Boas Práticas no desenvol

Aprenda a desenvolver componentes e serviços

A tecnologia EJB permite a construção de aplicações portáveis, reusáveis e escaláveis.

Serviços como transações, segurança, persistência e distribuição não precisam ser reinventados, fazendo com que o desenvolvedor foque na implementação das regras de negócio e não perca tempo com a infra-estrutura do código (veja a **Figura 1**).

A versão EJB 3.0, mantida pela JSR 220 (*http://www.jcp.org/en/jsr/detail?id=220*), oferece inovações na codificação de componentes EJB, com um foco em melhorias de facilidade de desenvolvimento.

Através do suporte a POJOs e o uso de anotações (padronizadas a partir da plataforma Java EE 5 e baseada no Java 5), a criação de EJBs ficou mais simples e fácil, pois não depende da configuração de arquivos XML, o que era considerado um trabalho maçante para o desenvolvedor (**Figura 2**).

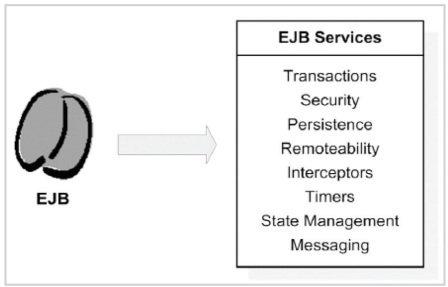


Figura 1. Serviços disponibilizados pela arquitetura EJB.

Figura 2. Definindo o modelo EJB 3.

Nota do DevMan

Veja o que você vai aprender adicionalmente neste artigo:

- Os conceitos sobre Container-Management Transaction (CMT) e Bean-Management Transaction (BMT);
- A funcionalidade das APIs JNDI e JTA;
- A definição das annotations do Java 5;
- Para relembrar, a definição de POJOs;
- Comparação entre EJB Stateless/Stateful Session Beans.

Nota do DevMan

POJO (Plain-Old Java Object): São objetos java que seguem um design mais simplificado. Por definição, seguem a estrutura rígida de ter um construtor default, sem argumentos, e métodos que seguem o padrão getters e setters.

Resumo DevMan

De que se trata o artigo:

O artigo demonstra as boas práticas no desenvolvimento de aplicações com a tecnologia EJB 3.0, dando ênfase à modelagem orientada a objetos (como criar um modelo OO persistente), o uso de design patterns conhecidos, como: *Data Transfer Object* (DTO), *Session Facade*, *Data Access Object* (DAO), *Service Locator*; controle transacional: CMT ou BMT, controle de exceções e logs, e o uso de ferramentas para testes unitários.

Para que serve:

Este artigo serve para revisar conceitos, de orientação a objetos e uso de design patterns, e explorar o uso de ferramentas disponibilizadas pela tecnologia EJB 3.0, tais como interceptors, JTA e persistência.

Em que situação o tema é útil:

Em casos onde é possível implementar a camada de negócio (*business layer*) com a tecnologia EJB 3.0.

Boas práticas no desenvolvimento com EJB3:

Desenvolver componentes corporativos que sejam flexíveis, reusáveis e escaláveis não é uma tarefa fácil. Serviços como transações, segurança, persistência e distribuição são complicados e demandam muito tempo de codificação. Porém, a tecnologia EJB disponibiliza todos estes serviços implementados e prontos para usar. A nova tecnologia EJB 3 é baseada em POJOs e anotações (padrão Java 5), sendo um padrão definido pela plataforma Java EE 5, trazendo benefícios e um novo estilo de desenvolvimento de aplicações com EJBs.

vimento com EJB 3 utilizando a tecnologia EJB 3

Utilize o poder da tecnologia EJB 3 para criar componentes e serviços reutilizáveis e orientados a objetos

FÁBIO AUGUSTO FALAVINHA

Nota do DevMan

Anotações (Annotations): Incluída na versão Java 5, annotations são meta-informações que podem ser adicionadas a classes, métodos, variáveis, parâmetros e pacotes, podendo ser interpretadas pelo compilador Java, por ferramentas e bibliotecas (para geração de código, arquivos XML, por exemplo) e em tempo de execução utilizando *Reflection*. O uso de annotations permite especificar ou alterar o comportamento da aplicação, sem manipular diretamente a lógica de programação. Em versões anteriores do Java, o comportamento da aplicação poderia ser modificado via arquivos XML, conhecidos como *deployment descriptors*.

Listagem 1 . Uso da anotação @Inheritance.

```
// Herança por InheritanceType.JOINED
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Pessoa implements Serializable { ... }

@Entity
public class Contato extends Pessoa { ... }

// Herança por InheritanceType.SINGLE_TABLE
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class Pessoa implements Serializable { ... }

@Entity
public class Contato extends Pessoa { ... }

// Herança por InheritanceType.TABLE_PER_CLASS
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Pessoa implements Serializable { ... }

@Entity
public class Contato extends Pessoa { ... }
```

Neste artigo, vamos fazer uma revisão de boas práticas no desenvolvimento com EJB, visando à modelagem orientada a objetos (como criar um modelo OO persistente), o uso de design patterns conhecidos, como: *Data Transfer Object* (DTO), *Session Facade*, *Data Access Object* (DAO), *Service Locator*; controle transacional: CMT ou BMT, controle de exceções e logs, e o uso de ferramentas para testes unitários.

Criando um modelo Orientado a Objetos persistente

Na versão EJB 2.x era quase impossível criar um modelo orientado a objetos. A implementação baseada em *Entity Beans* era orientada a registros das tabelas do banco de dados, e o fraco mapeamento OO/relacional padronizado não dava suporte algum para herança e polimorfismo.

A versão EJB 3.0, traz um novo modelo de persistência, o *Java Persistence API* (JPA). Com este modelo é possível desfrutar, por exemplo, da anotação @Inheritance para a implementação de modelos OO com suporte a herança e polimorfismo (observe a **Listagem 1**).

Esta anotação contém o atributo strategy, que define a estratégia de persistência para as tabelas e propriedades do relacionamento (superclasse e classes filhas). Os tipos de estratégia são:

- InheritanceType.JOINED: este tipo define que as subclasses serão recuperadas através de um *join*, relacionando a própria chave da tabela ou especificando uma chave através da anotação @PrimaryKeyJoinColumn(name="PESSOA_ID");
- InheritanceType.SINGLE_TABLE: este tipo define que as propriedades da *superclass* e das *subclasses* serão mantidos em única tabela;
- InheritanceType.TABLE_PER_CLASS: este tipo define que os relacionamentos serão feitos via *UNION SQL*. Portanto, não é possível definir chaves primárias com geração de ID do tipo *AUTO* ou *IDENTITY*.

A criação de um modelo de entidades orientado a objetos não é um trabalho fácil. Há duas opções para criar este modelo:

- Implementar a lógica de negócio nas entidades:
  - Vantagem: para aplicações que NÃO atuam como serviços é a implementação ideal. Serviços são criados para exportar, através de uma ou mais operações, processos de negócio de uma determinada aplicação. Se o modelo criado não for dependente de outras aplicações, então este modelo será mais simples e flexível, e terá um aumento na reusabilidade das entidades como parte robusta da aplicação.
  - Desvantagem: uso deste modelo para aplicações que tenham acesso externo. Pois, a dependência de uma entidade será "espalhada" por diversas aplicações, e qualquer mudança na estrutura deste modelo irá causar grandes alterações nas aplicações que dependam destas entidades.
- Implementar a lógica de negócio utilizando-se o pattern Business Object (BO):
  - Vantagem: desacoplar a lógica de negócio das entidades, onde estas serão apenas representações das tabelas do

Listagem 2 . Uso da anotação @MappedSuperclass.

```
@MappedSuperClass
public class Pessoa implements Serializable {

 @Basic
 @Temporal(TemporalType.TIMESTAMP)
 public Date getDataNascimento() { ... }

}

@Entity
public Contato extends Pessoa { ... }
```

Listagem 3 . Uso da anotação @PersistenceContext.

```
-- "persistente.xml" dentro do arquivo jar do módulo EJB.

@Stateless
public class ContatoBean implements ContatoRemote, ContatoLocal {

  @PersistenceContext(unitName="jm-ejb3-entityPU")
  private EntityManager entityManager;

}

-- "persistente.xml" dentro do arquivo jar das entidades.

@Stateless
public class ContatoBean implements ContatoRemote, ContatoLocal {

  @PersistenceContext(unitName="jm-ejb3-entity.jar#jm-ejb3-entityPU")
  private EntityManager entityManager;

}
```

Nota do DevMan

EJB Stateless/Stateful Session Beans: a tecnologia EJB disponibiliza dois tipos de EJB Session Beans: *Stateless* e *Stateful*. *EJBs Stateless* não armazenam o estado da sessão do cliente nas invocações aos seus métodos, diferente de *EJBs Stateful* que armazenam este estado. Por este fato, *Stateless Session Beans* são considerados mais performáticos por não salvar o estado de sessão do cliente, isto é, ao finalizar a chamada de um método, o container recupera esta instância ao "pool" de EJBs, onde será associado a outro cliente através de uma nova chamada. Já *EJBs Stateful* armazenam o estado da sessão do cliente e só irão retornar ao "pool" de EJBs quando for finalizado a execução do processo (múltiplas chamadas a métodos do EJB). Este tipo de EJB é definido por *activation* e *passivation*. *Activation* é realizado quando o container recupera a instância EJB, que está localizada no repositório secundário para o repositório primário, com isso o container invoca o método callback "ejbActivate()" para que o bean recupere os recursos necessários para continuar a execução de seus processos. *Passivation* é realizado quando o container irá salvar a instância EJB para o repositório primário, porém, antes o container invoca o método callback "ejbPassivate()" para liberar os recursos em uso no processo.

banco de dados. Neste modelo, as lógicas de negócio são encapsuladas em classes java, muitas vezes terminam com o sufixo BO, retirando a dependência das entidades por parte da aplicação.

◦ Desvantagem: a má implementação deste pattern pode causar um forte acoplamento entre as camadas da aplicação, não conseguindo separar onde começa a lógica de negócio e o uso da entidade, ou seja, a lógica de negócio fica espalhada pelas camadas da aplicação. Usar este pattern é encapsular a lógica de negócio em um elemento único dentro da aplicação.

Muitos desenvolvedores não gostam de adicionar regras de negócio nas entidades, pois seguem, ainda, um modelo procedural, o que torna complexa a implementação de componentes EJBs reutilizáveis. Apenas como exemplo, vamos pensar em um EJB Stateful Session Bean **CartaoCreditoBean** que contém um método **efetuarPagamento()** com 30 linhas de código, ou seja, metade do código poderia estar organizado nas entidades ou em classes BOs, mas está proceduralmente implementado, o que torna difícil a manutenção e a extensão de novas implementações a esta funcionalidade.

Lembre-se que o JPA suporta a implementação de um modelo orientado a objetos, onde há possibilidade de implementar regras de negócio nas entidades, garantindo assim, um desacoplamento dos componentes Stateless/Stateful Session Beans.

Outra anotação, importante para criação de um modelo OO, é @MappedSuperclass. Esta anotação permite que atributos comuns sejam copiados para a *superclass*, mantendo assim um modelo mais organizado. Diferente da anotação @Inheritance, esta anotação não persiste a superclasse como tabela, apenas seus atributos são persistidos na tabela filha como colunas, garantindo a organização do modelo OO, como descrito na **Listagem 2**.

Estas duas anotações ajudam na criação de um modelo OO utilizando conceitos de herança e polimorfismo. Outras anotações podem ser utilizadas para ajustar um modelo Entidade Relacionamento (ER) para um modelo orientado a objetos. Sendo que a maioria das aplicações começa pela construção do banco de dados, o que, às vezes, dificulta a criação de um modelo organizado.

Um conceito antigo, mas muito comentado atualmente é o *Domain-Driven Design* (DDD). Este conceito enfatiza que objetos do modelo de negócio não são apenas réplicas das tabelas do banco de dados. Na arquitetura EJB 3, estes objetos são conhecidos como entidades e podem conter regras de negócio.

Uma boa prática à criação das entidades é criar um projeto independente da aplicação que conterá os componentes EJBs. Desta forma, uma biblioteca será criada com todas as entidades e classes do modelo de negócio, garantindo flexibilidade na estrutura da aplicação.

Vamos exemplificar esta boa prática utilizando a IDE Netbeans 6.1, onde criaremos um projeto do tipo *Java Class Library*. Este projeto irá conter apenas as entidades e classes necessárias do modelo de negócio, veja a **Figura 3**.

Esta estrutura permite que a aplicação tenha diversos *pesistence.xml*, pois cada um dos arquivos estará dentro do arquivo *jar* no diretório META-INF do componente

EJB, vide *jm-ejb3-module1*, como pode ser visto na **Figura 4**.

Caso o arquivo *persistence.xml* estiver dentro do *jar* onde estão as entidades, então deve-se informar o caminho respeitando a estrutura do arquivo EAR, veja a **Listagem 3**.

Observe que no arquivo *persistence.xml* está sendo utilizada tag <jar-file>. Esta tag informa ao container EJB que as entidades estão localizadas no arquivo, informado pelo valor desta tag, no exemplo, *jm-ejb3-entity.jar*. Caso haja mais de um módulo EJB, com *persistence.xml* diferente, porém utilizando a mesma biblioteca de entidades, apenas repita a configuração.

Portanto, ao implementar EJBs deve-se apenas criar um atributo **EntityManager** e marcar este atributo com a anotação **@PersistenceContext**. Caso haja mais de uma configuração de persistência, informe o nome da unidade de persistência (*persistence unit*) através do atributo **unitName** (**Listagem 3**).

Em código, a aplicação está organizada e permite flexibilidade e extensibilidade, entretanto, o arquivo EAR deve estar estruturado para comportar a configuração realizada (**Figura 5**).

A versão EJB 3 permite que as entidades sejam trafegadas de uma camada para outra, por exemplo da camada EJB (*business layer*) para a camada de apresentação (*view layer*), sem sofrer nenhuma alteração em sua estrutura. Nas versões antigas da tecnologia EJB, não era possível trafegar as informações contidas nas entidades. Assim, para trafegar as informações, foi necessária a criação de uma classe, que iria conter apenas as principais informações. Este modelo tornou-se o design pattern *Data Transfer Object*, mas este pattern **NÃO** deve ser utilizado ao desenvolver aplicações com a tecnologia EJB 3. Nos próximos tópicos será abordado a questão de uso de alguns patterns com a tecnologia EJB 3.

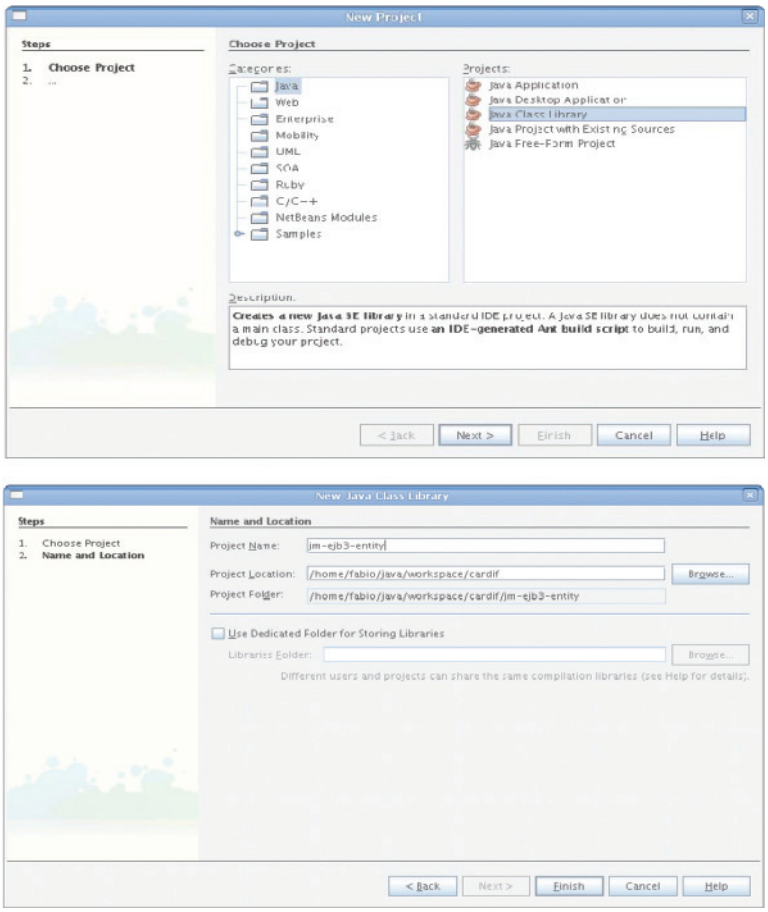


Figura 3. Criação do projeto Java Class Library na ferramenta Netbeans.

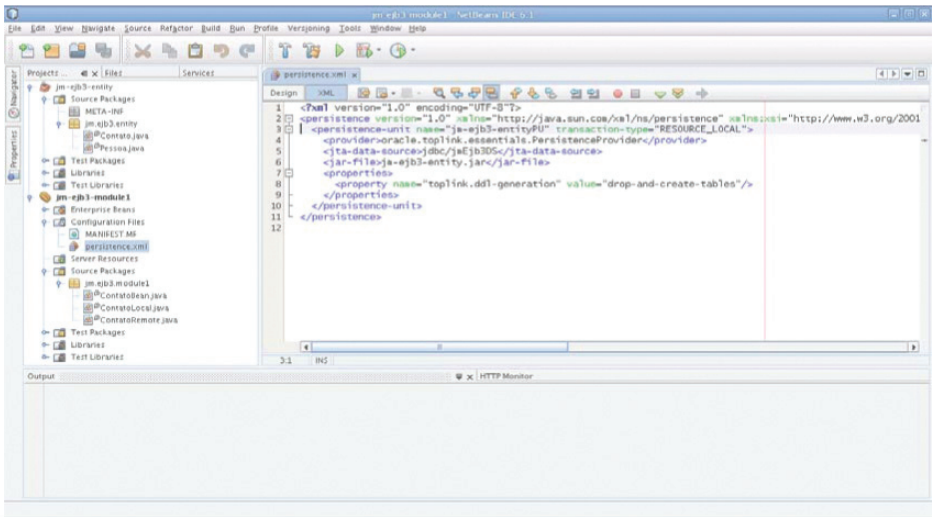


Figura 4. Estrutura da aplicação EJB.

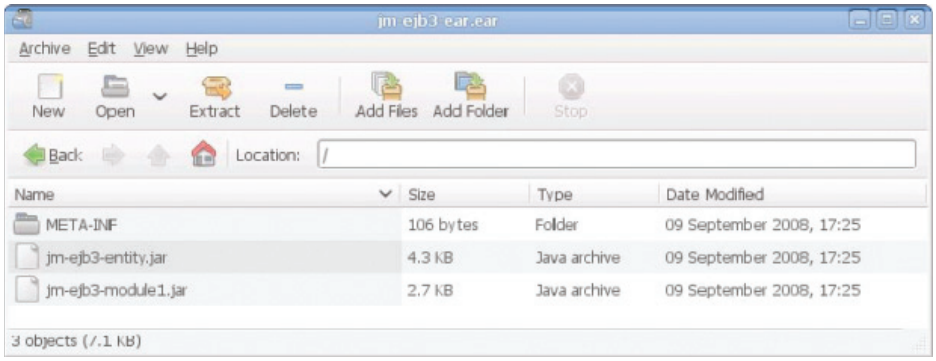


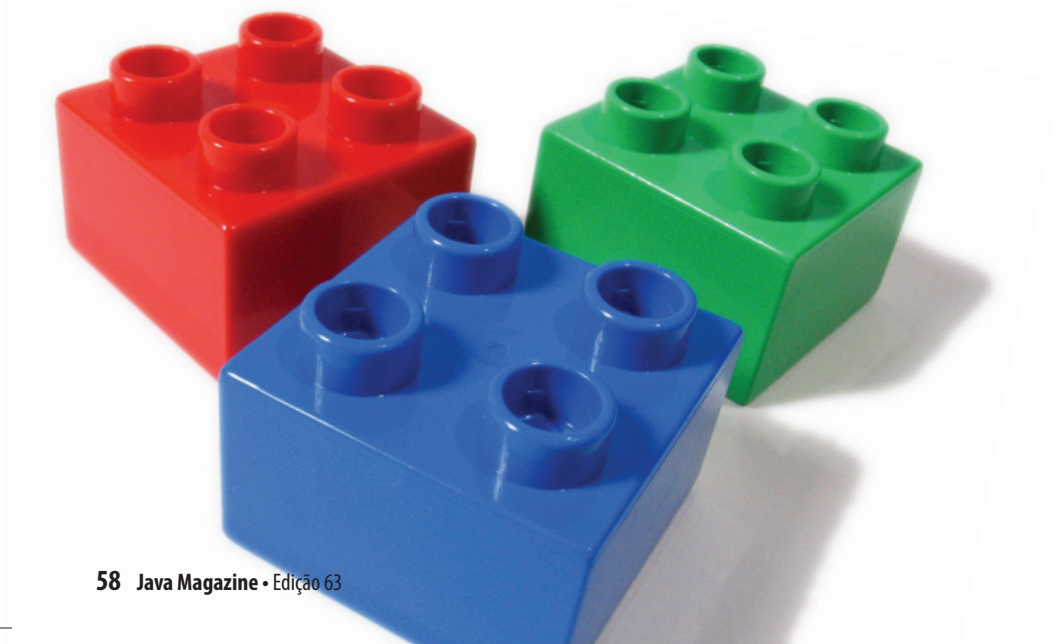
Figura 5. Estrutura do arquivo EAR.

Listagem 4 . Uso do design pattern Session Facade.

```
public class BazarControllerServlet extends HttpServlet {

  @EJB
  private ItemManager itemManager;

  @EJB
  private CategoriaManager categoriaManager;

  public void doPost(HttpServletRequest req, HttpServletResponse res) {
    // registro de um novo item/categoria
    ...
    categoriaManager.addCategoria(categoria);
    Item item = itemManager.createItem();
    item.setCategoria(categoria);
    ...
  }
}
```

Class Type	Injection
Servlets, filters, event listeners	Yes
JSP tag handlers, library event listeners	Yes
JSF managed beans	Yes
Helper classes, JSPs	No

Figura 6. Injeção automática de recursos em classes gerenciadas pelo container Java EE.



Implementando com Patterns: A saga dos Design Patterns

Alguns design patterns são famosos e estão sempre sendo implementados em quase todas as aplicações com EJB. Porém, alguns patterns tiveram sucesso com versões antigas da tecnologia EJB, mas se tornaram desnecessários com o EJB 3.

Vamos explicar a utilização dos quatro patterns escolhidos:

• ***Data Transfer Object (DTO):*** Este pattern foi criado para resolver problemas de tráfego de informações para fora do container EJB. Os velhos Entity Beans não podiam ser trafegados para fora do container EJB.

A idéia, na época, foi criar uma nova camada de objetos, que iria conter informações necessárias para ser transmitida ao *client*. Em Java, cria-se uma classe com atributos necessários para armazenar as informações que serão lidas por outros objetos. Na arquitetura EJB 3 isto **não** é mais necessário. Os objetos (POJOs) *podem ser* trafegados para fora do container. O ciclo de vida destes objetos inclui os estados *new, managed, detached* e *removed*. Muitas aplicações contêm classes DTO e a desculpa para o contínuo uso delas é: "expor os dados a outras camadas de uma maneira mais simples". DTO em EJB 3 é um **anti-pattern**, e por isso, não deve ser utilizado para encobrir *problemas* de uma aplicação não orientada a objetos: *bad-smell-design*;

• ***Session Facade:*** Este pattern é um dos mais utilizados, contudo, deve-se aplicá-lo com sabedoria. Quando há múltiplos acessos aos EJBs, dois problemas podem ser encontrados:

  ○ Um *client* acessa mais de um EJB para realizar um processo de negócio. Este problema é comum, e deve ser resolvido o mais rápido possível, pois, as camadas clientes serão acopladas ("arquitetura aranha") pela dependência de acesso a estes EJBs. A definição deste pattern implica na construção de subsistemas para manterem o acesso único às diversas partes da aplicação (**Listagem 4**).

Vejam que sem a implementação de um Session Facade, a *servlet* BazarControllerServlet fica dependente de dois EJBs,

e a lógica de negócio, para registro de um novo item/categoria, fica distribuída entre a *servlet* e os EJBs. Outro problema que pode ser encontrado neste cenário, caso os EJBs sejam remotos, é a chamada de múltiplos métodos (via RMI), pois, se forem muitas, pode causar perda de performance da aplicação.

◦ Múltiplos acessos a EJBs Stateless Session Bean para realizar um processo de negócio. Isto é, muitos EJBs são implementados como *Stateless Session Bean*, porém, a lógica de negócio é na verdade um processo contínuo de chamada a métodos de um ou mais EJB(s), que, também, podem ser dependentes de outros EJBs. Estes acessos identificam a necessidade de um EJB *Stateful Session Bean*. Implemenar um *Session Facade Stateful Session Bean* é estender o contexto (persistência, segurança, transação) pelo ciclo de vida do processo de negócio, e ao término do mesmo, ser removido de uma forma onde todas as dependências sejam também removidas do container EJB. Deste jeito a lógica de negócio é realizada em acessos a um EJB (Stateful Session Bean) que faz acesso a outros EJBs, propagando assim, o contexto para todos os recursos gerenciados pelo container EJB.

Portanto, a implementação de um EJB *Session Facade*, seja ele *Stateless* ou *Stateful*, implica em um serviço responsável por deter a lógica de um **processo de negócio**. Este serviço pode ser denominado, utilizando-se um conceito Orientado a Serviço (SOA), de web service. A forma mais fácil de criar serviços é através de uma modelagem orientada a objetos, onde componentes são expostos ao contexto corporativo (heterogêneo), através de novas camadas – *Session Facade* – que são implementadas para eles, formando-se um serviço de negócio.

• ***Service Locator:*** Na versão EJB 2.x, este pattern foi muito utilizado para encapsular a lógica de acesso ao contexto JNDI para recuperação de recursos (EJBs, DataSources, etc).

A nova versão da tecnologia EJB 3 suporta o conceito de injeção de dependência, implementado pela plataforma Java EE 5.

Listagem 5 . Implementação genérica do pattern DAO com JPA (exemplo retirado do livro Java Persistence with Hibernate).

```
public interface GenericDAO<T, Serializable> {
  // Métodos CRUD (persist, remove, findAll, get)
}

public abstract class GenericEjb3DAO<T, PK extends Serializable>
  implements GenericDAO<T, PK> {

  private Class<T> entityBeanType;

  private EntityManager entityManager;

  public GenericEjb3DAO() {
    this.entityBeanType = (Class<T>) ((ParameterizedType)
     super.getClass().getGenericSuperclass()).getActualTypeArguments()[0];
 }

  @PersistenceContext
  public void setEntityManager(EntityManager entityManager) {
    this.entityManager = entityManager;
  }

 ...

}
```

Listagem 6 . Especificando um contexto de persistência estendido.

```
@Stateful
public class ContatoBean implements ContatoRemote, ContatoLocal {

  @PersistenceContext(type=PersistenceContextType.EXTENDED)
  private EntityManager entityManager;

  ...

}
```

Este pattern deve ser utilizado em classes não gerenciadas pelo container Java EE, ou seja, onde recursos não podem ser injetados automaticamente (observe a **Figura 6**).

Note que os recursos não são injetados automaticamente em classes *helpers* e *JSPs*. Nestes casos, o uso do pattern *Service Locator* é necessário.

Em muitos casos, EJBs necessitam de acesso ao contexto EJB. Para estas situações, é recomendável a injeção do contexto javax.ejb.EJBContext, ou, mais especificamente, das subclasses **javax.ejb.SessionContext** (para *Session Beans*) e **javax.ejb.MessageDrivenContext** (para *Message Driven Beans*). A classe **EJBContext** comporta o método **lookup()**, que recupera a referência de objetos na árvore JNDI, mas, como Java EE 5 suporta injeção de dependência somente de objetos gerenciados pelo container EJB, para outros casos, o uso do método **lookup()** pode ser necessário. Outra alternativa é utilizar um framework que permita a injeção de objetos arbitrários em componentes EJBs, por exemplo, o Spring.

• ***Data Access Object (DAO):*** Este pattern é muito utilizado para encapsular a lógica de acesso a repositórios de dados (arquivos, banco de dados, etc.). Com a injeção automática de objetos gerenciados pelo container EJB, o acesso aos recursos fica mais fácil e direto para implementar componentes, que por exemplo, persistam objetos em bancos de dados. A injeção automática de **EntityManager** em EJBs é constante, entretanto, não é uma boa prática expor a lógica de acesso, mesmo sendo injeção pelo container em classes de negócio.

Uma boa prática é implementar DAOs como *Stateless Session Bean*, pois cada método não necessita de um estado, apenas de um **EntityManager**. Sendo assim, vamos implementar um DAO genérico utilizando JPA, como pode ser visto na **Listagem 5**.

Vejam, que a implementação abstrata da classe **GenericEjb3DAO** permite a injeção automática do **EntityManager** através da anotação

Listagem 7 . Criação de um interceptor para realizar log dos métodos de um EJB.

```
public class LogInterceptor {

  @AroundInvoke
  public Object log(InvocationContext invocationContext) throws Exception {
    Logger logger = Logger.getLogger(invocationContext.getTarget().getClass().getName());

    logger.info("Método [" + invocationContext.getMethod().getName() + "] chamado!");
    logger.info("Argumentos do método:");
    for (Object arg : invocationContext.getParameters()) {
      logger.info("Argumento do método: " + arg);
    }

    return (invocationContext.proceed());
  }

}
```

Listagem 8 . Criação de um interceptor para realizar log dos métodos de um EJB.

```
public class LogInterceptor {

  @AroundInvoke
  public Object log(InvocationContext invocationContext) throws Exception {
    Logger logger = Logger.getLogger(invocationContext.getTarget().getClass().getName());

    logger.info("Método [" + invocationContext.getMethod().getName() + "] chamado!");
    logger.info("Argumentos do método:");
    for (Object arg : invocationContext.getParameters()) {
      logger.info("Argumento do método: " + arg);
    }

    Object result = invocationContext.proceed();

    logger.info("Retorno do método: " + result);

    return (result);
  }

}
```

Listagem 9 . Criação de um interceptor para realizar log em nível de métodos de um EJB.

```
@Stateless
public class ContatoBean implements ContatoRemote, ContatoLocal {

  @PersistenceContext
  private EntityManager entityManager;

  @Interceptors(LogInterceptor.class)
  public Contato criarContato(String nome, String email) {
    Contato novoContato = new Contato();
    novoContato.setNome(nome);
    novoContato.setEmail(email);
    return (this.entityManager.merge(novoContato));
  }

}
```

Listagem 10 . Criação de um interceptor para realizar logs um EJB

```
@Stateless
@Interceptors(LogInterceptor.class)
public class ContatoBean implements ContatoRemote, ContatoLocal {

  @PersistenceContext
  private EntityManager entityManager;

  public Contato criarContato(String nome, String email) {
    Contato novoContato = new Contato();
    novoContato.setNome(nome);
    novoContato.setEmail(email);
    return (this.entityManager.merge(novoContato));
  }

}
```

@PersistenceContext, marcada no método **setEntityManager()**. Esta anotação poderia estar marcada no atributo **entityManager**, porém, caso a aplicação estivesse fora de um container EJB 3.0, a injeção deveria ser feita manualmente.

Controle Transacional: CMT ou BMT?

A tecnologia EJB 3 oferece o serviço de transação, baseada na JTA (*Java Transaction API*), para *beans* gerenciados pelo container. Assim como no caso da JPA, isso é feito usando metadados (annotations) ao invés de um modelo programático, onde os *beans* podem participar das transações JTA e controlar o estado transacional de cada método de negócio do componente EJB.

O modelo transacional, da tecnologia EJB 3, traz dois tipos de gerenciamento de transações: JTA e não-JTA, mais conhecido como *resource-local*. Transações *resource-local* são restritas a bancos de dados, o que pode resultar em otimização no desempenho, evitando sobrecarga de uma transação distribuída. Ao escolher JTA como modelo transacional, o gerenciamento dos métodos de negócio do componente EJB é de responsabilidade do container.

Escolher entre transações gerenciadas pelo container – CMT (*Container-Managed Transaction*) ou pela própria aplicação – BMT (*Bean-Managed Transaction*) – implica em como recuperar uma instancia da classe **EntityManager**. Caso for injetado automaticamente (ou realizado um lookup via JNDI) dentro do container EJB, o contexto de transação do componente será gerenciado. Para deixar o controle de transação pela aplicação, a responsabilidade é do desenvolvedor, de criar um objeto da classe **EntityManager** a partir da classe **EntityManagerFactory**.

Definir o estilo de transação a ser adotado para o desenvolvimento da aplicação é importante, porém, definir o escopo da transação é que merece mais atenção. Pode-se classificar o escopo da transação em dois tipos:

• ***Transaction-Scoped Persistence Context:*** Este contexto de persistência é padrão e está associado a transação. A partir do

momento que um **EntityManager** é injetado, usando a anotação **@PersistenceContext**, o contexto de transação está definido;

• ***Extended Persistence Context:*** Este contexto de persistência é estendido pelo ciclo de vida da transação. Para implementar um contexto extendido, basta usar o atributo **type** da anotação **@PersistenceContext**, veja a **Listagem 6**.

Este contexto deve ser utilizado apenas em *EJB Stateful Session Beans*, pois o contexto de transação estendido é propagado pelo ciclo de vida do componente, garantindo cache e acesso ao domínio da aplicação (controle das entidades de negócio) através das múltiplas chamadas aos métodos do EJB.

Portanto, deixar a responsabilidade do controle de transação por parte do container EJB – CMT – é uma boa prática, mas, caso o nível de controle for de granularidade baixa, pode-se usar, em conjunto, o controle via BMT, garantindo assim um gerenciamento transacional mais robusto e evitando problemas no desenvolvimento da aplicação.

Controle de Logs e Exceções

O objetivo deste tópico é mostrar o uso de EJB Interceptors para facilitar o controle de exceções e logs e abrir novas possibilidades para implementações mais flexíveis e robustas para a aplicação.

Realizar controle de logs e gerenciar exceções sem precisar alterar o código-fonte dos métodos de negócio é uma tarefa difícil, e às vezes chata!

A tecnologia EJB 3 oferece uma ferramenta chamada – *interceptors* – para implementar conceitos referentes a infra-estrutura.

Interceptors implementam os conceitos do paradigma AOP (*Aspect-Oriented Programming*), que podem ser descritos pelo termo *crosscutting concerns*. Este termo significa que a aplicação terá um corte lógico – "horizontal" – ao processamento das regras de negócio. Todo código comum, que refere-se à infra-estrutura, como por exemplo, log, auditoria e exceção, pode ser implementado utilizando este conceito.

A arquitetura EJB 3 oferece suporte para interceptar chamadas aos métodos de negócio e callbacks de um EJB. Com esta ferramenta, podemos interceptar os métodos realizando log do que está sendo enviado e retornado, e controlando as exceções de infra-estrutura e negócio da aplicação.

Para implementar um interceptor, utilizando a tecnologia EJB 3, crie uma classe, que para o nosso exemplo vamos chamar de **LogInterceptor**, e adicione o método chamado **log()**. Para estar aderente à especificação EJB 3, interceptors devem ter como argumento no método a classe **InvocationContext**, neste caso o método ficará **log(InvocationContext context)**. Anote o método com **@AroundInvoke** (do pacote **javax.interceptor**). Esta anotação identifica o método que irá interceptar as chamadas aos EJBs registrados a este interceptor (**Listagem 7**).

Veja que o código da classe **LogInterceptor** irá realizar log do que está sendo passado aos métodos dos EJBs, porém, caso queira realizar log após a execução do método do EJB, antecipe a execução da operação **invocationContext.proceed()**. Este método procede com a execução do método do EJB, veja a **Listagem 8**.

Agora que temos o interceptor implementado, vamos aplicá-lo a um EJB. Para registrar o interceptor ao EJB, utilize a anotação **@Interceptors**. Esta anotação recebe como argumento um conjunto de classes que contém um método marcado com a anotação **@AroundInvoke**. Esta anotação pode ser utilizada para marcar a classe, interceptando todos os métodos públicos no EJB, e os métodos do EJB, controlando apenas aqueles que foram marcados por esta anotação. Observe a **Listagem 9**.

Note que o interceptor **LogInterceptor** foi aplicado, através da anotação **@Interceptors**, ao método **criarContato()**. Neste caso, apenas este método será controlado pelo interceptor. Caso todos os métodos do EJB necessitem de log, aplique a anotação **@Interceptors** na classe do EJB, de acordo com a **Listagem 10**.

Com esta implementação, todos os métodos registrados nas interfaces **@Local** e/ou **@Remote** do EJB serão controlados pelo interceptor **LogInterceptor**.

Já sabemos como criar um interceptor e controlar os logs em nível de classe e métodos dos EJBs. Porém, precisamos controlar as exceções, tanto de negócio como de infra-estrutura.

Nota do DevMan

CMT (Container-Management Transaction): Serviço de transação padrão para EJBs Session Bean e Message Driven Beans. O container gerencia a transação, iniciando-se a partir da chamada ao método do EJB, realizando *commit* ao fim da execução ou *rollback* para casos de exceção.

Nota do DevMan

BMT (Bean-Management Transaction): Serviço de transação manual para EJBs. O gerenciamento da transação é feita pelo container, porém a chamada aos métodos "begin", "commit" e "rollback" são realizadas pelo *bean*. A vantagem de uso deste serviço é a granularidade do controle de transação na aplicação, onde métodos são chamados dentro de um processo de negócio, e ao final da execução deste processo é realizado o "commit".

Exceções de infra-estrutura são mais fáceis de controlar, pois acontecem por problemas em persistência (arquivos e banco de dados), segurança, transações, comunicação, etc.

Já exceções de negócio acontecem no processamento das regras de negócio, ou seja, são referentes aos estados que os objetos de negócio irão obter. Estas exceções são particulares ao contexto da aplicação. Portanto, não há uma lógica definida de saber o que deve ser feito, a não ser que crie um modelo que permita que as exceções de negócio, através de uma classe abstrata, por exemplo, possam ter inteligência para resolver o controle de tais eventos.

O que pode ser feito para amenizar este "problema", é usar interceptors para controlar as exceções de infra-estrutura e realizar logs de exceções de negócio com um tipo específico. Por exemplo, criaremos a classe abstrata **BusinessException** para identificar estas exceções da aplicação (veja a **Listagem 11**).

Veja que a classe **BusinessException** apenas identifica as exceções de negócio, cuja implementação irão herdar suas características.

Listagem 11 . Criação de um interceptor para realizar controle de exceção nos EJBs.

```
-- Classe abstrata para identificar exceções de negócio.
public abstract class BusinessException extends Exception { ... }

-- Interceptor para controlar responsável pelo controle das exceções.
public class ExceptionHandlerInterceptor {

  @AroundInvoke
  public Object handleException(InvocationContext invocationContext) throws Exception,
    BusinessException {
    final String targetClassName = invocationContext.getTarget().getClass().getName();
    final String methodName = invocationContext.getMethod().getName();

    Logger logger = Logger.getLogger(targetClassName);

    try {

      return (invocationContext.proceed());

    } catch (BusinessException businessException) {

      // exceções de negócio
      logger.severe(businessException.getMessage());

      throw (businessException);

    } catch (Exception exception) {
      // exceções de infra-estrutura
      logger.severe("Erro ao realizar operação [" +
        methodName + "] em [" + targetClassName + "]");
      throw (exception);
    }

  }
}
```

Listagem 12 . Implementação da abstração BusinessException e controle de exceção no EJB ContatoBean.

```
-- Classe que exceções de negócio.

public class NomeContatoInvalidoException extends BusinessException {

  private String nome;

  public NomeContatoInvalidoException(String nome) {
    this.nome = nome;
  }

  @Override
  public String getMessage() {
    return ("O nome informado para o contato é inválido: " + this.nome);
  }

}

-- EJB ContatoBean

@Stateless
@Interceptors(LogInterceptor.class)
public class ContatoBean implements ContatoRemote, ContatoLocal {

  @PersistenceContext
  private EntityManager entityManager;

  @Interceptors(ExceptionHandlerInterceptor.class)
  public Contato criarContato(String nome, String email) throws NomeContatoInvalidoException {

    if ((nome == null) || ("".equals(nome))) {
      throw (new NomeContatoInvalidoException(nome));
    }

    Contato novoContato = new Contato();
    novoContato.setNome(nome);
    novoContato.setEmail(email);
    return (this.entityManager.merge(novoContato));
  }
}
```

Vamos implementar a exceção de negócio – **NomeContatoInvalidoException** – que herda a classe **BusinessException** e alterar o código do método **criarContato()**, no EJB **ContatoBean**, para que o interceptor realize o controle de exceção deste método (veja a **Listagem 12**).

O interceptor **ExceptionHandlerInterceptor** foi aplicado ao método **criarContato()** e irá realizar o controle de exceções disparadas por este método.

Esta é apenas uma alternativa para controlar exceções de negócio. Outra alternativa é criar anotações que irão marcar os métodos dos EJBs, informando ao interceptor os tipos de exceção a serem controladas por ele. Controles de exceções de negócio são particulares para cada aplicação, então devemos achar soluções "limpas" para não poluirmos o modelo de negócio da aplicação.

Testes Unitários: Testando EJB como POJOs

Nas versões anteriores à tecnologia EJB 3, realizar testes unitários era quase impossível. Para realizar estes testes, deve-se realizar *deployment* dos EJBs, e, a partir daí, os testes unitários devem ter uma configuração para capturar o contexto JNDI, obtendo acesso aos EJBs publicados no container EJB.

Já com a versão EJB 3, os componentes são apenas POJOs e as regras de negócio podem ser testadas de uma forma simples, sem ter que configurar ou realizar deployment de recursos para execução dos testes.

Métodos que não utilizam objetos injetados automaticamente pelo container EJB podem ser testados utilizando a técnica de teste unitário, com a ferramenta *JUnit*.

Métodos que necessitam de injeção de objetos (por parte do container) e informações metadata aplicadas às entidades, podem ser testadas utilizando a ferramenta ***Ejb3Unit*** (*http://ejb3unit.sourceforge.net/*). Esta ferramenta é uma extensão do JUnit e facilita a execução de testes unitários a EJBs implementados com a versão 3.0.

A tecnologia EJB 3 facilitou em termos de codificação o desenvolvimento de EJBs, e

muitas ferramentas foram disponibilizadas simplificando a tarefa de testes destes componentes em uma aplicação.

Conclusão

A tecnologia EJB 3.0 trouxe melhorias na codificação de componentes corporativos e na criação de serviços web (*WebServices*).

O futuro da tecnologia EJB é implementar uma arquitetura que facilite o desenvolvimento de componentes, baseando-se cada vez mais em POJOs.

A **JSR 318: Enterprise JavaBeans 3.1** (*http://jcp.org/en/jsr/detail?id=318*), descreve a nova especificação da tecnologia EJB. O objetivo desta especificação será o core da arquitetura – os *Session Beans* e os *Message Driven Beans* – o que na versão EJB 3.0 foi a simplicidade na codificação de componentes EJBs.

Outra evolução da tecnologia EJB será a implementação da **JSR 299: Web Beans** (*http://jcp.org/en/jsr/detail?id=299*). O objetivo desta especificação é trazer o conceito de que EJBs podem ser acessados a partir da camada de apresentação, implementada com a tecnologia JSF (*Java Server Faces*).

A nova versão da plataforma Java EE, definida pela **JSR 316: Java™ Platform, Enterprise Edition 6 (Java EE 6) Specification** (*http://jcp.org/en/jsr/detail?id=316*) irá incluir as especificações citadas acima, garantindo assim, uma plataforma mais robusta e simples para o desenvolvimento de aplicações Java EE.

Dessa forma, o uso da tecnologia EJB tende a crescer e novos horizontes serão alcançados para a evolução de aplicações e serviços corporativos. ●

Nota do DevMan

JNDI (Java Naming and Directing Interface): É uma API para acesso a serviços de diretórios, permitindo a pesquisa de um recurso através de um nome.

Nota do DevMan

JTA (Java Transaction API): Define um padrão de interfaces entre o gerenciador de transação e partes envolvidas em sistemas de transações distribuídos, tais como: servidor de aplicação e aplicações que utilizam transação.

Fábio Augusto Falavinha
fabio@summa-tech.com
é Consultor Java EE pela Summa Technologies do Brasil, atuando há mais de 8 anos com Java/Java EE. Bacharel em Ciência da Computação pela Faculdade Sumaré e pós-graduado em Gestão de Qualidade de Software pela faculdade Senac.

Dê seu feedback sobre esta edição!

A Java Magazine tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/javamagazine/feedback

jcp.org/en/jsr/detail?id=220
Página oficial da JSR Enterprise JavaBeans 3.0.

java.sun.com/javaee/overview/faq/ejb.jsp
FAQ sobre a tecnologia EJB 3.0.

javaworld.com/javaworld/jw-01-2006/jw-0130-pojo.html
Excelente artigo sobre a escolha de tecnologias para o desenvolvimento de aplicações Java EE.

www.ibm.com/developerworks/library/j-ejbexcept.html
Excelente artigo sobre controle de exceções.

java.sys-con.com/node/325149
Artigo referente a conceitos de transações na arquitetura EJB 3.

EJB3 in Action, Debu Panda, Reza Rahman e Derek Lane, Manning, 2007
Execelente livro sobre EJB3! Com muitos exemplos e explicações sobre todos os mecanismos que compõe o novo modelo de implementação.

Java Persistence with Hibernate, Christian Bauer, Gavin King, Manning, 2007
Este livro aborda excelentes conceitos sobre persistência com JPA.