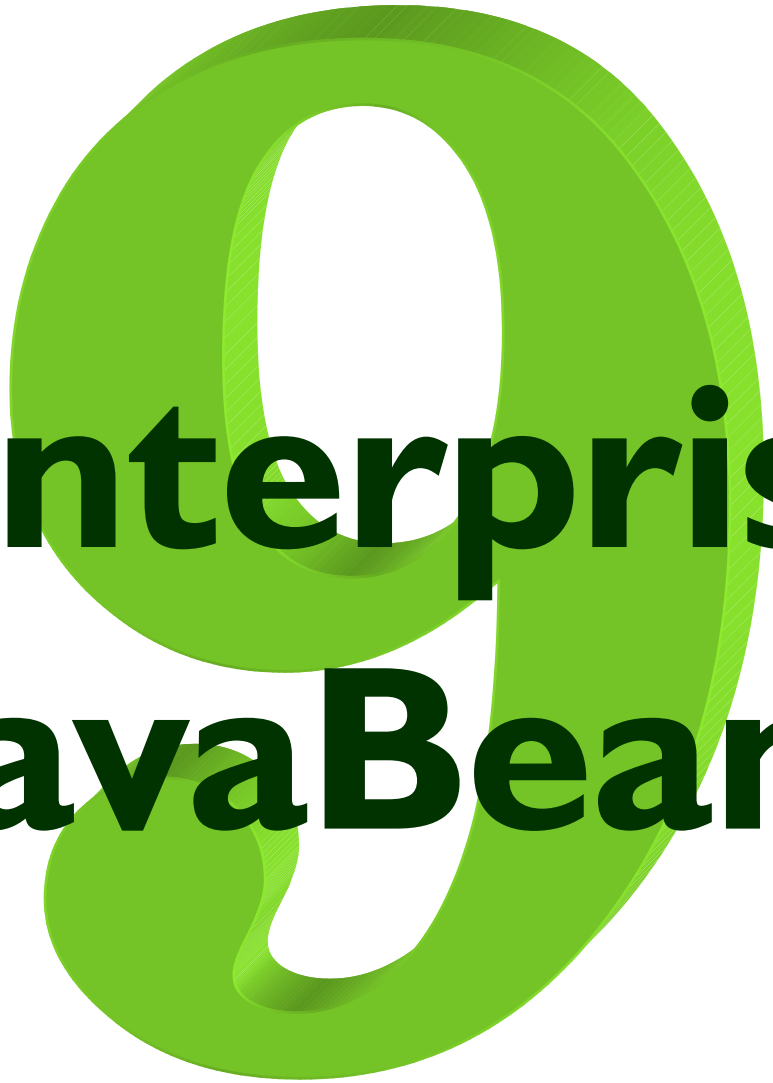


Enterprise JavaBeans

Helder da Rocha
www.argonavis.com.br

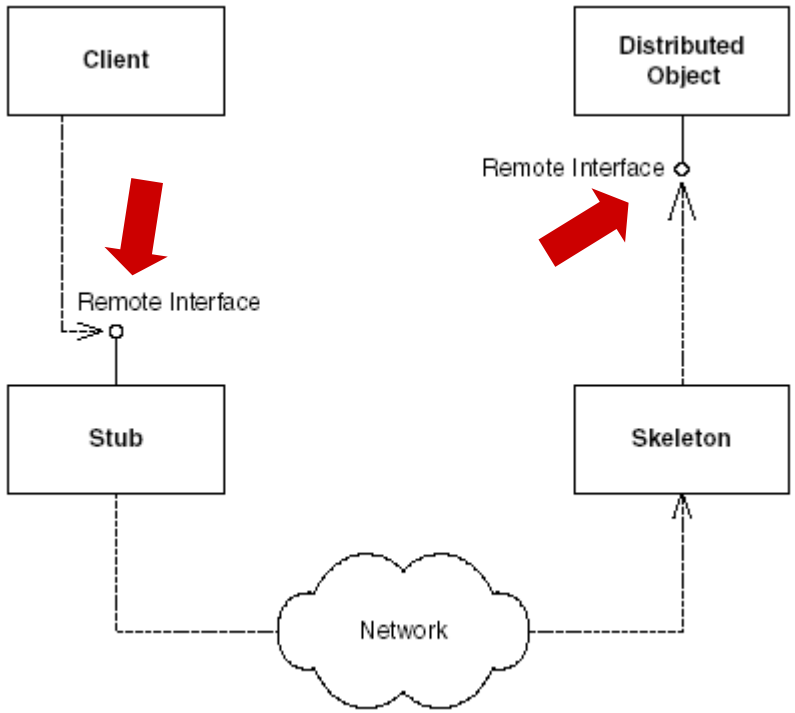
Sumário

- *Este módulo é uma breve introdução aos conceitos comuns a todos os EJBs*

- *Por que EJB?*
  - *Como funciona o middleware implícito*
- *O que é um Enterprise Bean*
  - *Componentes de um EJB*
  - *EJB Object*
  - *EJB Home*
  - *Deployment Descriptor*
  - *Arquivos do fabricante*

Por que EJB? (I)

- Sistemas de objetos distribuídos têm em comum uma arquitetura que consiste de
  - O objeto remoto
  - Um stub que representa o objeto remoto no cliente
  - Um esqueleto, que representa o cliente no servidor



- O cliente conversa com o stub pensando tratar-se do próprio objeto
- O esqueleto conversa com o objeto remoto que pensa que é o cliente quem fala com ele
- O stub e o objeto remoto implementam a **mesma interface**!

Fonte da ilustração: Ed Roman, Mastering EJB 2

Por que EJB? (2)

- Desenvolver um sistema distribuído não é simples.
  - Exige se concentrar em aspectos da aplicação que nada tem a ver com o problema de negócio a ser solucionado
  - Exige a observação de condições que geralmente são menos importantes ou dispensáveis em aplicações locais
- É preciso se preocupar com
  - A performance, que pode sofrer abalos em rede
  - Os custos de uma possível ampliação da capacidade
  - A segurança dos dados, controle de acesso, permissões
  - Transações: tarefas poderão falhar no meio do caminho
  - Integridade dos dados: clientes irão acessá-los simultaneamente
  - O que acontecerá se parte do sistema sair do ar?
- Mas esses são problemas de qualquer sistema distribuído
  - Será que precisamos implementar tudo isto?
  - Será que não estaremos reinventando a roda?

Por que EJB? (2)

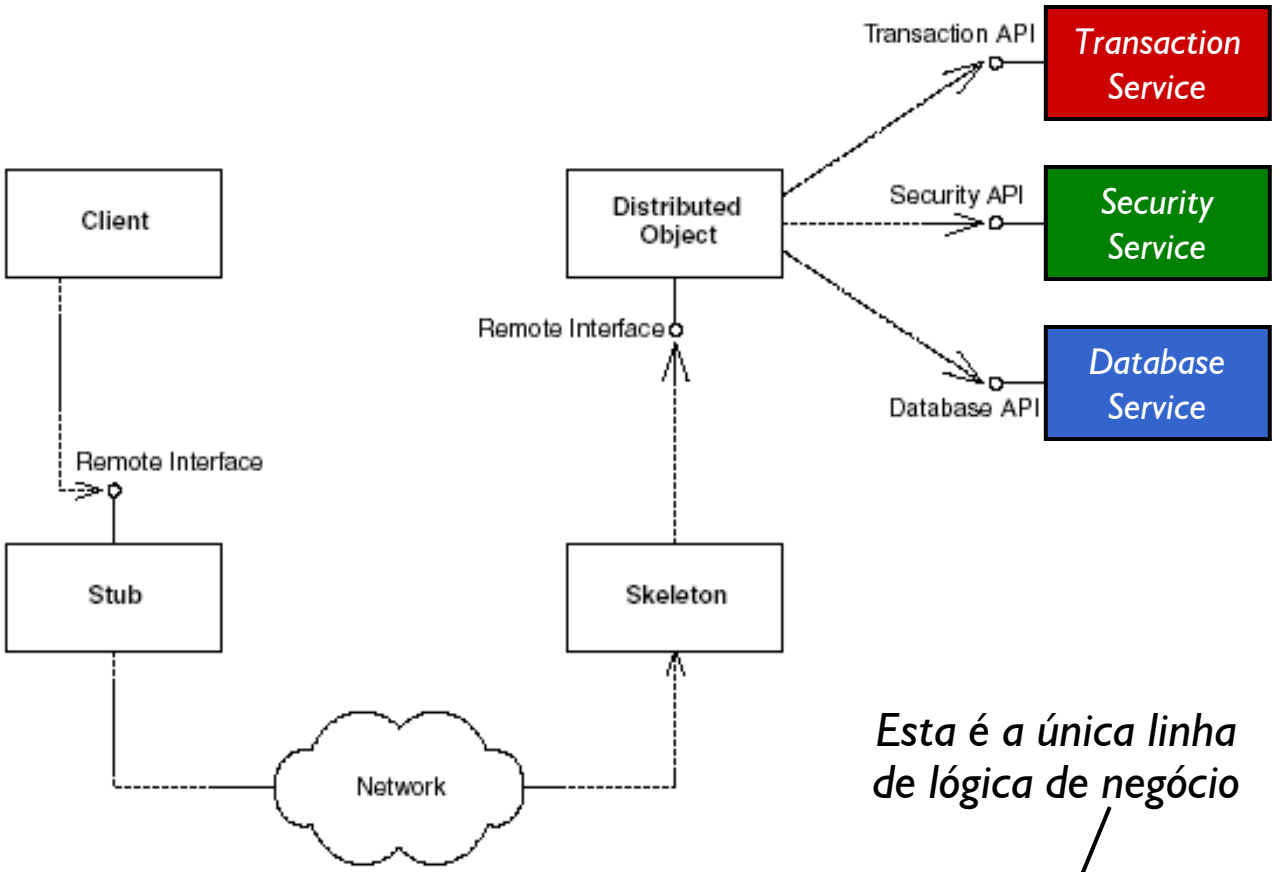
- *Vários serviços que são necessários ou desejáveis em qualquer sistema distribuído*
  - *Comunicação síncrona e assíncrona (acoplamento fraco - messaging)*
  - *Autenticação, autorização, criptografia e segurança em geral*
  - *Redirecionamento transparente e balanceamento de cargas*
  - *Controle de transações e garantia de integridade das informações*
  - *Suporte a threads*
  - *Clustering (replicação), caching e gerenciamento de recursos*
  - *Acesso otimizado a conexões de banco de dados, soquetes, objetos*
  - *Integração com sistemas legados e de outros fabricantes*
  - *Gerenciamento remoto; geração de logs e estatísticas*
- *Desenvolver esses serviços exige amplos conhecimentos em áreas que pouco tem a ver com a aplicação*
- *Solução: usar middleware! Mas qual?*

Tipos de middleware

- Temos uma aplicação com vários objetos distribuídos e um servidor que oferece serviços de middleware. E agora? Como acessar os serviços?
- Serviços podem ser acessados através de uma API
  - Sua aplicação precisará escrever o código para usar a API, por exemplo, JDBC para controlar acesso a bancos de dados, ou JTA para controlar transações, ou APIs de segurança: middleware explícito
- Serviços podem ser configurados através de interfaces ou arquivos de configuração externos ao código
  - Sua aplicação não precisa conter código algum de acesso a recursos externos. Você declara os serviços que precisa externamente: middleware implícito

Middleware explícito

- *Tem que ser implementado em cada método de cada objeto que deseja utilizar o serviço*



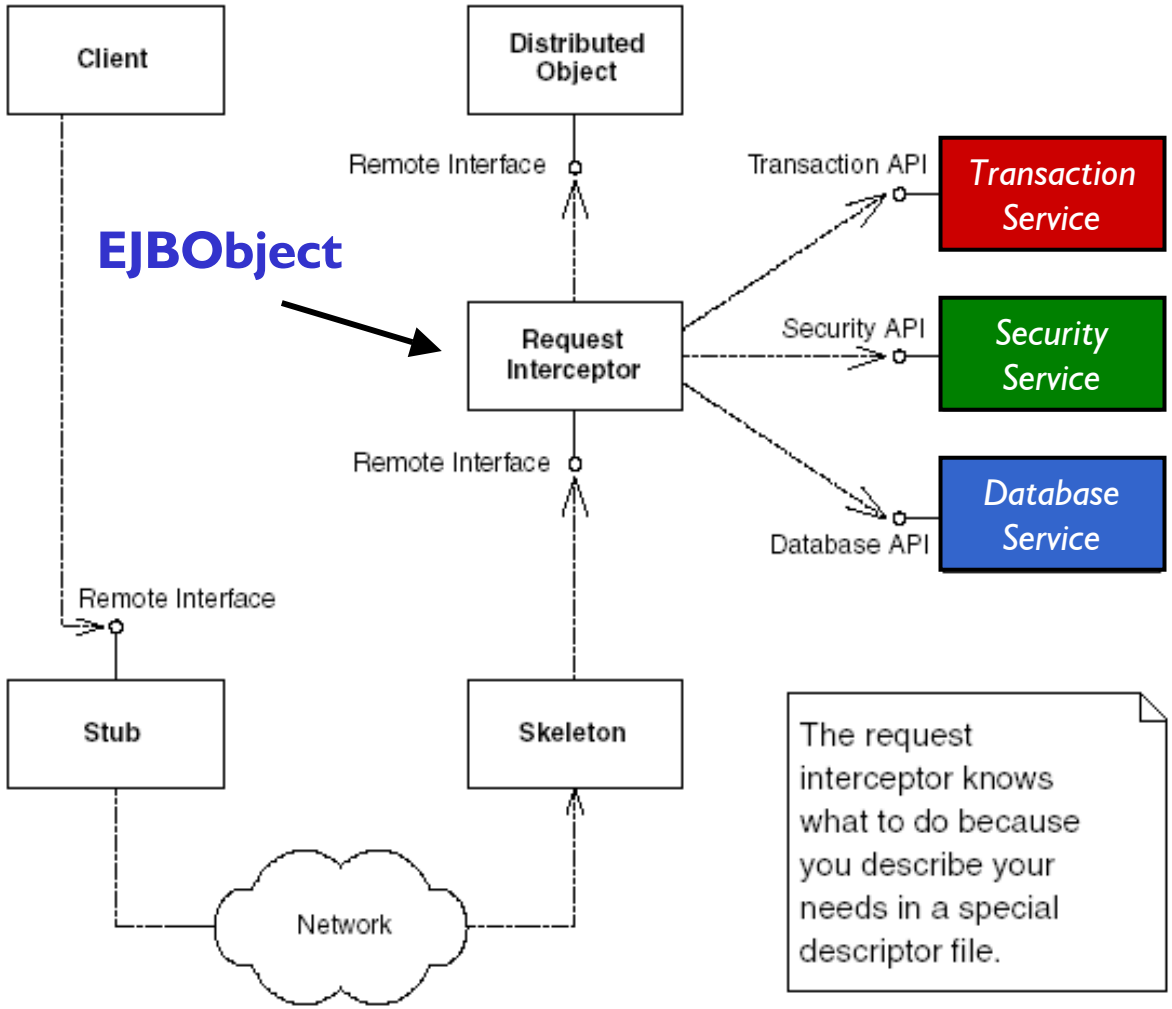
Esta é a única linha de lógica de negócio

```
public void transferir(Conta um, Conta dois, double valor) {
    // chama API para verificar segurança da requisição
    // chama API para iniciar a transação
    // chama API para carregar dados do banco
    // Subtraia o valor do saldo da conta um e some na conta dois
    // chama API para guardar dados no banco
    // chama API para fechar a transação
}
```

Fonte da ilustração: [4]

Middleware implícito

- *Escreva seu objeto para conter apenas lógica de negócio*
- *Declare os serviços de middleware que seu objeto precisa em um arquivo separado*
- *Rode uma ferramenta que pega o descritor e gera um objeto interceptador de requisições*




```
public void transferir(Conta um, Conta dois, double valor) {
    // Subtraia o valor do saldo da conta um e some na conta dois
}
```

Fonte da ilustração: [4]

Componentes de um EJB

- *Um EJB não é um único arquivo monolítico: são vários arquivos que trabalham juntos*
  - *Classe EJB*
  - *Interface Remote*
  - *Interface Home*
  - *Deployment descriptor*
  - *Empacotamento em JAR (EJB-JAR)*

Classe EJB

- *Contém a lógica de negócio*
- *É uma classe Java que obedece a uma interface definida e obedece certas regras*
- *Implementações diferem, dependendo do tipo*
  - *Session Beans - lógica relacionada a processos de negócio (computar preços, transferir fundos)*
  - *Entity Beans - lógica relacionada a dados (mudar o nome de um cliente, reduzir o saldo)*
  - *Message-driven Beans - lógica orientada a eventos (lógica assíncrona)*

Interfaces padrão

- A especificação defina algumas interfaces padrão que sua classe pode implementar
  - Forçam seu bean a expor certos métodos (definidos pelo modelo de componentes)
  - Container chama esses métodos para gerenciar seu bean e alertá-lo sobre eventos
- Todos implementam a interface *javax.ejb.EnterpriseBean*
  - implementa Serializable e não contém métodos
- Interfaces derivadas de EnterpriseBean
  - SessionBean
  - EntityBean
  - MessageDrivenBean

EJB Object

- *Enterprise Beans não são objetos remotos que podem ser usados de forma independente, como outros objetos RMI-IIOP*
  - *Um cliente (ou o skeleton) nunca chama o método diretamente em uma instância do bean: a chamada é interceptada pelo Container e delegada à instância*
  - *Ao interceptar requisições, o container pode automaticamente realizar middleware implicito*
- *O **EJBObject** é quem implementa a interface remota. Ele é o interceptador que chama o Bean*
  - *É objeto inteligente que sabe realizar a lógica intermediária que EJB container requer antes que uma chamada seja processada*
  - *É a "cola" entre o cliente e o bean: replica e expõe cada método de negócio do bean e delega todas as requisições*

Containers

- *Entidades abstratas*
- *Não são intermediários entre o cliente e o bean*
- *Podem interagir com beans chamando métodos de gerência quando necessário (métodos que só o container invoca) e avisam quando algum evento de middleware vai acontecer*
- *Conectam os clientes aos beans, oferecendo persistência, gerenciando o ciclo de vida, etc.*
- *Nem clientes nem beans explicitamente codificam para a API do container*
- *Container implicitamente gerencia o overhead de uma arquitetura de componentes distribuídos*

Interface Remote

- *Como as ferramentas sabem quais metodos de negocio devem ser replicados no EJBObject?*
- *Interface Remote é quem define todos os métodos de negócio expostos pelo bean*
- *Precisam ser derivadas de interface javax.ejb.EJBObject*
  - *Você declara mas não implementa os métodos de Remote (como faria com RMI). O container é quem gera o EJBObject que implementa a interface!*

RMI-IIOP versus EJB Objects

- Qualquer objeto que implementa javax.rmi.Remote é um objeto remoto e chamável de outra máquina virtual
  - Seus EJB Objects (gerados pelo Container) são objetos RMI-IIOP!
  - Interfaces Remote EJB são simplesmente interfaces Remote RMI-IIOP, só que precisam aderir à especificação EJB
- Regras de Remote
  - Todos os métodos provocam RemoteException
  - Interfaces Remote precisam suportar tipos de dados que podem ser passados via RMI-IIOP.
  - Tipos permitidos: primitivos, objetos serializáveis e objetos remotos RMI-IIOP

Instance pooling

- Gerência do ciclo de vida e atividade das instâncias de EJBs no servidor
- Servidor inteligentemente gerencia recursos (threads, conexões e objetos) para oferecer melhor desempenho, menos consumo de recursos
- Container dinamicamente instancia, destroi e reusa beans como for necessário
  - Se bean já existe na memória, container pode reutilizá-lo
  - Se bean está inativo, container pode destruí-lo
- Pool de beans pode ser bem menor que número de clientes conectados: "think time"

EJB Home

- Como clientes adquirem referências a objetos EJB?
  - Para obter uma referência a um objeto EJB, cliente usa uma fábrica de objetos EJB
  - Fábrica é responsável por operações do ciclo de vida de um objeto EJB (criar, destruir, encontrar)
  - Fábricas de EJBs são chamados de objetos EJBHome
- Objetos **EJBHome** servem para
  - Criar objetos EJB
  - Encontrar objetos EJB existentes
  - Remover objetos EJB
- São parte do container e gerados automaticamente
  - Implementam interface Home definida pelo programador

Interfaces locais

- Um problema das interfaces remotas é que criar objetos através dela é um processo lento
  1. Cliente chama um stub local
  2. Stub transforma os parâmetros em formato adequado à rede
  3. Stub passa os dados pela rede
  4. Esqueleto reconstrói os parâmetros
  5. Esqueleto chama o EJBObject
  6. EJBObject realiza operações de middleware como connection pooling, transações, segurança e serviços de ciclo de vida
  7. Depois que o EJBObject chama o Enterprise Bean, processo é repetido no sentido contrário
- Muito overhead! Em EJB 2.0 é possível chamar EJBs através de sua interface local
  1. Cliente chama objeto local
  2. Objeto EJB local realiza middleware
  3. Depois que o trabalho termina devolve o controle a quem chamou

Interfaces locais: conseqüências

- Interfaces locais são definidas para objetos EJBHome e para EJBObject (interfaces EJBLocal e EJBLocalHome)
- Benefícios
  - Pode-se escrever beans menores para realizar tarefas simples sem medo de problemas de performance
  - Uso típico: fachadas Session que acessam Entities
- São opcionais
  - Substituem ou complementam as interfaces remotas existentes
- Efeitos colaterais
  - Só funcionam ao chamar beans do mesmo processo (não dá para mudar o modelo de deployment sem alterar o código)
  - Parâmetros são passados por referência e não por valor: muda a semântica da aplicação!!!

Deployment descriptors

- Para informar ao container sobre suas necessidades de middleware, o bean provider deve declarar suas necessidades de middleware em um deployment descriptor (arquivo XML)
  - Informações de gerência e ciclo de vida (quem é o Home, o Remote, se é Session, Entity ou MDB)
  - Requisitos de persistencia para Entity Beans
  - Requerimentos de transações
  - Segurança (quem pode fazer o que com que beans, que metodos pode usar, etc.)
- Pode-se usar uma ferramenta (geralmente a ferramenta do próprio container) para gerar o arquivo

Vendor-specific files

- Cada servidor tem recursos adicionais que podem ser definidos em arquivos específicos
  - Como configurar load-balancing
  - clustering, pooling
  - monitoramento
  - mapeamento JNDI
- Geralmente são gerados por ferramentas no próprio servidor
- Podem também ser codificados à mão (jboss.xml)
- Devem ser incluidas no bean antes do deployment

EJB-JAR

- Arquivo JAR que contém tudo o que descrevemos
- Podem ser gerados de qualquer forma
  - ZIP, Ant, ferramentas dos containers
  - IDEs (NetBeans e deploytool)
- Uma vez feito o EJB-JAR, seu bean está pronto e é unidade implantável em application server
  - Ferramentas dos containers podem decomprimir, ler e extrair informações contidas no EJB-JAR
  - Depois, ferramentas realizam tarefas específicas do fabricante para gerar EJB Objects, EJB Home, importar seu bean no container, etc.
- Pode-se ter vários beans em um ejb-jar

helder@ibpinet.net

www.argonavis.com.br