTTP, sendo introduzido para implementar *Web Services* fracamente acoplados. O *JavaScript Object Notation* (JSON) é um formato para intercâmbio de dados (baseado em um subconjunto da notação de objetos da linguagem Javascript [**?**]) utilizado preferencialmente neste tipo de serviço como uma alternativa mais simples e leve ao XML [**?**]. REST baseia-se em recursos e verbos [**?**]. Cada recurso pode ser referenciado através de sua URI (por exemplo, `http://sistemas.unb.br/alunos/100` obtém o recurso aluno cuja a identificação é 100). As ações de um recurso são providas pelos verbos do protocolo HTTP, que compreendem os métodos *GET* para obter a representação de um recurso; *POST* para criar novos recursos; *PUT* para modificar um recurso existente; *DELETE* para excluir um recurso existente. Também podem ser usados os métodos *HEADER*, para recuperar os metadados de uma representação do recurso e *OPTIONS* para obter a descrição ou a documentação sobre o recurso desejado.

Os *Web Services* facilitam a interoperabilidade entre aplicações heterogêneas. Nesta seção, foram investigadas duas opções de tecnologia para implementar uma abordagem SOA, embora existam outras alternativas disponíveis no mercado. Salienta-se que ambas tecnologias (SOAP e REST) são perfeitamente viáveis para o propósito do trabalho. No entanto, optou-se por experimentar o REST, uma abordagem emergente que tem tido cada vez mais aceitação na indústria devido a sua flexibilidade e porque o CPD/UnB está interessado em utilizá-lo. No restante deste capítulo será apresentado em mais detalhes a abordagem REST e as restrições deste estilo arquitetural que se fazem importantes para o trabalho de dissertação.

2.1.3 Justificativa para o Uso de REST

Os *Web Services* impõem uma camada de abstração que permitem aos componentes do tipo cliente, solicitar serviços aos componentes do tipo servidor [**?**]. No entanto, essas abstrações frequentemente causam alguma lentidão (*overhead*), comprometendo o desempenho das aplicações, como observado em [**?**]. Isso é mais evidente em SOAP devido ao tamanho das mensagens, já que tais mensagens são descritas na linguagem WSDL e envelopadas pelo protocolo SOAP, ambos no formato XML [**?**, **?**].

REST representa uma alternativa ao SOAP, pois aceita o uso de JSON em vez do formato XML para a especificação das mensagens [**?**, **?**]. Segundo [**?**], uma das suas principais vantagens consiste na facilidade no desenvolvimento, o aproveitamento da infraestrutura web existente e um esforço de aprendizado menor. Daí que, como este trabalho tem como foco a modernização de sistemas legados, através de uma abordagem orientada a serviço, quer se evitar que as aplicações tenham que lidar com vários protocolos (caso fosse escolhido o tipo de *Web Service* SOAP). Sem dúvida, o estilo arquitetural REST pode ser mais facilmente adotado no CPD/UnB, pois os sistemas legados precisam apenas ter condições de fazer requisições HTTP/HTTPS aos serviços disponíveis e manipular o formato de dados JSON.

Por outro lado, *Web Services* SOAP apresentam algumas vantagens sobre REST, conforme cita o autor [**?**]: a) da perspectiva do desenvolvedor, possuem um apelo a contrato de serviços; b) a linguagem WSDL usada neste tipo de serviço, permite descrever o layout das mensagens, o que de certa forma, facilita a integração dos sistemas, permitindo gerar automaticamente a implementação do cliente do *Web Service* de forma padronizada; e c) o protocolo SOAP compreende um método de transporte genérico, podendo usar qualquer meio de transporte para enviar a requisição (não somente HTTP/HTTPS). REST é mais fácil de entender e acessível, porém faltam padrões e a tecnologia é considerada apenas um estilo arquitetural [**?**]. Em contrapartida, SOAP é um padrão da indústria, com protocolos bem definidos e um conjunto de regras bem estabelecidas.

Na abordagem proposta neste trabalho, um dos requisitos definidos foi o uso de um catálogo de serviço (em formato JSON) para catalogar os serviços disponíveis. Espera-se que deficiências de REST quanto a ausência de um formato para especificação dos serviços sejam suplantadas com uma certa flexibilidade. Não obstante, os *Web Services* REST já foram utilizados no CPD, na implementação de um serviço para enviar dados requeridos para o Sistema de Gerenciamento de Protocolo (UnBDoc) em ASP a partir do sistema SIEX em Java. Essa experiência mostrou-se positiva, razão pela qual os membros da Divisão de Serviço de Sistemas de Informação (SSI) querem experimentar *Web Services* REST em uma abordagem de modernização orientada a serviços.

Contudo, para que REST possa ser utilizada no CPD/UnB de maneira correta, devem ser observados um conjunto de restrições de arquitetura, conforme afirma [**?**]. Esse conjunto de restrições serão apresentados na Subseção 2.1.4.

2.1.4 As Restrições REST

Para o desenvolvimento de sistemas orientado a serviços com REST, existe um conjunto de restrições específicas para auxiliar o desenho das aplicações [**?**]. Quando tais restrições são seguidas, os sistemas denominam-se *RESTful*. Note que, como sugere [**?**, **?**], as restrições de arquitetura REST são mais do que um guia com regras e, quando aplicadas, torna-se possível explorar os benefícios da *Web* em seu benefício. Em resumo, as cinco restrições do estilo arquitetural REST são descritas a seguir:

Cliente-Servidor

A restrição Cliente-Servidor demanda a separação dos componentes cliente e servidor e estabelece uma arquitetura em camadas. Segundo [**?**], o princípio que norteia esta restrição é a separação das responsabilidades entre os componentes para que possam evoluir separadamente. Entre os benefícios disto, de acordo com [**?**], está a maximização da portabilidade ao permitir múltiplas interfaces com o usuário entre diferentes plataformas e o aumento da escalabilidade ao simplificar os componentes de servidor.

Stateless

Esta restrição estabelece que a interação entre o cliente e o servidor não deve manter estados entre as comunicações. Assim, as requisições que o cliente envia ao servidor precisam conter toda a informação para descrever a solicitação, uma vez que no lado do servidor, não vai existir qualquer tipo de armazenamento de sessão.

Entre os benefícios desta restrição está a escalabilidade do servidor, pois não há recursos alocados entre as requisições, permitindo liberar rapidamente os recursos após seu

uso e a visibilidade, pois o servidor somente processa as requisições sem se preocupar com a natureza integral do pedido [**?**, **?**].

No entanto, existem alguns *trade-offs*, por exemplo, a performance da rede pode diminuir com o aumento dos dados repetitivos vindo nas requisições pela ausência de estado no servidor [**?**].

Embora os sistemas *Web* da UnB não sejam RESTFul, como comparação, verificou-se o emprego de uma abordagem oposta a esta restrição, chamada *stateful*. Nesse caso, os sistemas *Web* da Instituição fazem uso de sessão, que mantêm os dados do cliente na memória do servidor de aplicação. Como consequência, em ocasiões de muitos acessos aos sistemas, como nos eventos de extensão da Universidade ou no início das aulas, alguns sistemas tornam-se instáveis, como é o caso dos sistemas Sistema de Gestão do Restaurante Universitário (SISRU) e o Sistema de Informações e Extensão (SIEX). Isso é devido aos problemas de escalabilidade nos servidores de aplicação identificados pelos técnicos do CPD juntamente com uma consultoria externa contratada pelo CPD em 2013.

Cache

A restrição de *cache* impõem que a resposta de uma solicitação de serviço seja implicitamente ou explicitamente rotulada como *cacheable* (que faz uso de *cache*) ou *não cacheable* (não está sujeito ao *cache*). De acordo com [**?**], mecanismos de *cache* podem ser colocados em vários locais ente o servidor e o cliente. Além disso, o protocolo HTTP também pode ser utilizado através dos campos do cabeçalho para controle do *cache* das mensagens.

O maior benefício do uso desta restrição, na visão de [**?**, **?**, **?**], é o aumento do desempenho com a redução de chamadas ao serviço na rede. Outro benefício observado pelo autor é que eles podem eliminar parcialmente ou completamente algumas interações entre o cliente e o servidor, melhorando a eficiência das aplicações e a escalabilidade do servidor. Um *trade-off* é a perca da confiabilidade, pois os dados contidos no *cache* podem estar defasados em relação aos dados obtidos diretamente do servidor. Em razão deste *trade-off*, essa restrição não será utilizada na arquitetura proposta neste trabalho.

Interface Uniforme

Interface Uniforme impõem uma restrição na interface de troca de mensagens entre o cliente e o servidor, a partir de um conjunto predefinido de operações, sendo esta restrição obtida com o uso dos verbos HTTP com suas respectivas semânticas [**?**, **?**, **?**, **?**].

Segundo [**?**, **?**], esta restrição enfatiza o uso de uma interface uniforme entre os componentes, simplificando a arquitetura e melhorando a visibilidade das interações. Contudo, isso pode diminuir a eficiência das aplicações, pois os dados são enviados em um formato padrão, em vez de um formato específico utilizado pelas aplicações.

Para que seja possível obter uma interface uniforme, REST define 4 restrições de interface: Identificação dos recursos, representação de recursos, mensagem auto descritivas e utilização de hipermídia para o estado da aplicação. A identificação de recursos estabelece que os recursos devem ser identificados. Esta restrição é implementada pelo HTTP através da URI do recurso que representa uma sequência de caracteres para localizar tal recurso físico ou abstrato [**?**]. A representação de recursos diferencia o recurso de sua representação, permitindo múltiplos formatos de dados para um mesmo recurso, como o JSON, o XML ou apenas texto puro [**?**]. Para implementar esta restrição, REST baseia-se no *Multipurpose Internet Mail Extensions* (MIME) das mensagens que são definidas no cabeçalho da requisição do serviço. Desta forma, o *payload* (corpo das mensagens) contêm a representação do recurso pré-acordado entre o cliente e o servidor na solicitação da mensagem [**?**]. Mensagens auto descritivas requerem que as mensagens contenham toda a informação sobre o recurso para descrever a sua representação. Assim, as mensagens devem informar os metadados no cabeçalho para indicar como o seu conteúdo será tratado [**?**]. Por fim, na restrição de utilização de hipermídia para estado da aplicação (HATEOAS), os recursos solicitados pelas aplicações devem possuir *hiperlinks* para possibilitar a navegação entre os recursos relacionados [**?**]. A razão disso é que o servidor não armazena estado de sessão, sendo responsabilidade do desenvolvedor da aplicação criar as representações adequadamente [**?**]. Contudo, conforme salienta [**?**], esta restrição não é muito utilizada por ser um conceito novo entre os desenvolvedores.

Sistema em camadas

Essa restrição tem a finalidade de dividir o sistema em camadas para que os componentes participantes somente interajam com o componente adjacente. Um dos benefícios desta restrição é promover a independência entre as camadas e reduzir a complexidade dos sistemas. Outro benefício importante desta restrição é encapsular os serviços legados [**?**]. Como *trade-off*, existe uma possível redução do desempenho do sistema por causa do *overhead* da inclusão de camadas [**?**].

Code-On-Demand

Esta é uma restrição opcional. Permite que as funcionalidades dos clientes sejam estendidas e simplificadas com o *download* e execução de códigos no lado do cliente [**?**].

2.2 Modelagem de Domínio do Negócio

A fim de modelar um sistema, a literatura descreve diversos padrões de design de software, tais como a utilização de uma arquitetura em camadas contendo artefatos especificamente projetados para um determinado interesse [?, ?, ?, ?, ?, ?, ?]. Em [?], por exemplo, os autores abordam uma arquitetura de quatro camadas que permite a separação das responsabilidades entre as camadas e favorece a educação de novos desenvolvedores quanto a arquitetura, o planejamento do desenvolvimento incremental, o planejamento das tarefas e a definição dos prazos, entre outros.

Entretanto, alguns autores acreditam [?, ?, ?, ?], que o maior desafio está, na maioria das vezes, na complexidade para entender e modelar o domínio de negócio com o qual a aplicação deve lidar e não na arquitetura em si do software, razão pela qual a separação em camadas possivelmente não é o suficiente.

Para lidar com esses desafios, há alguns padrões de design identificados na literatura que objetivam organizar a lógica negocial. São eles: o *Domain-Driven Design* (DDD) e o *Transaction Script*. Esses padrões focam-se nos aspectos de modelagem da aplicação, para que o software construído reflita adequadamente as necessidades que deverão ser contempladas para o usuário final.

O padrão DDD organiza a lógica de domínio do negócio de um sistema em um modelo de objetos ricos, sendo indicado quando há muita complexidade [?, ?]. Por outro lado, existem situações, nas quais um modelo de domínio rico não seria tão indicado, como na criação de cadastros simples, que não possui muita regra de negócio, por exemplo. Nessas situações, o padrão *Transaction Script* poderia ser utilizado [?].

O padrão *Transaction Script* organiza a lógica negocial do sistema em um conjunto de métodos que lidam com as requisições desde a camada de apresentação até a camada de persistência. O uso de *Transaction Script* é bem conhecido pelos desenvolvedores da Divisão de Serviço de Sistemas de Informação (SSI), pois alguns sistemas da UnB que foram migrados para Java utilizam este padrão, como é o caso do Sistema de Tabelas (SITAB), do Sistema de Transporte (SITRAN), do Sistema de Compras de Materiais (SIMAR) e do Sistema de Informações e Extensão (SIEX).

Capítulo 3

Configurando o Serviço LDAP v3