Java Enterprise Edition

Professor Rodrigo Postai

Quem sou eu

- Formado em Bacharelado em Informática pela Universidade Estadual de Maringá – UEM
- Atuo no mercado de TI desde 2002 como desenvedor, coordenador de equipe, líder técnico/arquiteto de software
- Atuei como professor da graduação e atuo como professor de Pós-graduação
- Atuei com tecnologias Delphi, C#, VB.NET, VB6, Java (e seu mundo)
- SCJP – Sun Certified Java Programmer
- Sun Certified Web Component Developer for the Java Platform, Enterprise Edition 5 – SCWCD
- Sun Certified Business Component Developer Java EE Platform 5
- Oracle Certified Master, Java EE 5 Enterprise Architect I II e III (Completa)

Agenda

- Noções de arquitetura e JEE
- EJB3
  - Stateful, stateless, singleton, message-driven
- Testes automatizados

Noções de JEE e Arquitetura de Aplicações

Arquitetura de Aplicações

- Responde a questões de alto nível tais como:
  - Quais são as tecnologias fundamentais?
  - Quais suas partes e como se relacionam?
  - Como se integra a outros sistemas?
  - Como suporta os requisitos não-funcionais?
    - escalabilidade, confiabilidade, segurança, performance, flexibilidade, acessibilidade, etc.
- Uma arquitetura ruim pode causar
  - custos excessivos, baixa qualidade, atrasos
  - em último caso, o fracasso completo do projeto
  - em alguns casos, uma POC (Proof of Concept) pode ser necessária

Arquitetura de Aplicações: Requisitos NFs

- escalabilidade
  - capacidade de aumentar o *throughput* quando se incrementa o hardware
    - melhor uso do hardware
    - permite aumento gradual da carga com o tempo
- segurança
  - capacidade de garantir que dados/operações serão armazenados/disponibilizados exatamente como definido pela política da empresa
    - autenticação/autorização
    - privacidade
    - auditoria

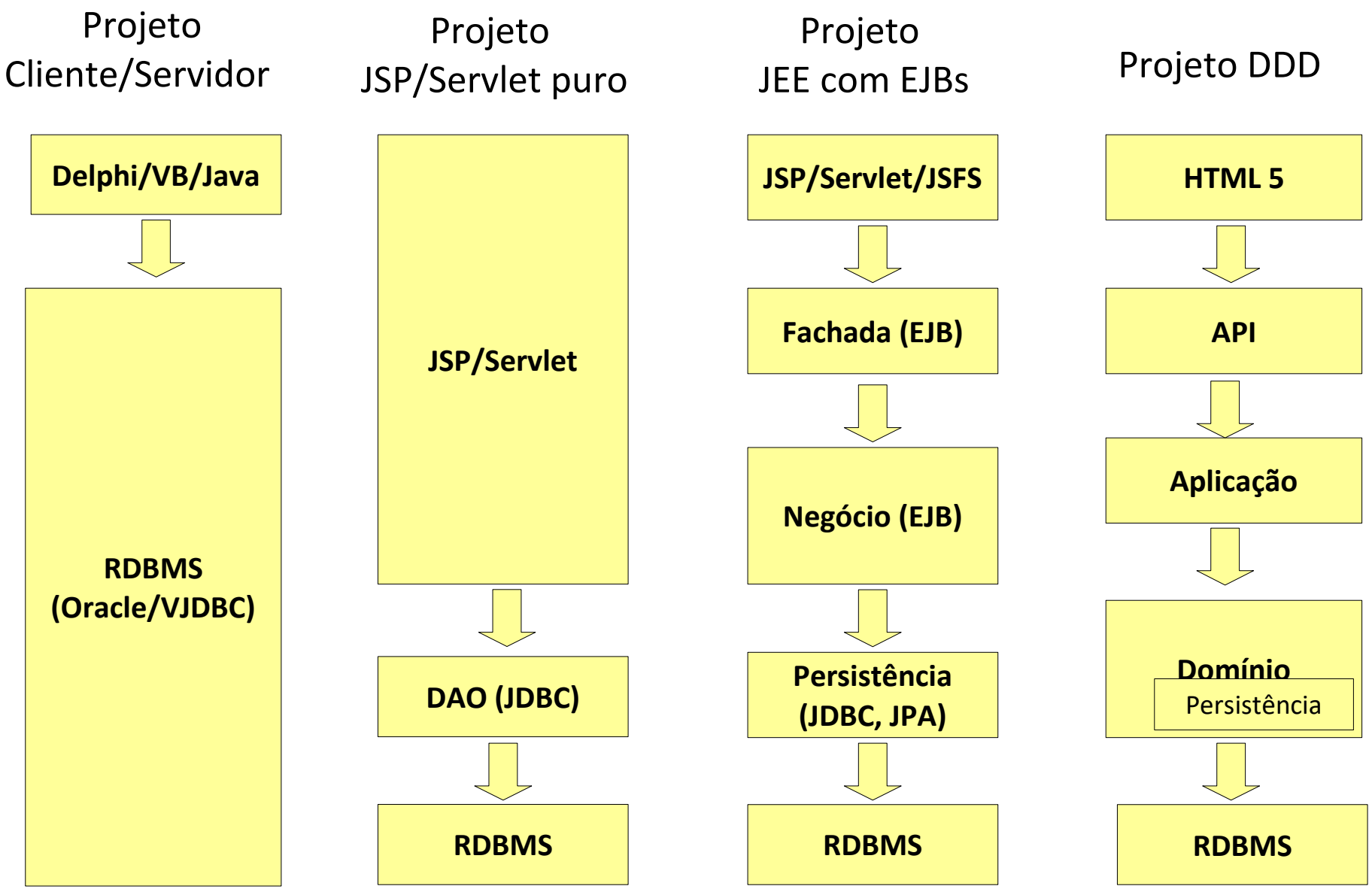
Arquitetura de Aplicações: Requisitos NFs

- confiabilidade
  - capacidade de garantir a consistência do estado do sistema a todo instante, mesmo em condições adversas
    - alterações devem ser transacionais
- disponibilidade
  - capacidade do sistema em estar operacional e acessível aos usuários
    - Sistema 24x7?
- desempenho
  - capacidade de executar operações em intervalos de tempo definidos
    - Medida em operações/tempo ou tempo médio de resposta

Arquitetura de Aplicações: Camadas

- A divisão em camadas facilita o desenvolvimento e manutenção
  - permite a criação de testes por camada
  - camadas se relacionam umas com as outras por uma “interface” que oculta detalhes de implementação
  - as dependências são sempre de cima para baixo
    - Camada de serviço jamais deve depender da camada de apresentação, por exemplo
  - cada camada pode ser potencialmente executada
    - em servidores separados
    - em mais de um servidor ao mesmo tempo (clustered)
- Há inúmeras combinações no mercado

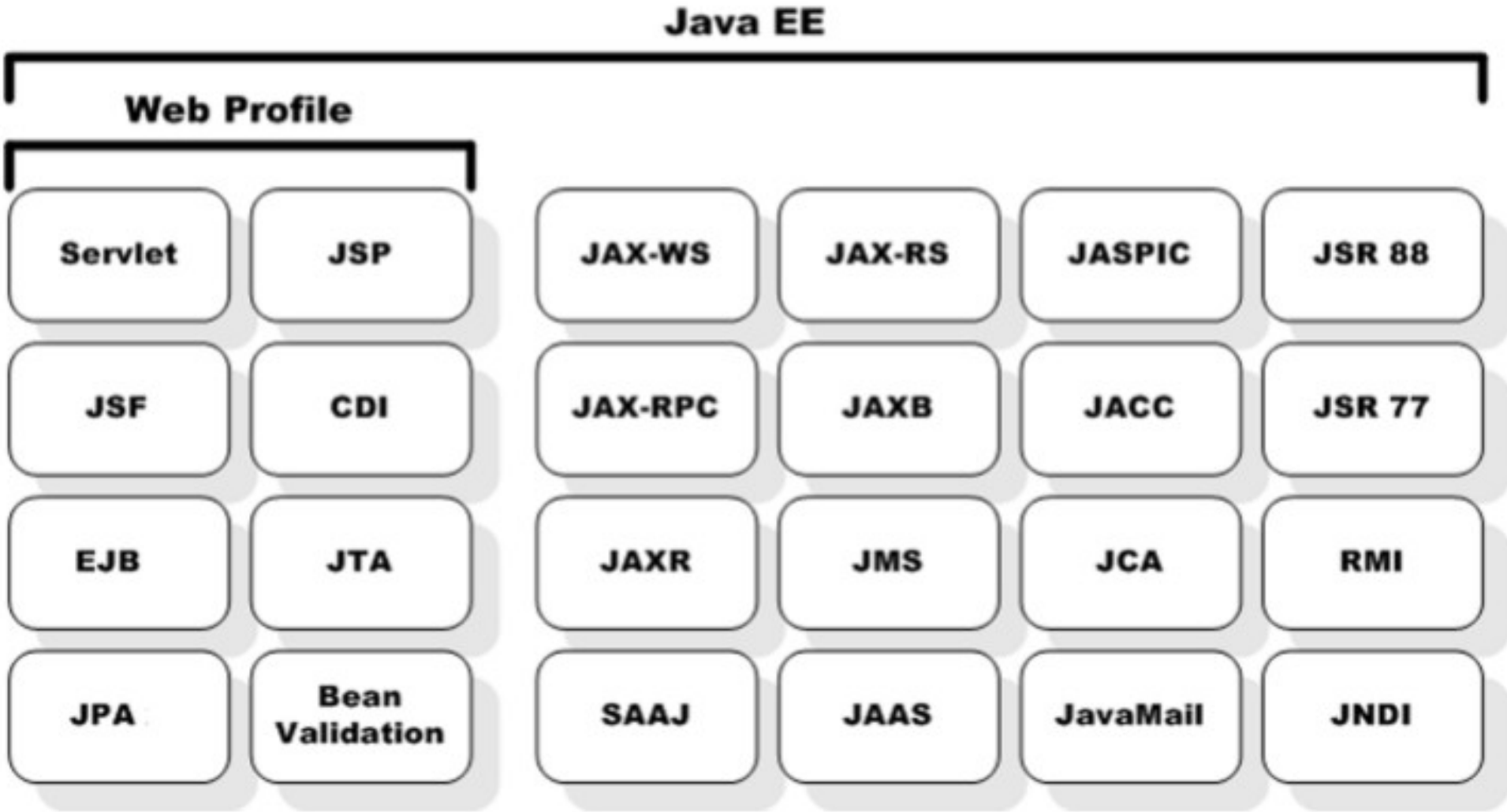
Arquitetura de aplicações: Exemplos



Java Enterprise Edition

- O que é?
  - Um conjunto de tecnologias que oferecem serviços para aplicações corporativas usando uma arquitetura multi-camadas
- É composta por uma grande gama de tecnologias interdependentes ditadas por especificações JSR onde participam os principais fornecedores
- É implementado por um Servidor de Aplicações
  - Alguns servidores suportam parcialmente

Especificações JEE 7



EJB 3.2 Lite

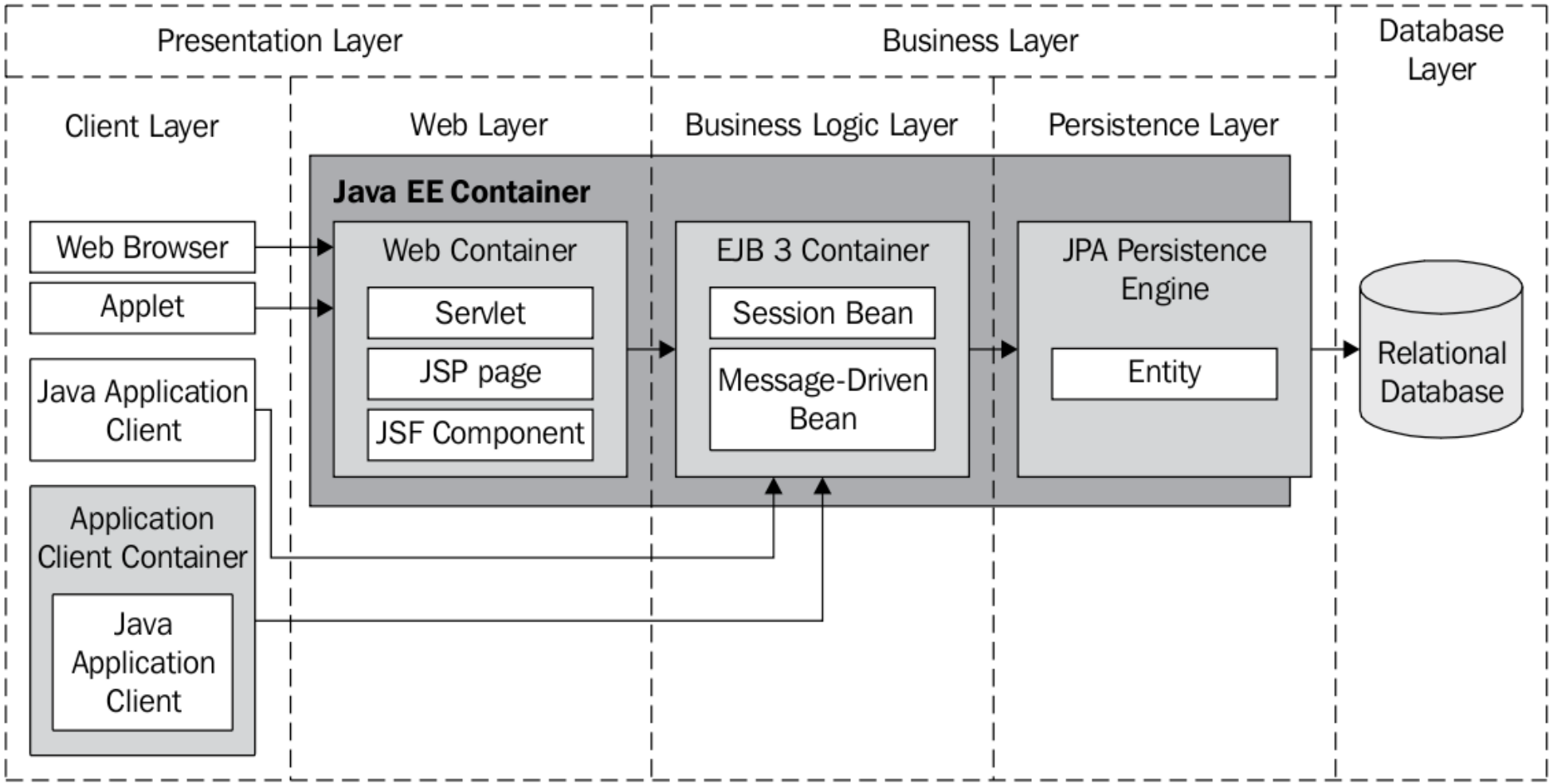
- Remove o suporte a EJB2
- O container pode ser muito mais leve
- Ideal para desenvolvimento WEB

Feature	EJB LIte	EJB
Stateless beans	☑	☑
Stateful beans	☑	☑
Singleton beans	☑	☑
Message-driven beans		☑
No interfaces	☑	☑
Local interfaces	☑	☑
Remote interfaces		☑
Web service interfaces		☑

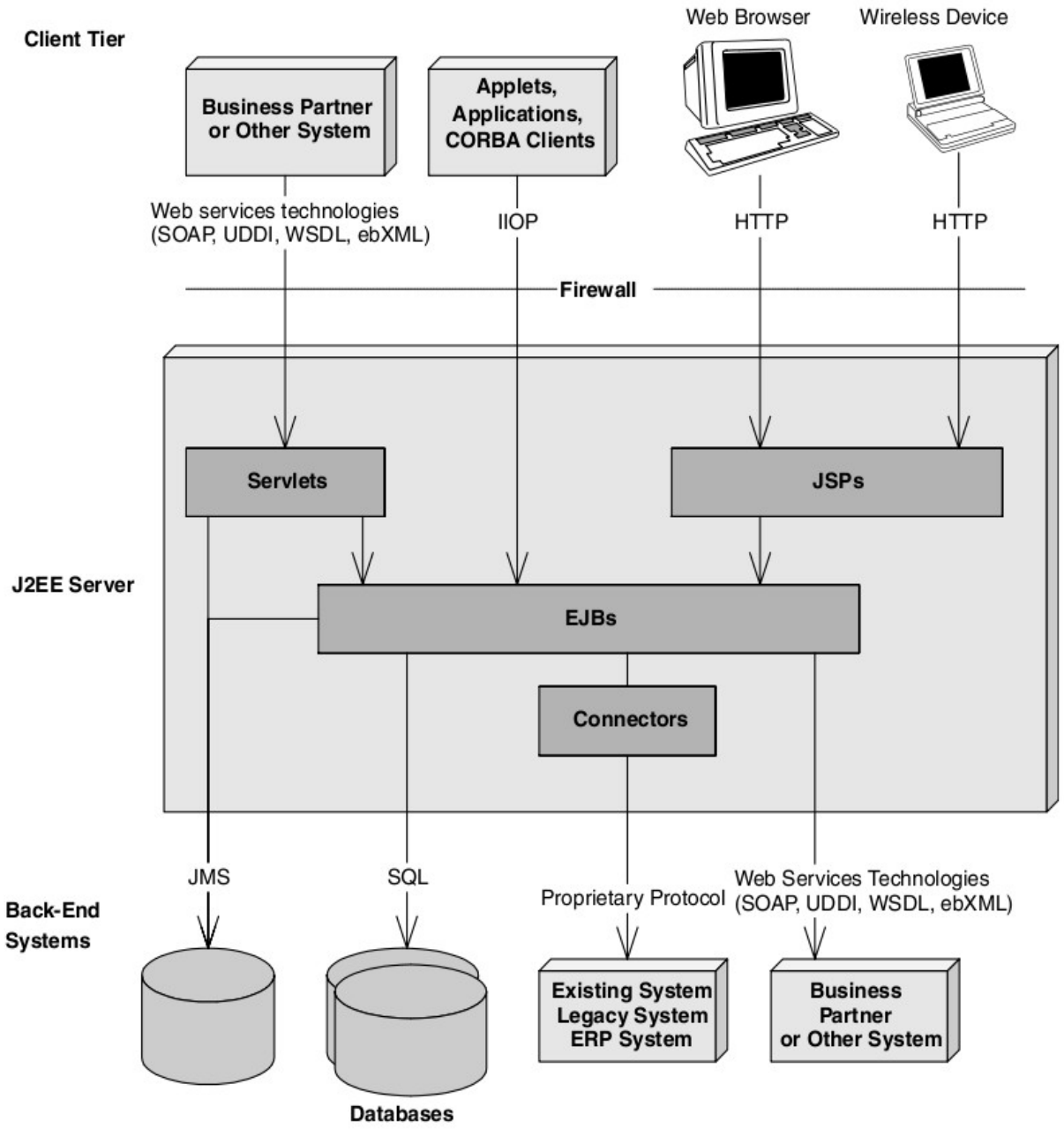
EJB 3.2 Lite

Feature	EJB LIte	EJB
Asynchronous invocation	☑	☑
Interceptors	☑	☑
Declarative security	☑	☑
Declarative transactions	☑	☑
Programmatic transactions	☑	☑
Timer service	☑	☑
EJB 2.x support		☑
CORBA interoperability		☑

Java Platform Enterprise Edition



Exemplo de deploy JEE



Introdução a EJB 3.2

EJB3: Características

- Simples de desenvolver
  - EJB2 era muito complexo
  - Uso intenso de annotations
  - Nova API para persistência (JPA)
  - Teste *out-of-container* possível pois EJBs são POJOs
- Padrão de mercado
  - Java Community Process
  - Inúmeras implementações, inclusive open source
    - JBoss AS, Wildfly, Glassfish, Oracle Weblogic, IBM Websphere, Oracle IAS, Jonas, Apache Geronimo

EJB3: Características

- Recursos principais
  - Injeção de dependências
  - Gerenciamento automático de transações distribuídas
  - Chamadas remotas via RMI
  - Interceptors AOP
  - EJBs podem ser exportados como WebServices
  - Segurança (autenticação/autorização)
  - Suporte a JMS (Java Message Service)
  - Gerenciamento automático de threads

EJB3: Características

- Outros recursos avançados
  - Clustering
  - Failover
  - Load balancing
  - Distributed caching
  - Monitoring

EJB3: Conceito

- O que são EJBs?
  - São componentes Java executados dentro de um container que oferece serviços como:
    - controle de transações distribuídas
    - chamadas remotas
    - injeção de dependência
    - segurança
    - disponibilização como Web Service
    - mensagens assíncronas
    - pooling de instâncias

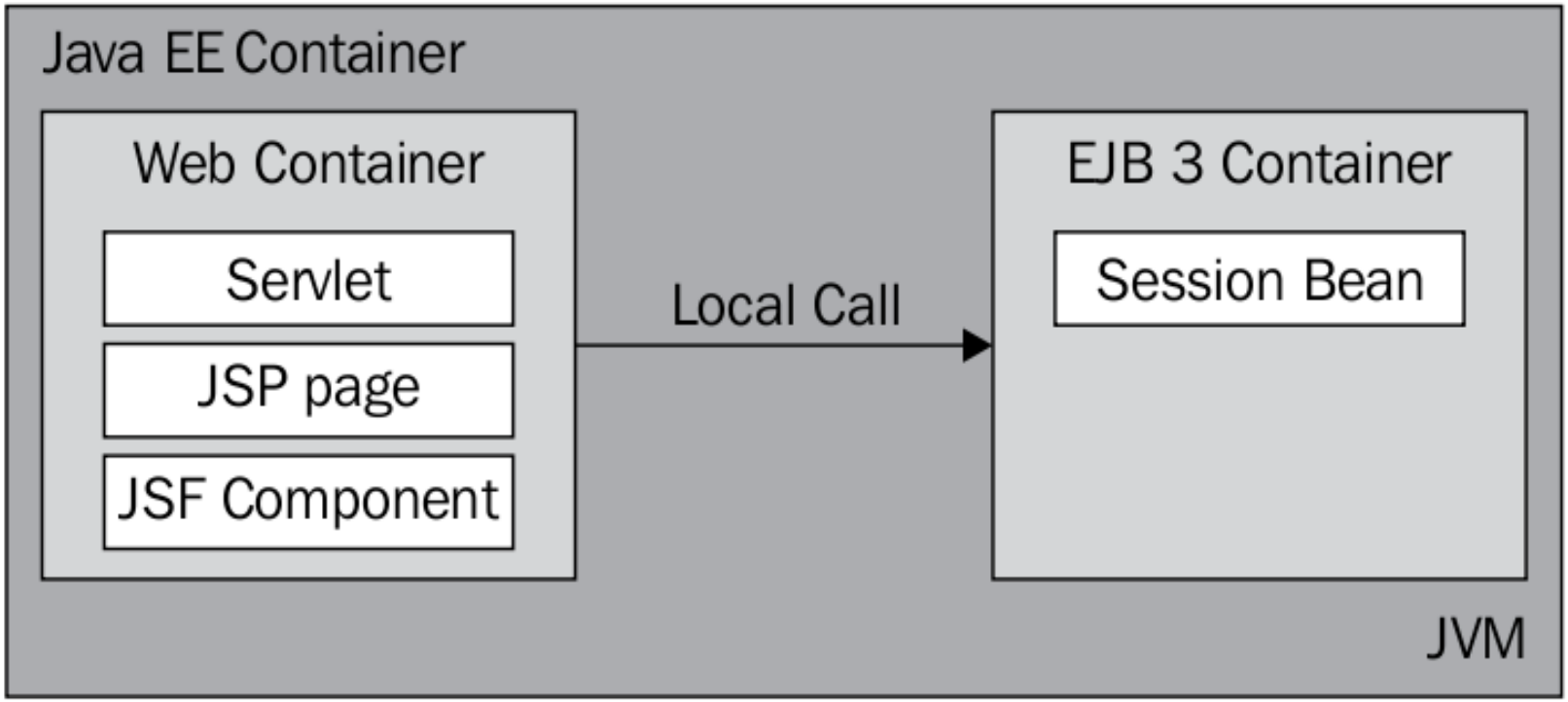
Por que usar EJB?

- Facilidade de uso
- Tecnologias integradas e completas (transações, segurança, mensageria, agendamento, chamadas distribuídas, processamento assíncrono, injeção de dependência, interceptadores)
- Padrão Java Aberto
- Grande variedade de fornecedores
- Clustering, Load Balancing e Failover
- Performance e Escalabilidade
- Versão Lite

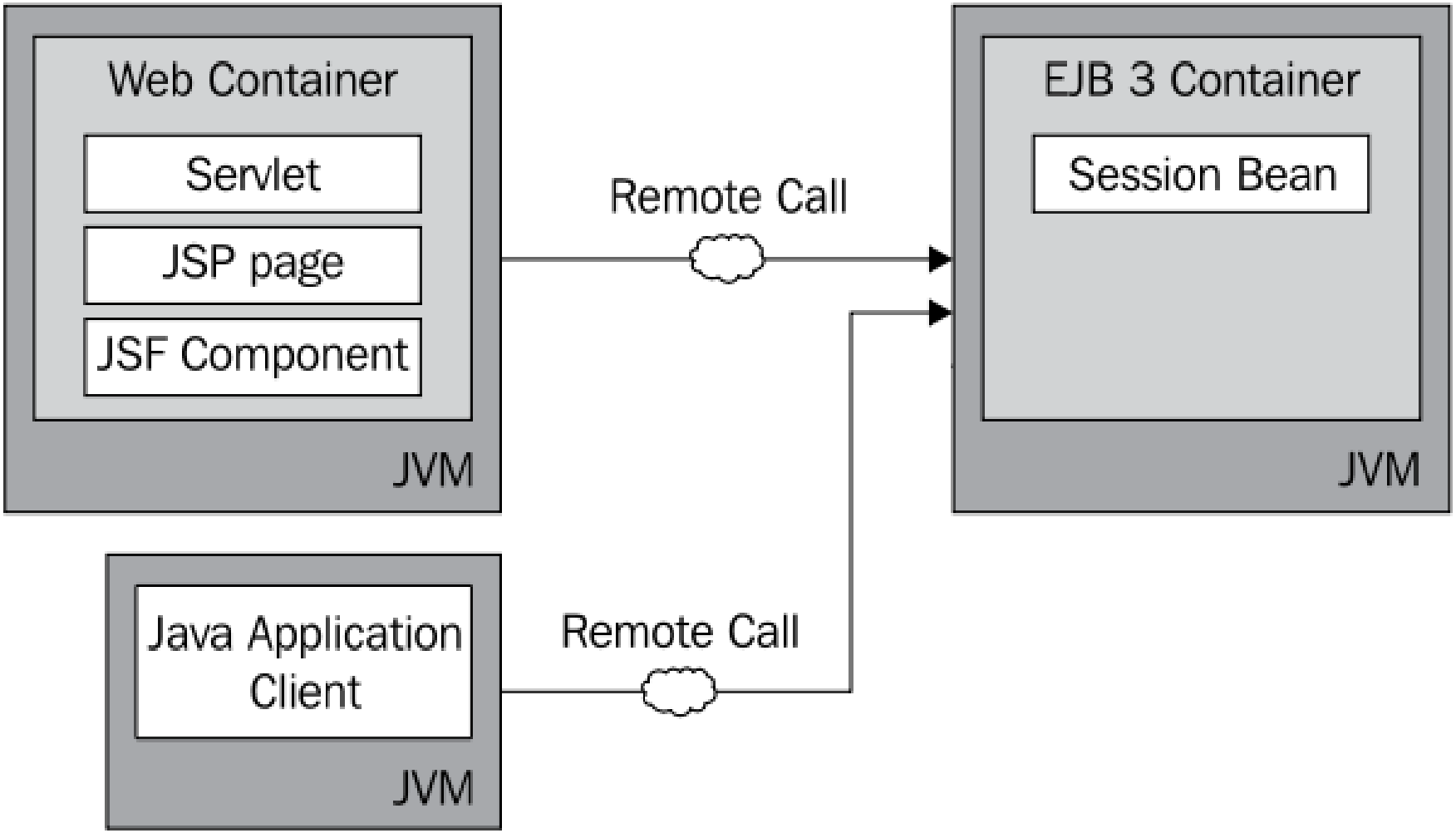
EJB3

- Tipos de beans
  - Session Beans
    - @Stateless (SLSB)
    - @Stateful (SFSB)
    - @Singleton
  - Message-driven Beans (MDB)
- Podem ser chamados de 3 formas
  - Síncrona local/remota (@Local/@Remote)
  - Assíncrona
- Entity Beans (JPA)
  - JPA 2 desconectado de EJB

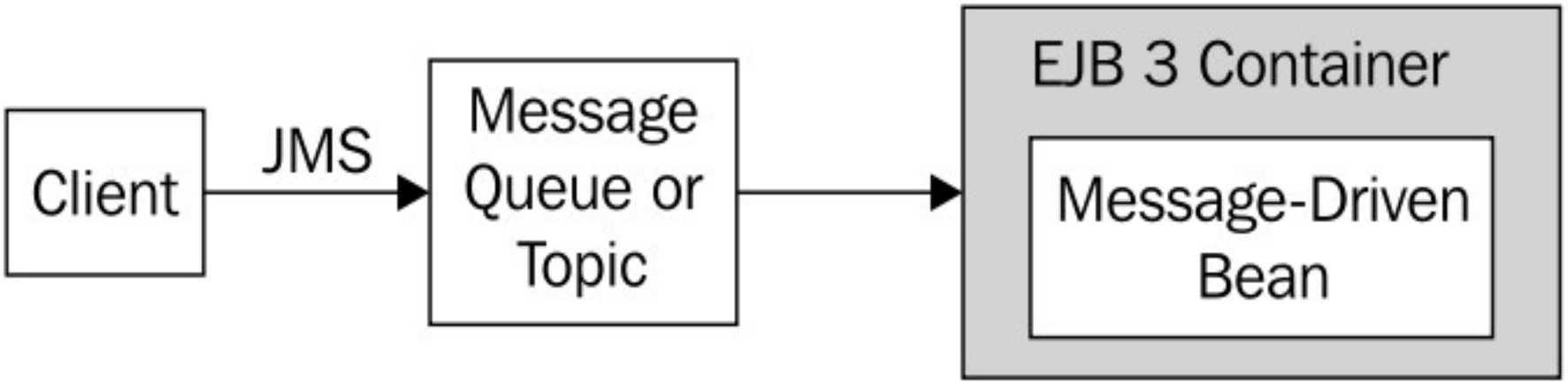
EJB3: Chamada Local



EJB3: Chamada Remota



EJB3: Chamada Assíncrona com MDB



Session Beans

EJB3: Session Beans

- O que são?
  - São componentes chamados por clientes com o propósito de realizar alguma operação que modela algum aspecto funcional da aplicação
    - implementam comportamentos descritos nos casos de uso

EJB3: Session Beans

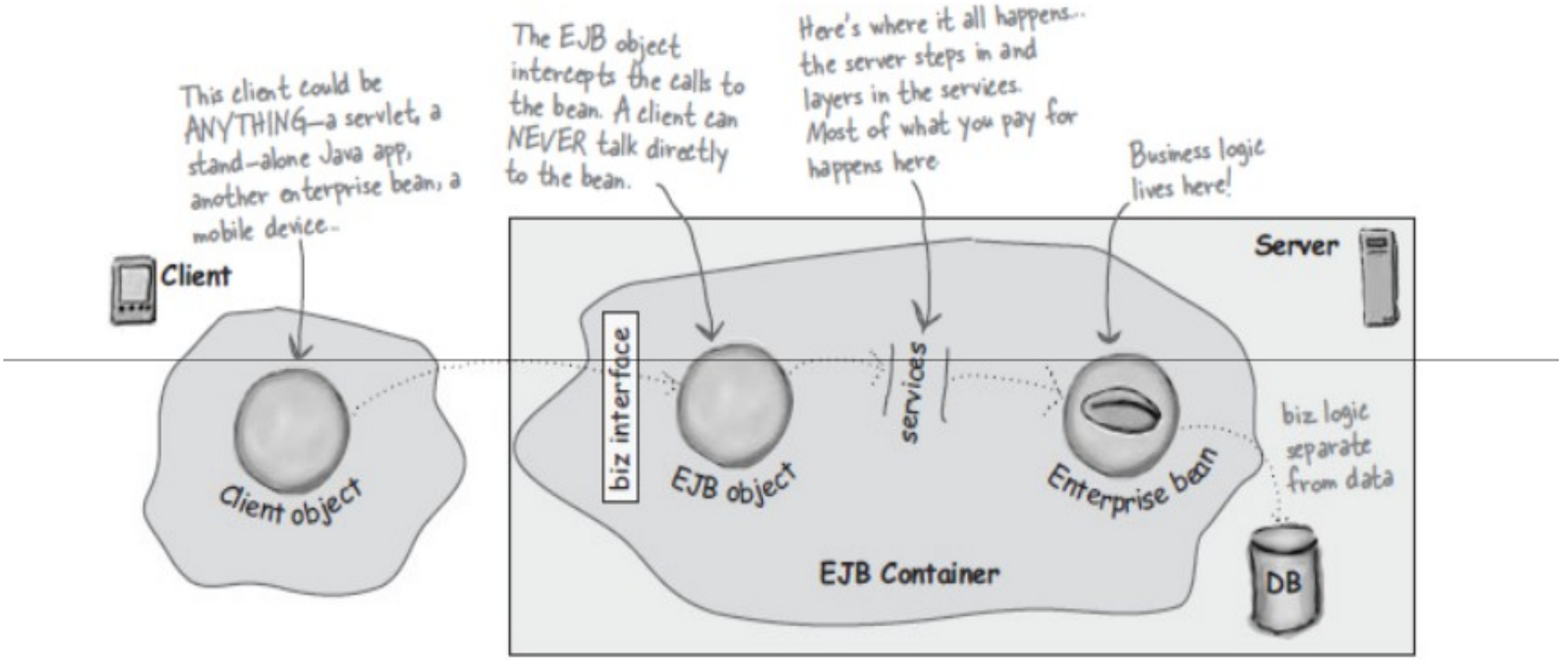
- Podem ser compostos por 1+N classes na versão 3.0
  - Uma ou mais interfaces de negócio que descrevem quais operações são disponíveis
    - Cada interface pode ser @Local ou @Remote, mas não ambas
  - A classe concreta que implementa os métodos
    - Pode injetar outros @EJB via annotation
- Podem ser implementados também sem nenhuma implementação de interfaces a partir da versão 3.1

EJB3: Session Beans – Serviços do Container

- Controle de concorrência e sincronização de threads
  - Não é necessário se preocupar com a sincronização no acesso por múltiplas threads pois uma instância nunca é compartilhada por mais de um cliente ao mesmo tempo
- Acesso remoto e web services
  - SLSBs podem ser acessados remotamente de maneira transparente para a implementação e para os clientes
- Controle de transações
  - Transações são criadas e terminadas automaticamente mesmo quando existirem múltiplos datasources e EJBs em servidores remotos acessados via RMI

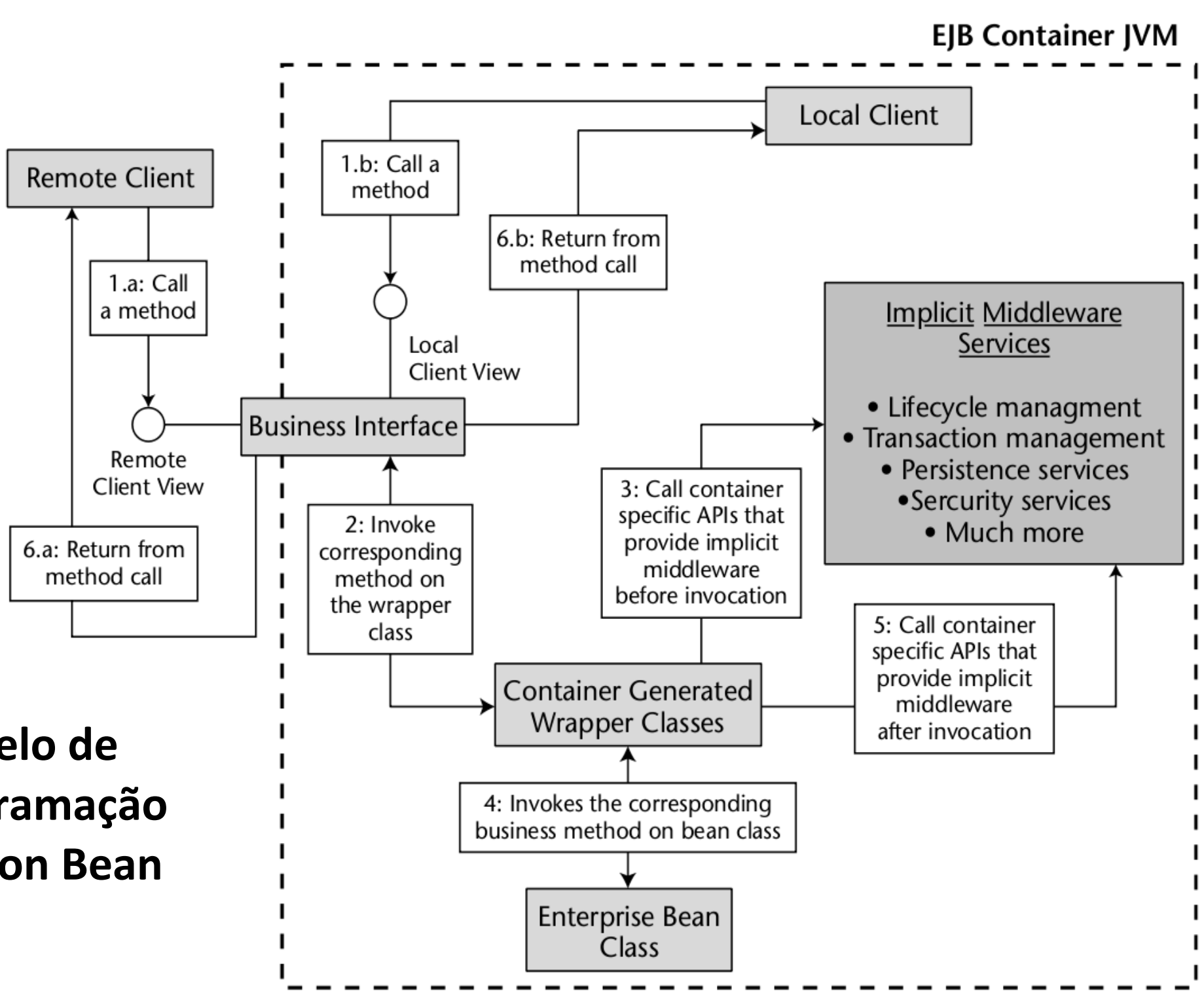
EJB3: Session Beans – Serviços do Container

- Serviço de timer
  - Suporta de maneira simples o disparo temporizado de um método de um SLSB, de maneira recorrente ou não
  - De qualquer forma é possível usar o Quartz ou algo equivalente de acordo com o servidor JEE
- Interceptadores
  - Versão simplicada de AOP que permite construir classes que são chamadas quando um SB é criado/destruído, ou quando uma operação é invocada por algum cliente
    - Útil para requisitos não-funcionais



Modelo de programação Session Bean

Modelo de programação Session Bean



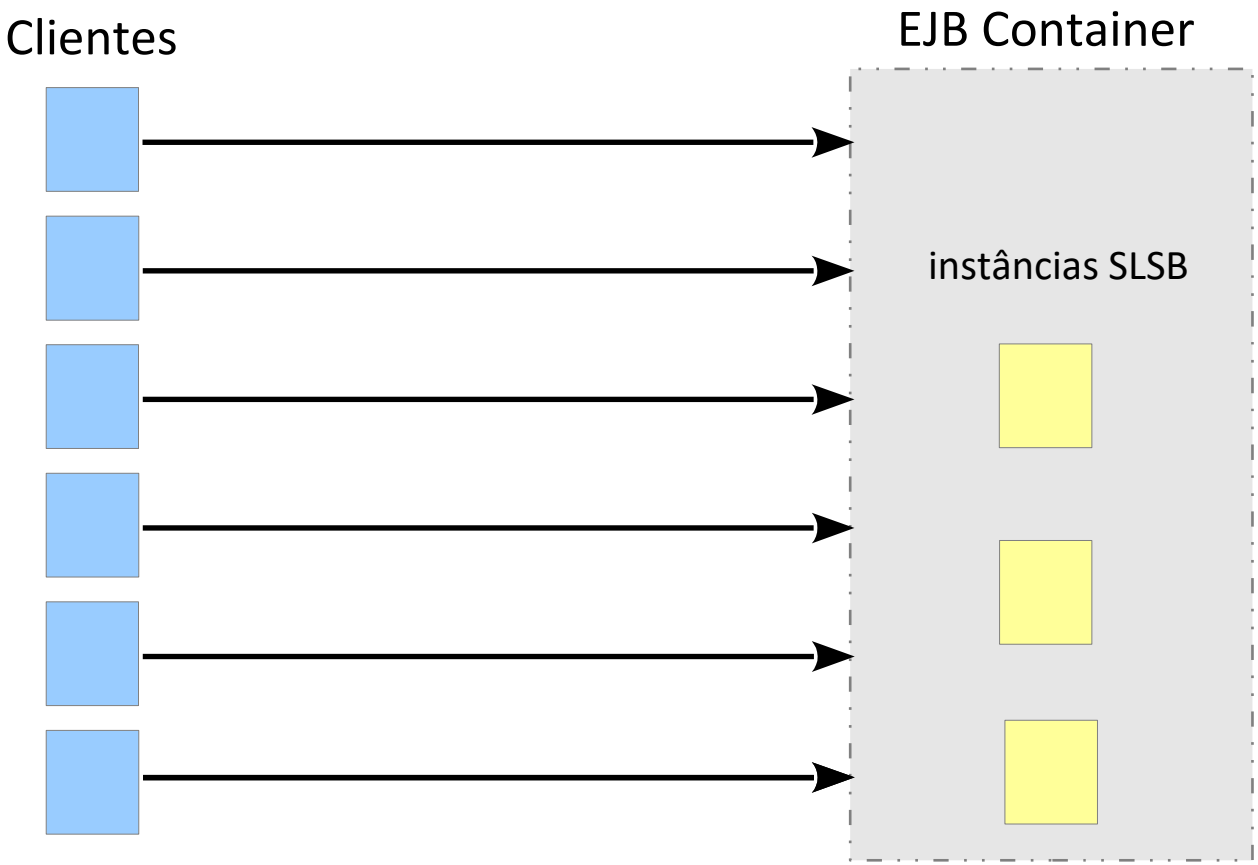
Stateless Session Beans

EJB3: Stateless Session Beans (SLSBs)

- Não mantém estado conversacional
- O container mantém um pool de SLSBs prontos pra uso
  - Duas chamadas sucessivas pelo mesmo cliente não devem assumir que a mesma instância SLSB será usada
- Quando um cliente realiza uma chamada o SLSB é alocado, invocado, e depois retornado para o pool
  - Em geral, poucas instâncias podem atender muitos clientes, tornando SLSBs altamente escaláveis
- Difere do modelo usado pelo Spring Framework, onde em geral usa-se o *pattern singleton*
  - Na JEE6 entretanto introduz-se o conceito de @Singleton para EJBs

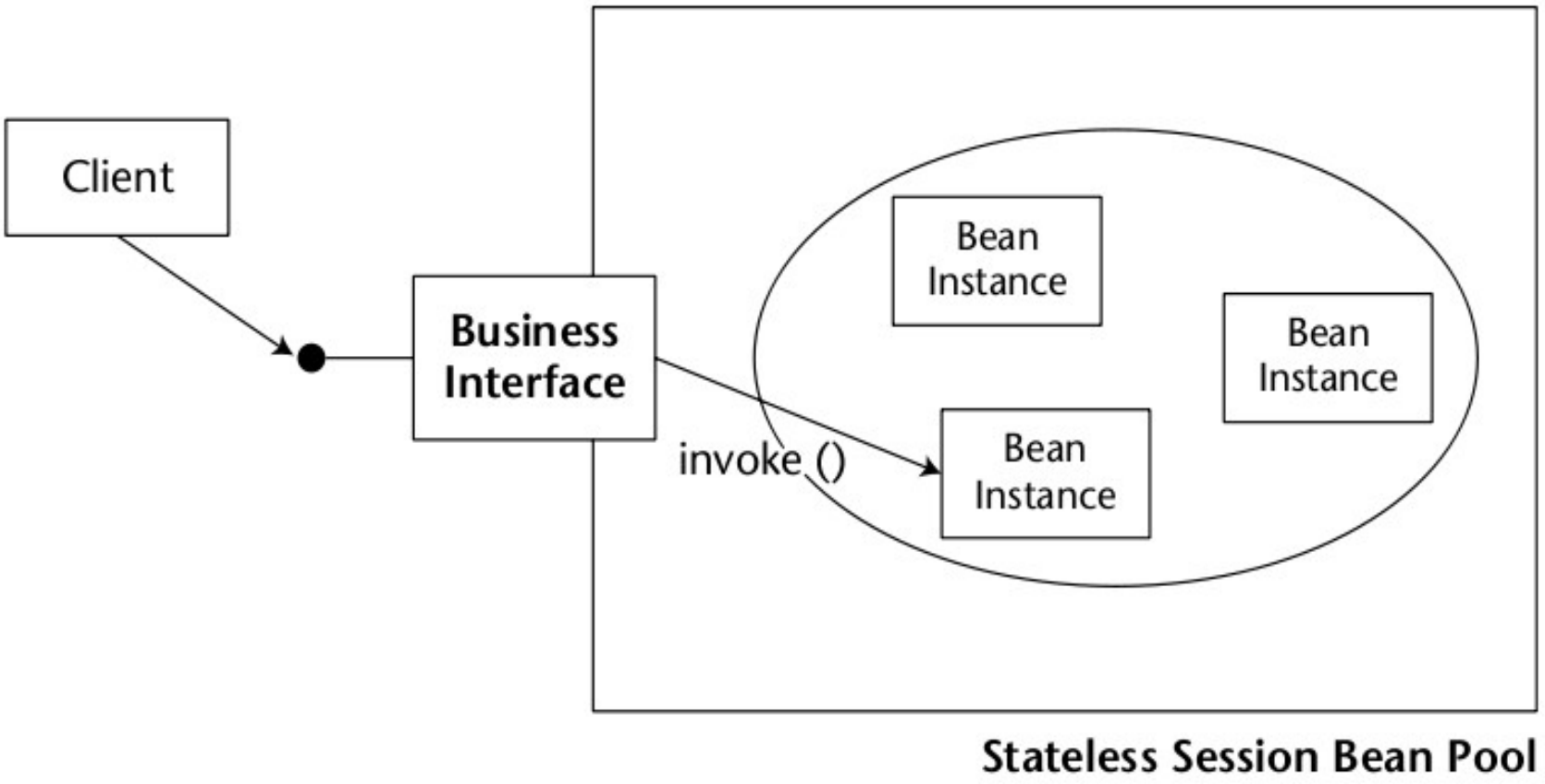
EJB3: Stateless Session Beans (SLSBs)

SLSBs tendem a ser mais escaláveis e de simples implementação



EJB3: Stateless Session Beans (SLSBs)

Pooling de SLSBs



EJB3: Stateless Session Beans - Regras

- Regras de programação de um SLSB
  - Classe deve ser concreta e ter um construtor default
  - Não deve ter atributos → caracterizam estado
  - Não devem injetar beans @Stateful → caracterizam estado
  - Pelo menos uma interface de negócio implementada
    - Elas devem ter @Local ou @Remote (solução mais simples)
  - Ter a annotation @Stateless
  - métodos não podem começar com “ejb”
  - Um EJB jamais deve iniciar threads!
- Pode estender outra classe
  - @Stateless jamais é herdada! → sempre declare
  - @PostConstruct e @PreDestroy são herdadas

EJB3: Stateless Session Beans - Interfaces

- Interfaces de negócio locais
  - Indicadas por @Local
  - Usadas por clientes localizados na mesma JVM do container
  - Possui semântica de invocação Java, onde referências para os objetos java são repassadas
  - Possuem excelente performance e devem ser usadas sempre que possível
  - A anotação @Local é opcional em uma interface a partir da versão 3.2. Neste caso, a interface assume que a mesma é Local.

EJB3: Stateless Session Beans - Interfaces

- Interfaces de negócio remotas via RMI
  - Indicadas por @Remote
  - todos os parâmetros e tipos de retorno devem ser serializáveis
  - Possuem excelente flexibilidade de deploy mas têm custo muito superior em comparação a chamadas locais
  - Apenas clientes Java são suportados
  - Propagam transações e contexto de segurança para beans remotos
  - Não propagam referências!!!

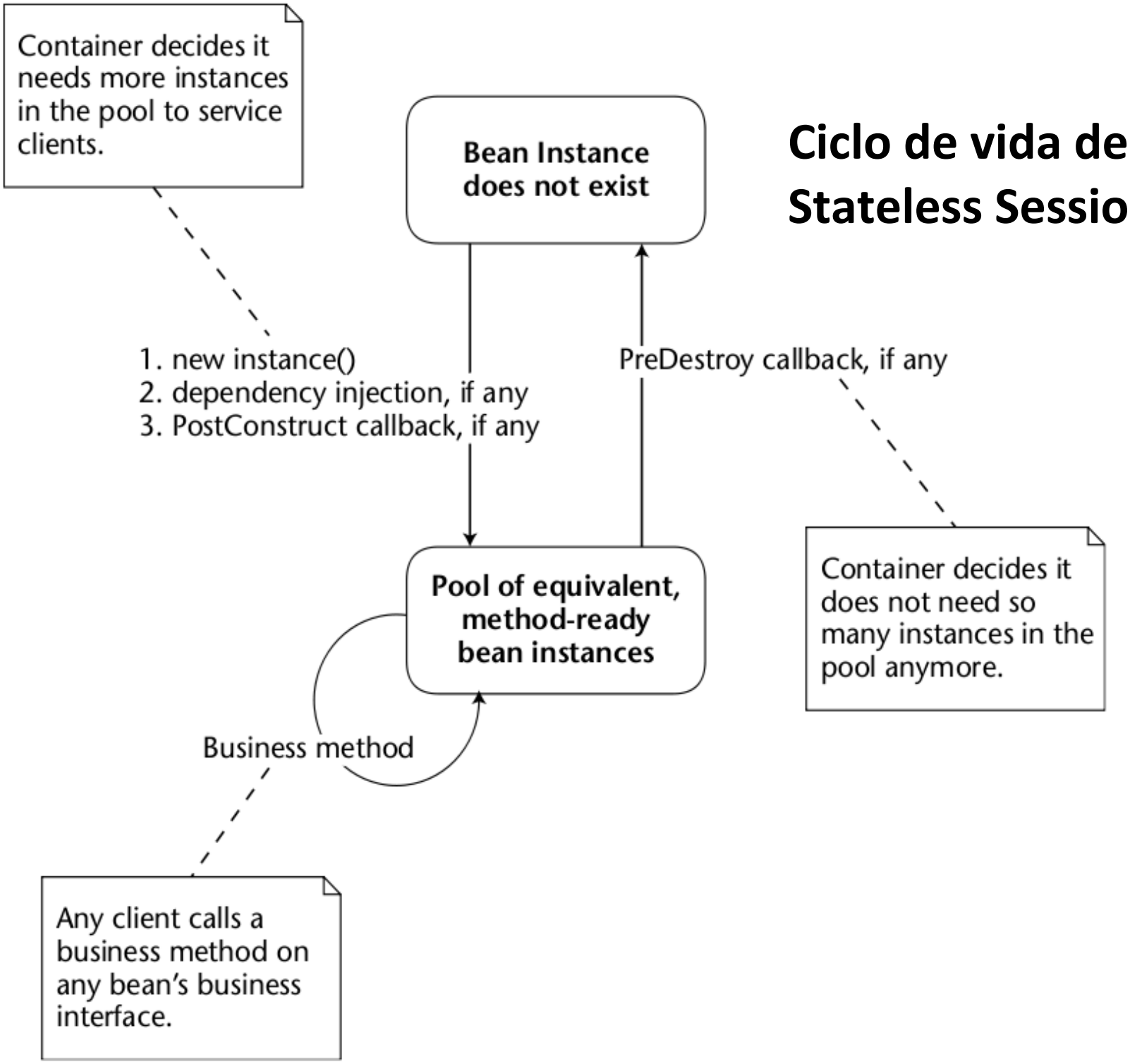
EJB3: Stateless Session Beans - Interfaces

- Interfaces de negócio remotas via WebService
  - Indicadas por @WebService
  - Usa SOAP sobre HTTP como protocolo de transporte
  - Pode ser chamado de qualquer tipo de cliente em qualquer arquitetura
  - Possui a menor performance de todas, portanto deve ser usado apenas em casos de integração onde se disponibiliza uma API para sistemas de terceiros
  - Não propaga transações

EJB3: Stateless Session Beans - Anotações

- @PostConstruct
  - Marca o método do bean a ser chamado logo depois de que a instância for criada e suas dependências injetadas
- @PreDestroy
  - Marca o método do bean a ser chamado imediatamente antes do container destruir definitivamente a instância
- Métodos podem ser private, protected, default, public

Ciclo de vida de Stateless Session Beans



EJB3: SLSBs - Exemplo (lifecycle callbacks)

```
@Stateless
public class ContaCorrenteService {

    ...

    @PostCreate
    void criarCache() {...}

    @PreDestroy
    void limparCache() {...}

}
```

EJB3: SLSBs - Exemplo (criação típica)

```
@Local
public interface CalculatorLocal {
    int sum(int a, int b);
    int mul(int a, int b);
}
```

```
@Remote
public interface CalculatorRemote {
     int sum(int a, int b);
}
```

```
@Stateless
public class CalculatorBean implements CalculatorLocal, CalculatorRemote {
    public int sum(int a, int b) {
        return a+b;
    }
    public int mul(int a, int b) {
        return a*b;
    }
}
```

EJB3: SLSBs - Exemplo (criação de outra forma)

```
public interface CalculatorLocal {
    int sum(int a, int b);
    int mul(int a, int b);
}
```

```
public interface CalculatorRemote {
     int sum(int a, int b);
}
```

```
@Stateless
@Local(CalculatorLocal.class)
@Remote(CalculatorRemote.class)
public class CalculatorBean implements CalculatorLocal, CalculatorRemote {
    public int sum(int a, int b) {
        return a+b;
    }
    public int mul(int a, int b) {
        return a*b;
    }
}
```

EJB3: SLSBs - Exemplo (invocando)

Utilizando JNDI

```
public class CalculatorServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res) {
        ...
        InitialContext ic = new InitialContext();
        CalculatorLocal calculator = (CalculatorLocal)
            ic.lookup("demoapp/CalculatorBean/local");
        int result = calculator.sum(v1, v2);
        ...
    }
}
```

Utilizando Annotations

```
public class CalculatorServlet extends HttpServlet {
    @EJB private CalculatorLocal calculator;
    public void doGet(HttpServletRequest req, HttpServletResponse res) {
        ...
        int result = calculator.sum(v1, v2);
        ...
    }
}
```

Exercício

- Montar a infra-estrutura inicial de um projeto JEE7 utilizando como ferramentas o Maven 3+ e Eclipse
- Construa um Stateless SessionBean que seja capaz de receber um arquivo XML (enviando como String) e salvá-lo no disco
- Crie testes automatizados deste serviço dentro do container

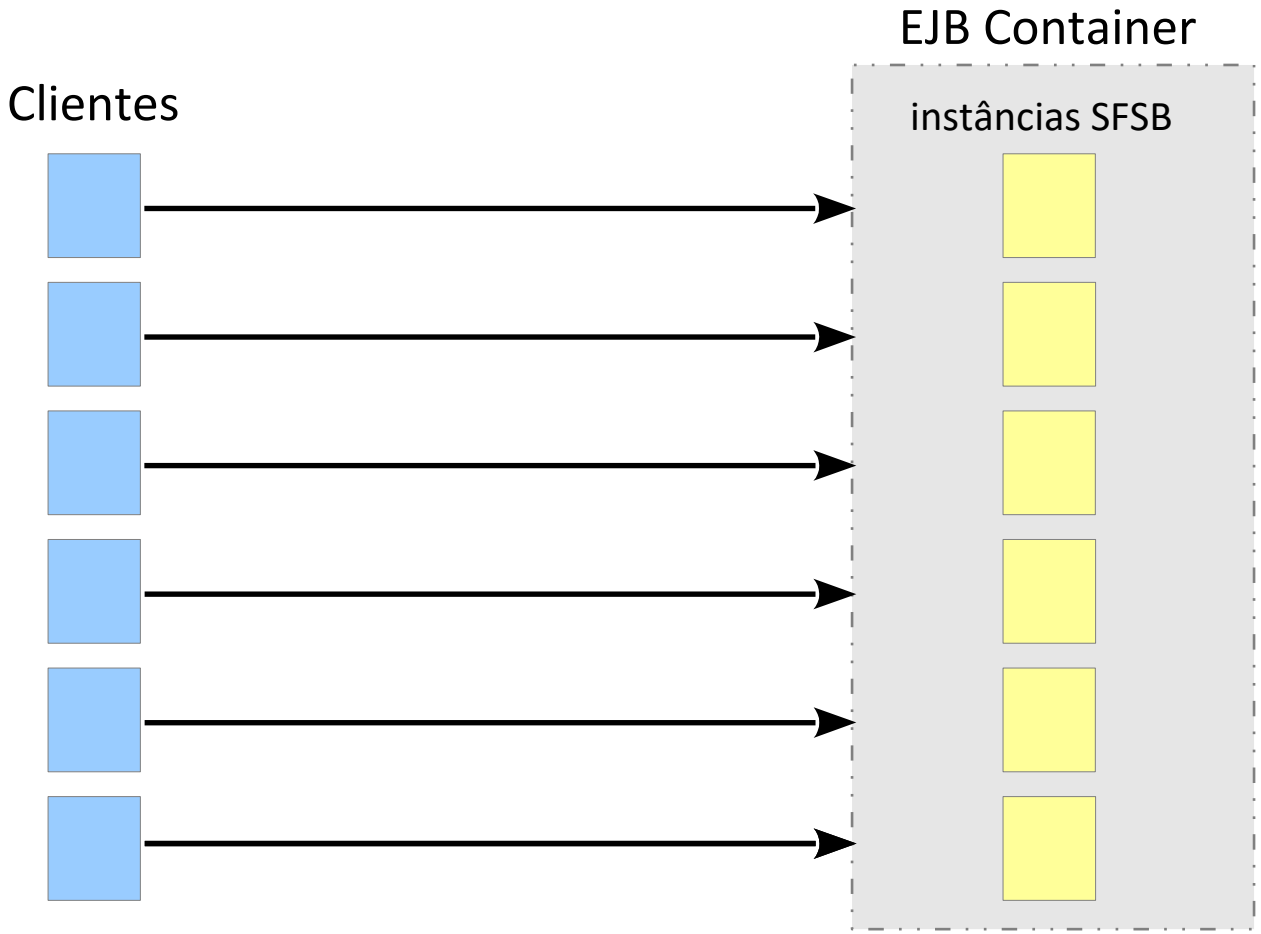
Stateful Session Beans

EJB3: Stateful Session Beans (SFSBs)

- Mantém estado conversacional
  - Chamadas sucessivas de um cliente para um mesmo SFSB acessam a mesma instância Java, portanto é possível armazenar temporariamente valores em atributos do EJB
  - Modelam muito bem cenários do tipo “carrinho de compras”
- A idéia de manter estado conversacional é equivalente a usar a sessão HTTP para armazenar valores temporariamente ao longo de várias requisições HTTP
- Não são tão escaláveis quanto SLSBs pois é necessário uma instância do SFSB por cliente

EJB3: Stateful Session Beans (SFSBs)

SFSBs tendem a usar mais recursos mas possibilitam comportamentos mais sofisticados



EJB3: Stateful Session Beans (SFSBs)

- Regras adicionais de programação
  - Atributos usados para armazenar informações da conversação devem ser serializáveis
  - É altamente recomendado que o cliente remova explicitamente o SFSB quando o estado da conversação não for mais necessário
    - Invocar algum método marcado com @Remove
  - Existem mais duas annotations de ciclo de vida
    - @PrePassivate
    - @PostActivate

EJB3: Stateful Session Beans (SFSBs)

- @Remove
  - Marca o método que se chamado pelo cliente causa a destruição da instância do SFSB
  - O método deve estar definido na interface de negócio pois o mesmo deve ser chamado pelo cliente, que não tem conhecimento da implementação

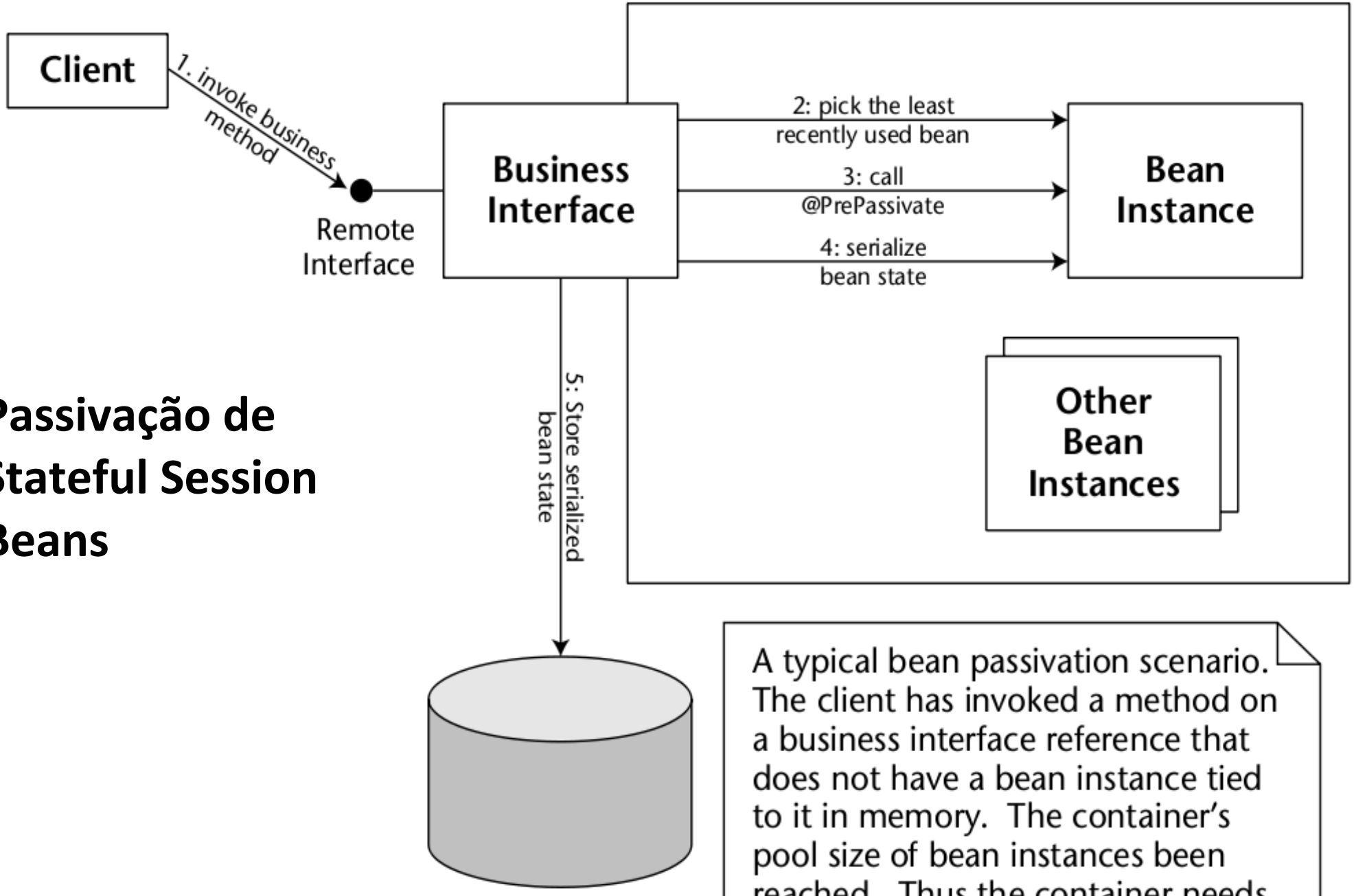
EJB3: Stateful Session Beans (SFSBs) - Passivação

- Passivar significa gravar o estado em disco
  - Pode acontecer quando atinge-se o número máximo de instâncias permitido pelo container
- O que é gravado em disco?
  - atributos não-transientes (primitivos ou objetos)
  - atributos injetados pela container como
    - referências para outros EJBs
    - referências para SessionContext
    - referências para contextos JNDI
- SFSBs devem ser removidos explicitamente pelo código cliente para evitar desperdício de recursos

EJB3: Stateful Session Beans (SFSBs)

- @PrePassivate
  - Marca o método que deve ser chamado pelo container quando o mesmo decidir que a instância do SFSB deve ser passivada
    - O estado é serializado e gravado em disco
  - Após a passivação o SFSB pode ser ativado por uma invocação em algum de seus métodos via interface de negócio
  - Geralmente usada em conjunto com @PreDestroy para liberar recursos alocados fora do container como conexões com banco de dados, arquivos, etc.

Passivação de Stateful Session Beans

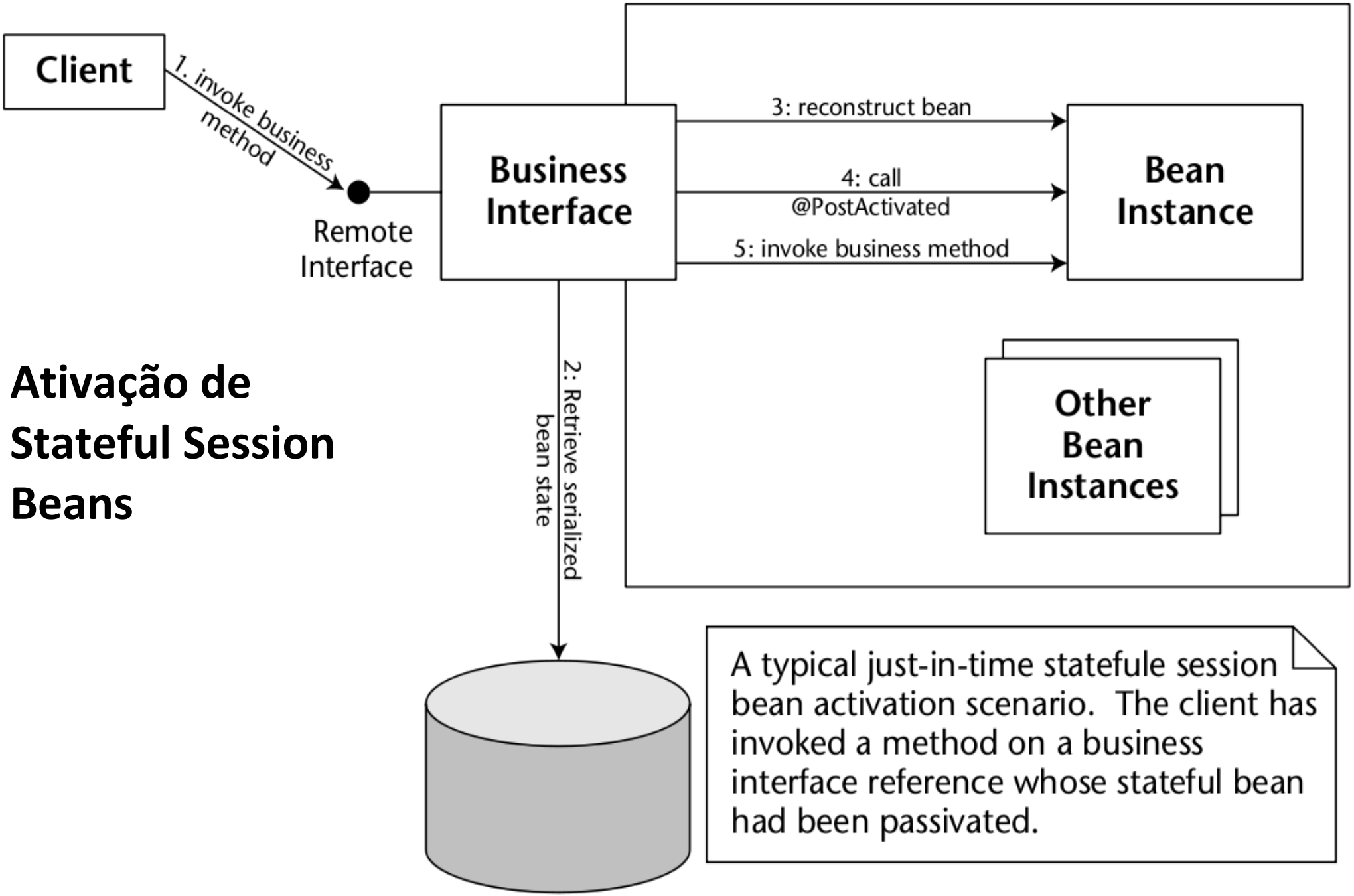


A typical bean passivation scenario. The client has invoked a method on a business interface reference that does not have a bean instance tied to it in memory. The container's pool size of bean instances been reached. Thus the container needs to passivate a bean before handling this client's request.

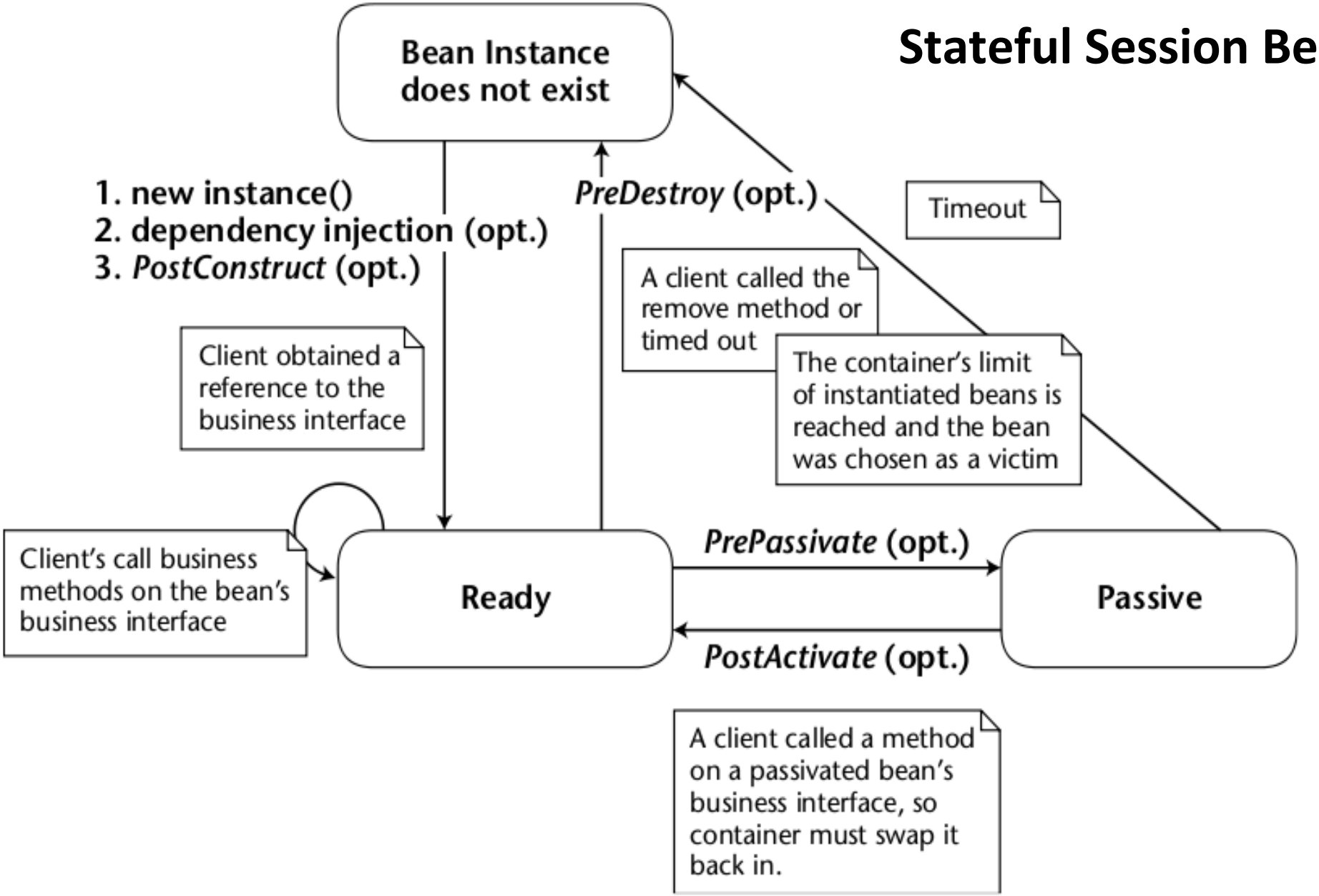
EJB3: Stateful Session Beans (SFSBs)

- @PostActivate
  - Marca o método que deve ser chamado pelo container logo após um SFSB ser deserializado do disco
  - Dependendo de container o estado em disco pode ser removido após um timeout. Neste caso, a tentativa de invocar qualquer método resultará em uma exception
  - Geralmente usado em conjunto com @PostCreate para alocar recursos necessários pelos métodos de negócio

Ativação de Stateful Session Beans



Ciclo de vida de Stateful Session Beans

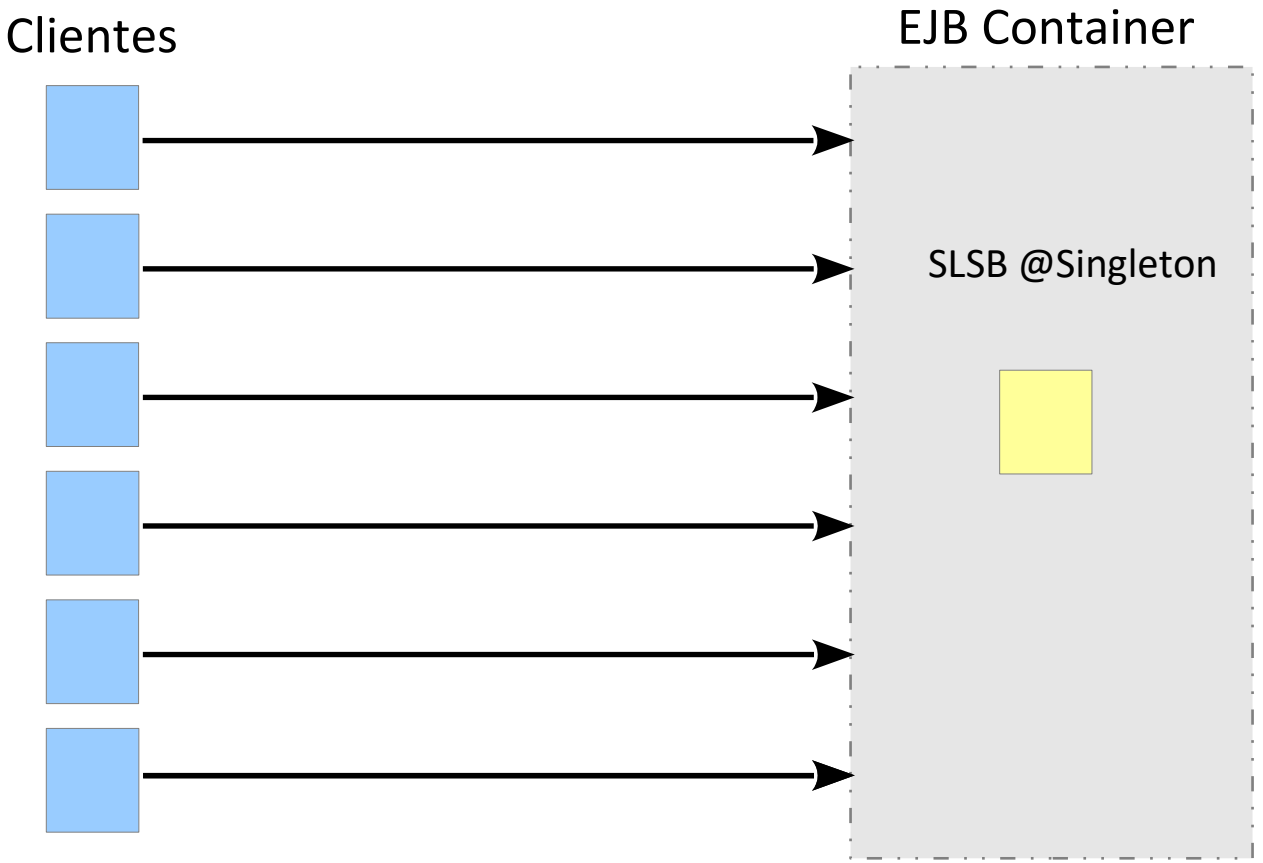


Exercício

- Crie um carrinho de compra capaz de adicionar, remover e alterar a quantidade de itens de um produto, totalizando o valor da compra
- Crie testes automatizados para este carrinho

EJB3: Singletons

- É criada apenas uma instância do Session Bean, assim todos os clientes acessam a mesma instância
- Pode manter estado



EJB3: Singletons

- A criação é muito simples: utilize a anotação @Singleton
- Pode ser utilizado para inicializar recursos no sistema, em conjunto com a utilização da annotation @Startup
- Pode ser utilizado também para realizar agendamentos de tarefas

Controle de concorrência em componentes Singleton

- Bean Managed Concurrency
  - Desenvolvedor sincroniza a chamada dos métodos
- Container Managed Concurrency
  - O container sincroniza a chamada dos métodos
- @Lock(LockType.READ)
  - Habilita o container a permitir que múltiplos clientes chamem o mesmo método ao mesmo tempo (alta escalabilidade)
- @Lock(LockType.WRITE)
  - Garante o acesso exclusivo ao Bean, permitindo somente 1 cliente por vez para executar o método

Singleton

```
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
@Singleton
public class ExampleSingletonBean {
  private String state;

  @Lock(LockType.READ)
  public String getState() {
    return state;
  }

  @Lock(LockType.WRITE)
  public void setState(String newState) {
    state = newState;
  }
}
```

Exercício

- Crie um componente EJB Singleton que seja capaz de armazenar em memória todos os estados do Brasil, que devem ser carregados de um arquivo texto comum;
- Fornecer métodos para recuperação de estados por Sigla e também todos os estados
- Criar um componente EJB Singleton que conte o número de requisições que os serviços disponibilizados pela aplicação estão tendo.
- Crie testes automatizados para todos os casos

Timers

- Os temporizadores permite a execução de serviços de forma agendada.
  - Exempo: Rotina de faturamento mensal executa apenas no último dia de todo mês
  - Exemplo: Rotina de expurgo de dados antigos executa todo dia a meia noite
- Pode ser usado com Stateless Session Beans e Singleton
  - Se usado com SLSB uma instancia sera escolhida para execução
  - Se usado com Singleton a única instancia do bean sera utilizada. Neste caso é possível recuperar o estado do bean na execução anterior, caso necessário

Timers

- Exemplo de um simples temporizador

```
@Singleton
public class TimerService {

    @EJB
    HelloService helloService;

    @Schedule(second="*/1", minute="*",hour="*", persistent=false)
    public void doWork(){
        System.out.println("timer: " + helloService.sayHello());
    }
}
```

Timer

- Os temporizadores podem ser utilizados com anotações ou programaticamente
- Exemplos de timers com anotações

```
@Schedule(second="0", minute="1", hour="23", dayOfMonth="1", month="Apr",
dayOfWeek="Mon", year="2019")
```

→ Executa as 23:01:00 de 01/04/2019

```
@Schedule(second="*", minute="*", hour="*", dayOfMonth="*",
month="*",dayOfWeek="*", year="*")
```

→ Executa a cada segundo de cada minuto de cada hora de cada dia de cada mês de cada ano

```
@Schedule(second="0,29", minute="0,14,29,59",
hour="0,5,11,17",dayOfMonth="1,15-31",year="2018,2019")
```

→ Executa no segundo 0 e 29 dos minutos 0, 14, 29, 59, nas horas 0, 5, 11 e 17 nos dias 1 e de 15-31 para os anos de 2018 e 2019

```
@Schedule(hour="0-11",dayOfMonth="15-31",dayOfWeek="Mon-Fri",month="11-12",
year="2010-2020")
```

→ Executa das 0 as 11 horas entre os dias 15 e 31 de Segunda a Sexta-feira nos meses de novembro e dezembro entre 2010 e 2020

Timer

- Exemplo de timer utilizando programação

```
@Singleton
@Startup
public class LogRemovalTimer {

	private static final SimpleDateFormat SD = new SimpleDateFormat("dd/MM/yyyy hh:mm:ss");

	@Resource
	private TimerService timerService;

	@PostConstruct
	public void init() {
		ScheduleExpression se = new ScheduleExpression();
		se.month("*").dayOfMonth("*").hour("*").hour("*").minute("*/10").second("*");
		TimerConfig tc = new TimerConfig("Ola Pos Graduacao", true);
		timerService.createCalendarTimer(se, tc);
	}

	@Timeout
	public void executar(Timer timer) {
		System.out.println(String.format("Executando timer em %s com o valor %s",
				SD.format(Calendar.getInstance().getTime()), timer.getInfo()));
	}

}
```

EJB Assíncronos

- Devem ser utilizados em 2 circunstâncias:
  - Transações longas que você quer começar e fazer outra coisa enquanto ela processa, independente do que aconteça com esta transação
  - Transações longas que você quer começar e monitorar, inclusive cancelar
  - A execução não é garantida em caso de falha do container.
  - A chamada de EJB´s assíncronos a partir de um EJB síncrono, não é executado na mesma transação

EJB Assíncrono (Exemplo)

```
@Stateless(name = "emailService")
public class EmailService {

    Logger logger = Logger.getLogger(AuthenticateBean.class.getName());

    @Resource(name="config/emailSender")
    private String emailSender;

    @Resource(name = "mail/notification")
    private Session session;

    @Asynchronous
    public void sendEmail(String emailAddress, String subject,
      String htmlMessage) {
        try {
            MimeMessage message = new MimeMessage(session);
            message.setFrom(new InternetAddress(emailSender));
            InternetAddress[] toAddress = new InternetAddress[] {
                new InternetAddress(emailAddress)};
            message.setRecipients(Message.RecipientType.TO, toAddress);
            message.setSubject(subject);
            message.setContent(createHtmlMessage(htmlMessage));
            Transport.send(message);
        } catch (Throwable t) {
```

Asynchronous annotation signals container to run method in another thread

No return parameters—fire and forget

EJB Assíncro (Exemplo)

```
@Stateless(name="billingService")
@Asynchronous
public class BillingService {

    @Resource
    private SessionContext sessionContext;

    public Future<Boolean> debitCreditCard(Order order) {
        boolean processed = false;
        if(sessionContext.wasCancelCalled()) {
            return null;
        }
        // Debit the credit card
        return new AsyncResult<Boolean>(processed);
    }
}
```

Marks class as asynchronous

Injects SessionContext for honoring cancel

Returns a value wrapped in Future<Boolean>

Checks to see if cancelled

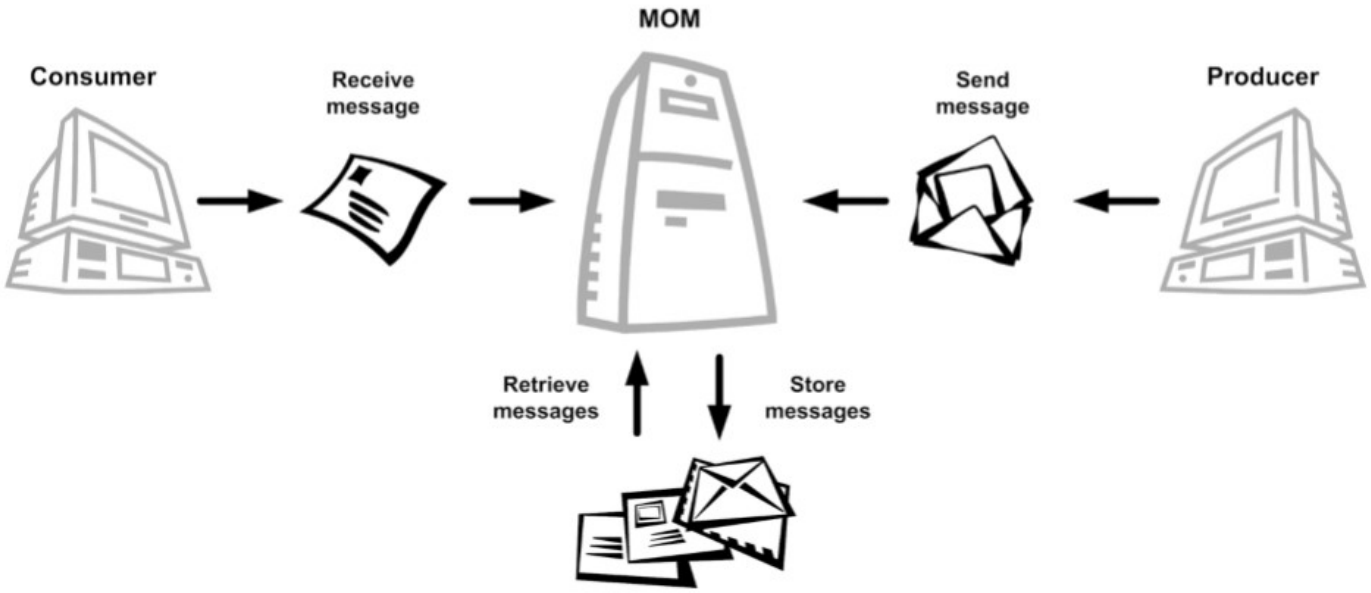
Wraps result in convenience implementation of Future

Exercício

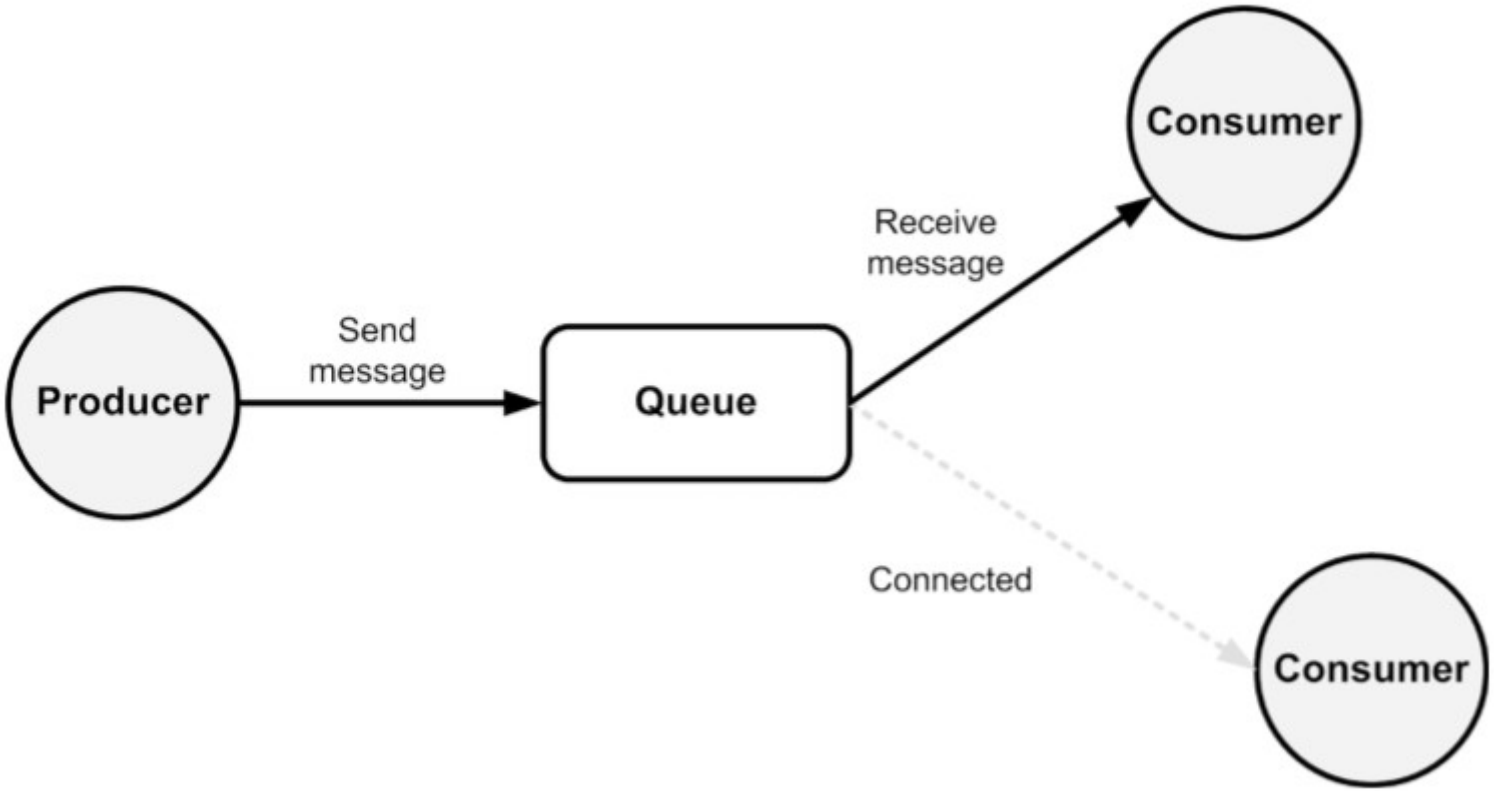
- Crie um EJB assíncrono que seja responsável por calcular a soma e a média de uma lista de inteiros.
- Crie um EJB síncrono, que faça chamada ao EJB assíncrono anterior respeitando a seguinte regra. Se a Lista de inteiros for maior que 10 elementos, separe em duas listas e processe independentemente cada lista. Ao finalizar as duas listas, chame novamente o EJB para calcular o resultado destas duas listas.
- Crie testes automatizados para exercitar estes casos.

Message Driven Beans

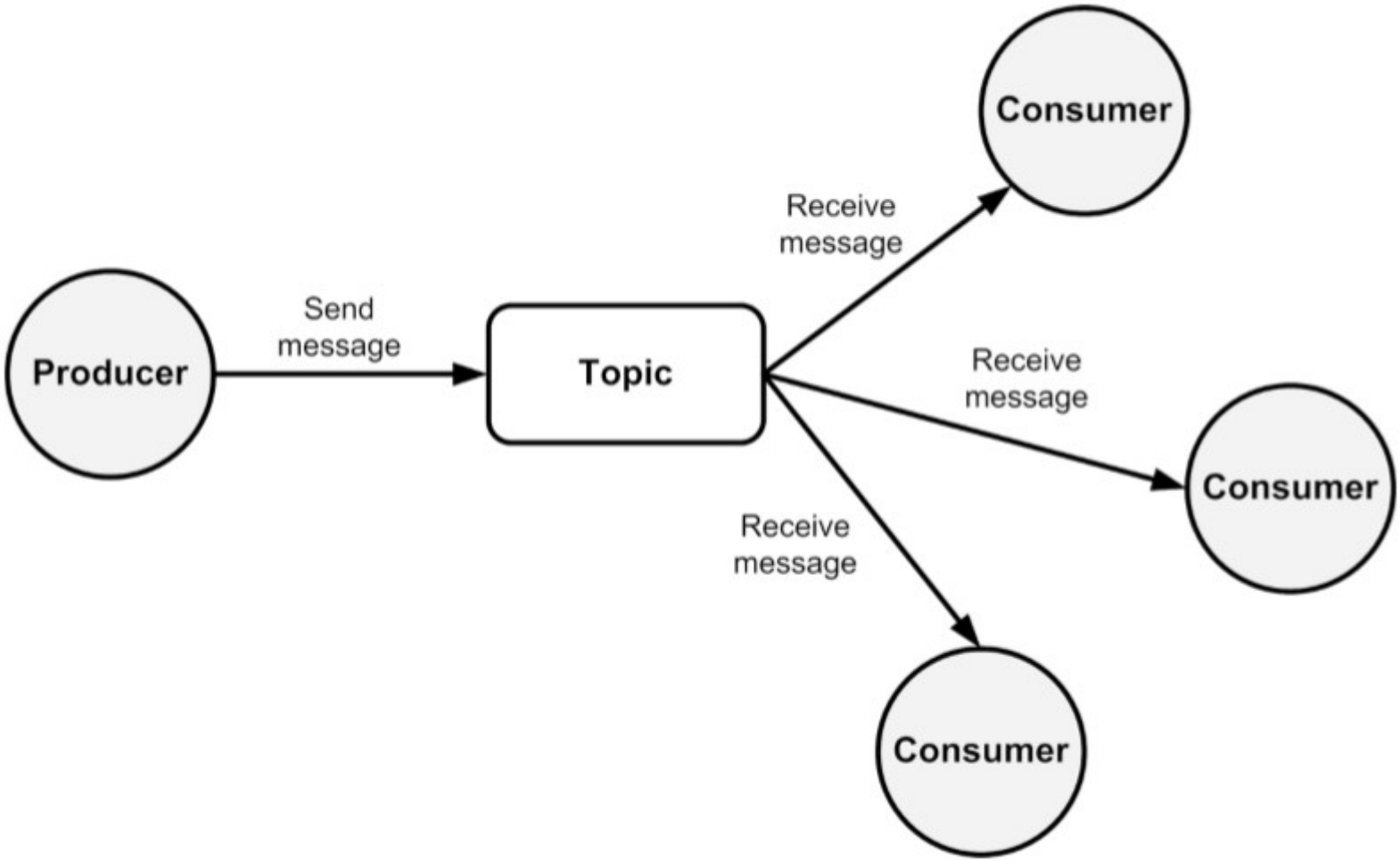
- Processo de envio de mensagens assíncronas, com baixo acoplamento, que são transferidas em um processo seguro
- Message-oriented-middleware (MOM) é o software responsável por receber e distribuir as mensagens. Pode ser também chamado de Message System ou Message Broker
- Exemplos de MOM: ActiveMQ, RabitMQ



Message Driven Beans – Point to Point



Message Driven Beans – Publish / Subscribe



Java Message Service

- JMS é uma API Java, parte da especificação JEE que define o protocolo de comunicação de aplicações Java com os MOM´s

Enviando uma mensagem

```
@Inject
@JMSConnectionFactory("jms/QueueConnectionFactory")
private JMSContext context;

@Resource(name="jms/ShippingRequestQueue")
private Destination destination;

ShippingRequest shippingRequest = new ShippingRequest();
shippingRequest.setItem("item");
shippingRequest.setShippingAddress("address");
shippingRequest.setShippingMethod("method");
shippingRequest.setInsuranceAmount(100.50);

ObjectMessage om = context.createObjectMessage();
om.setObject(shippingRequest);

JMSProducer producer = context.createProducer();
producer.send(destination, om);
```

1. Basic CDI @Inject annotation
3. Simplified interface for JMS operations
4. The queue to use
5. ShippingRequest to send to queue
6. JMSObjectMessage to send ShippingRequest
7. JMSProducer to send message

Consumindo uma mensagem com um MDB

```
@MessageDriven(activationConfig = {
  @ActivationConfigProperty(propertyName = "destinationType",
    propertyValue = "javax.jms.Queue"),
  @ActivationConfigProperty(propertyName = "destinationLookup",
    propertyValue = "jms/ShippingRequestQueue")
})
public class TurtleShippingRequestMessageBean implements MessageListener{

  @PersistenceContext()
  EntityManager entityManager;

  @Override
  public void onMessage(Message message) {
    try {
      ObjectMessage om = (ObjectMessage) message;
      Object o = om.getObject();
      ActionBazaarShippingRequest sr = (ActionBazaarShippingRequest) o;
      Logger.getLogger(TurtleShippingRequestMessageBean.class.getName())
        .log(Level.INFO, String.format("Got message: %s", sr));

      TurtleShippingRequest tr = new TurtleShippingRequest();
      tr.setInsuranceAmount(sr.getInsuranceAmount());
      tr.setItem(sr.getItem());
      tr.setShippingAddress(sr.getShippingAddress());
      tr.setShippingMethod(sr.getShippingMethod());
      entityManager.persist(tr);
    } catch (JMSException ex) {
      Logger.getLogger(TurtleShippingRequestMessageBean.class.getName())
        .log(Level.SEVERE, null, ex);
    }
  }
}
```

1. Configures bean with queue it listens on for messages
2. Implements MessageListener interface for processing JMS messages
3. Implements onMessage method
4. Retrieves ActionBazaar data
5. Converts ActionBazaar data to Turtle data
6. Persists shipping request to Turtle's database

Injects EntityManager for saving data to database

Transações com MDBs

- Por padrão, o container inicializará uma transação antes do método onMessage ser invocado e finalizará (commit) tão logo o método retorne
- Caso seja lançada uma exceção não tratada, será executado o rollback
- A transação também pode ser marcada como rollback através do contexto do Message Driven Bean (MessageDrivenContext)
- Pode-se “desligar” a transação se necessário
  - Ex: processamento de um arquivo não precisa de transação
- Lembre-se que a transação de um MDB não é a mesma transação de quem criou a mensagem

Exercicio

- Crie um aplicativo (cliente) que envie uma mensagem a uma fila de impressão de Notas Fiscais
- Crie um MDB que consumta as mensagens da fila de impressão de Nota Fiscal e mostre no console

Interceptors

- Interceptors implementam os conceitos de Programação Orientada a Aspectos
- É especialmente utilizar para implementar conceitos ortogonais do software como log, auditoria, segurança
- Um interceptor é disparado na entrada de um método
- Podem ser utilizados também para implementar regras de negócio ortogonais no sistema
- A execução dos interceptors é definida pela ordem que são definidas nas classes interceptadas
- É similar ao Filter Servlet
-

Interceptors

```
@Stateless
public class BidServiceBean implements BidService {
    @Interceptors(ActionBazaarLogger.class)
    public void addBid(Bid bid) {
    }
}

public class ActionBazaarLogger {
    @AroundInvoke
    public Object logMethodEntry(
    InvocationContext invocationContext)
        throws Exception {
    System.out.println("Entering method: "
        + invocationContext.getMethod().getName());
    return invocationContext.proceed();
    }
}
```

1 Attaching interceptor

2 Specifying interceptor method

Interceptors CDI

- Iguais aos EJB interceptors, mas são utilizados utilizando annotations

```
@InterceptorBinding
@Retention(RetentionPolicy.RUNTIME)
@Target({TYPE, METHOD, FIELD})
@Qualifier
public @interface Log { }
}

@Interceptor
@Log
@Priority(Interceptor.Priority.APPLICATION) // CARREGA O INTERCEPTADOR AUTOMATICAMENTE SEM A NECESSIDADE DE DEFINI-LO NO
BEANS.XML
public class LoggingInterceptor implements Serializable {

    private static final Logger logger =
            Logger.getLogger(LoggingInterceptor.class);
    @AroundInvoke
     public Object logMethodEntry(InvocationContext ctx) throws Exception {
           logger.info("Before entering method:" + ctx.getMethod().getName());
           return ctx.proceed();
     }
}

@Stateless
@Log
Public class Service {
       Public void execute() {
       …
       }
}
```

Exercício

- Crie um interceptor que imprima a data de início e fim da execução de cada chamada de método e vincule o mesmo as classes dos exercícios anteriores
- Crie um interceptor que, permita somente a execução de um método, se o mesmo estiver sendo executado entre 8h e 18h
- Crie testes automatizados para cada um dos interceptadores

Events e Observers

- Eventos permitem beans se comunicarem sem dependência em tempo de compilação
- Um bean pode definir um evento e outro bean pode disparar este evento e outro pode processar este evento.
- Na prática é a implementação do padrão de projetos Observer no em JEE
- Exemplo:
  - Ao finalizar o processamento da rotina de faturamento mensal, devem-se executar 3 ações: enviar e-mail ao administrador do sistema, enviar o relatório consolidado ao gestor da área, disparar a rotina de cálculo de impostos do faturamento

Events e Observers

```
public class LogRemovedEvent {
}

import javax.ejb.Stateless;
import javax.enterprise.event.Observes;

@Stateless
public class LogRemovedObserver {
	public void handleLogRemoved(@Observes LogRemovedEvent event) {
		System.out.println("Log foi removido com sucesso");
	}
}

@Singleton
@Startup
public class LogRemovalTimer {

	private static final SimpleDateFormat SD = new SimpleDateFormat("dd/MM/yyyy hh:mm:ss");

	@Resource
	private TimerService timerService;

	@Inject
	private Event<LogRemovedEvent> notifier;

	@PostConstruct
	public void init() {
		ScheduleExpression se = new ScheduleExpression();
		se.month("*").dayOfMonth("*").hour("*").hour("*").minute("*/10").second("*");
		TimerConfig tc = new TimerConfig("Ola Pos Graduacao", true);
		timerService.createCalendarTimer(se, tc);
	}

	@Timeout
	public void executar(Timer timer) {
		System.out.println(String.format("Executando timer em %s com o valor %s",
				SD.format(Calendar.getInstance().getTime()), timer.getInfo()));
		notifier.fire(new LogRemovedEvent());
	}

}
```

Exercício

- Crie um EJB Stateless que observe a ocorrência do evento PagamentoEfetuado e escreva no console o valor do pagamento efetuado
- Crie um EJB Stateless que receba como parametro um Funcionário, o mês e o valor e escreva no console o valor do pagamento do funcionario. Este ejb deve notificar a ocorrencia do evento PagamentoEfetuado

Transações

Transações

- ACID
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- Transações distribuídas (XA)
- Gerenciamento em EJBs
- Níveis de Isolamento

Transações: Tipos de Gerenciamento

- Em EJBs, o gerenciamento das transações pode ser feito de 2 formas
  - **Pelo container**: a demarcação é feita automaticamente ao se entrar ou sair de métodos
    - É o modo default caso nada seja especificado
    - Pode ser customizado pelos *atributos de transação*
    - É a forma recomendada
  - **Pelo bean**: a demarcação é feita programaticamente ao marcar o EJB com @TransactionManagementType
    - Via UserTransaction, injetado via @Resource, é possível fazer begin, commit, rollback
    - Evite se possível

Transações: Gerenciamento pelo Container

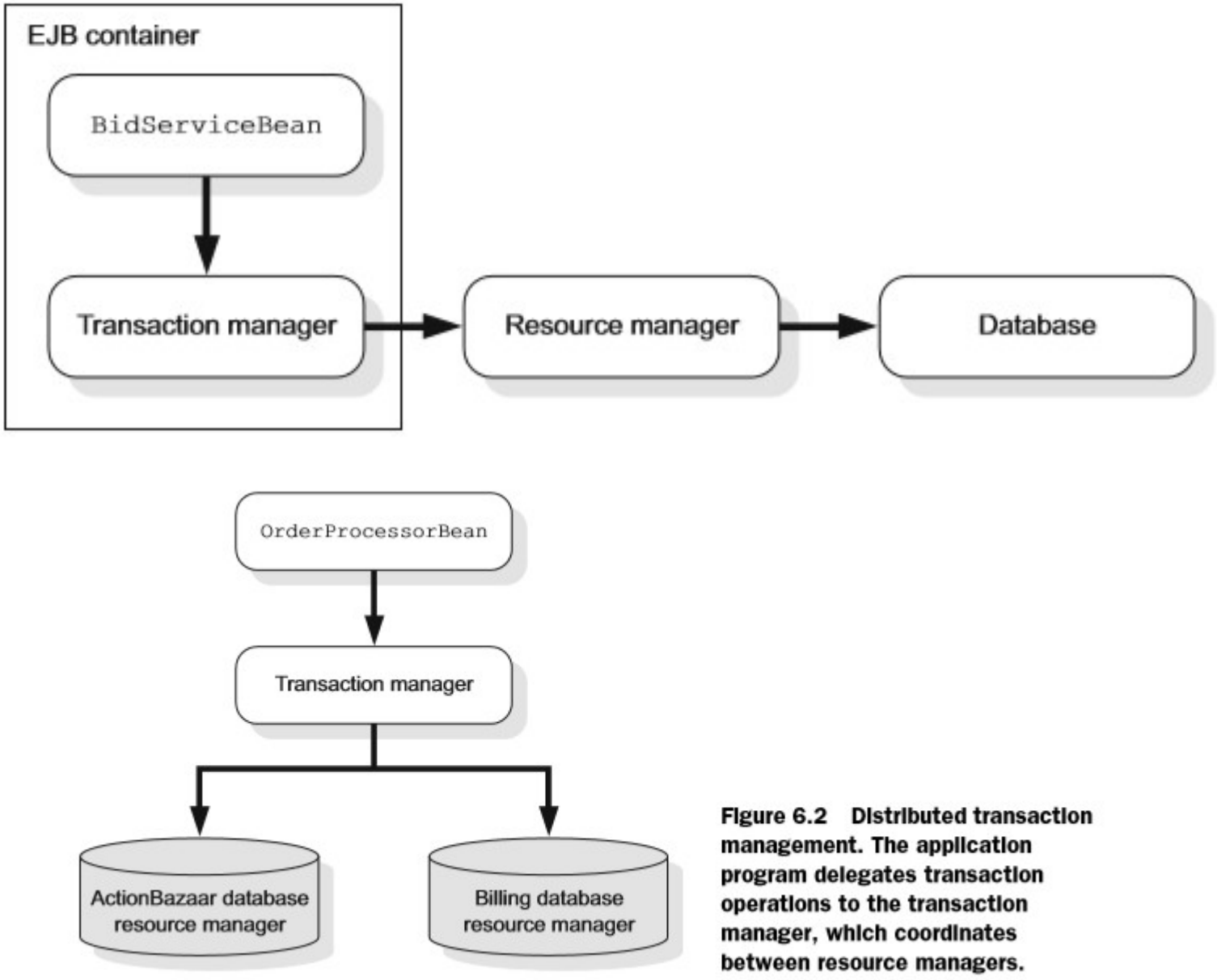


Figure 6.2 Distributed transaction management. The application program delegates transaction operations to the transaction manager, which coordinates between resource managers.

Transações: Tipos de Atributos

- O que são?
  - Indicam como o gerenciamento de transações gerenciado pelo container deve ser feito
- Podem ser especificados para a classe inteira ou para métodos específicos via @TransactionAttribute
    - @TransactionAttributeType.REQUIRED
    - @TransactionAttributeType.REQUIRES_NEW
    - @TransactionAttributeType.SUPPORTS
    - @TransactionAttributeType.MANDATORY
    - @TransactionAttributeType.NOT_SUPPORTED
    - @TransactionAttributeType.NEVER

Transações: Métodos transacional

- O que significa dizer que um método é transacional?
  - Significa que o mesmo atende a todos os critérios do acrônimo ACID. Em especial Atomicidade;
  - Equivalente a setar autoCommit = false em uma Conection;
- O que significa dizer que 2 métodos ocorrem na mesma transação?
  - Significa que compartilham a mesma Connection (XA ou não) e que seus comandos executam no mesmo bloco begin/commit/end;

Transações: Tipos de Atributos

- **Required**
  - Se existir uma transação, continue-a. Ao retornar do método não termine a transação. Deixe quem a criou fechá-la
  - Se não existir uma transação, crie um nova. Ela será terminada ao se retornar do método
  - É o modo default
- **RequiresNew**
  - Existindo ou não uma transação, crie uma nova. Ela será terminada ao se retornar do método
  - Útil para isolar o sistema de partes não confiáveis
  - Use com cuidado pois pode afetar a performance

Transações: Tipos de Atributos

- **Supports**
  - Se exisistir uma transação, deixe-a continuar
  - Se não existir, continue sem nenhuma transação
  - Não é recomendada pois pode trazer resultados não previstos de acordo com a existência/ausência de transação
- **Mandatory**
  - Se não exitir uma transação lance uma exception
  - Ou o método roda numa transação existente ou falha

Transações: Tipos de Atributos

- **NotSupported**
  - Se existir uma transação, suspenda-a temporariamente até o método retornar
  - O método nunca roda numa transação
- **Never**
  - Se existir uma transação, lance uma exception
  - O método nunca roda numa transação

Transações: Tipos de Atributos

TRANSACTION ATTRIBUTE	CLIENT'S TRANSACTION	BEAN'S TRANSACTION
`Required`	None	T2
	T1	T1
`RequiresNew`	None	T2
	T1	T2
`Supports`	None	None
	T1	T1
`Mandatory`	None	Error
	T1	T1
`NotSupported`	none	None
	T1	None
`Never`	none	None
	T1	Error

Transações: Níveis de Isolação

- O que é?
  - Indica como uma transação é afetada por outras transações durante a execução do sistema
    - Níveis de isolação:
      - READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE
    - Tipos de problemas:
      - Dirty Read, Unrepeteable Read, Phanton Read
  - De acordo com a aplicação, pode ser fundamental identificar o nível adequado
  - Cada nível de isolação possui um custo para o banco de dados

Transações: Níveis de Isolação - Problemas

- **Dirty Read Problem**
  - Quando uma transação T1 consegue ver modificações feitas por uma transação T2 antes de T2 fazer commit
  - Resolvido via READ_COMMITTED
  - Em geral é o default (melhor custo-benefício)
- **Unrepeteatable Read Problem**
  - Quando uma transação T1, ao ler novamente uma informação do BD, vê valores diferentes em relação à primeira leitura
  - Geralmente resolvido com locks ou pelo uso de REPEATABLE READ

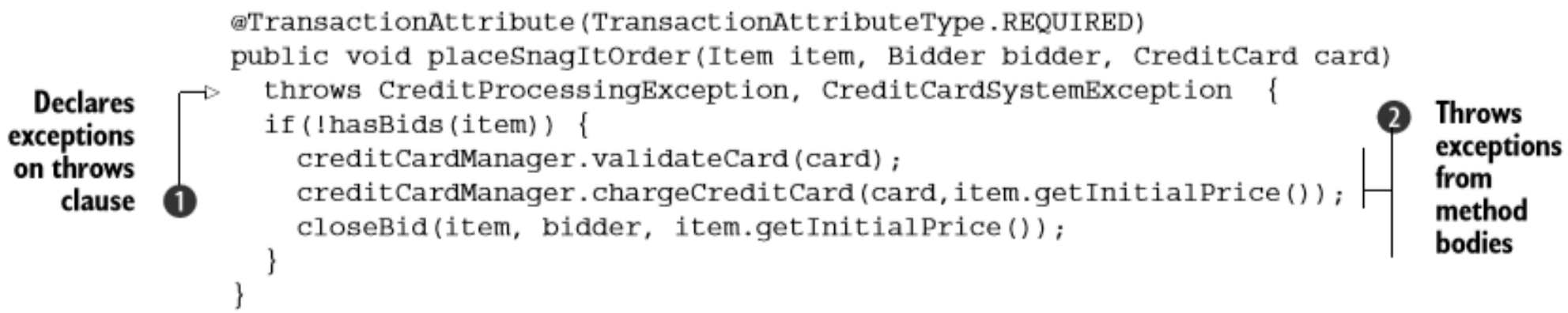
Transações: Níveis de Isolação - Problemas

- **Phanton Read Problem**
  - Quando uma transação T1, ao longo de sua execução, vê dados novos inseridos por outras transações.
  - Resolvido via SERIALIZABLE

Transações: considerações de uso

- Como mudar o nível de isolação?
  - EJB não oferece uma forma portável de definir o nível de isolação
    - Entretanto, todos os containers suportam isso de uma forma ou de outra
  - Se usando JDBC diretamente:
    - Connection.setIsolationLevel()

Exemplos de transações CMT



Transações: considerações de uso

- Em CMT, como forçar o rollback?
  - Lançar uma RemoteException
  - Lançar uma exceção de sistema (RuntimeException)
  - Lançar uma exceção de aplicação (Exception) com a annotation @ApplicationException(rollback=true)
  - Injetar SessionContext via @Resource e chamar SessionContext.setRollbackOnly()

Transações: considerações de uso

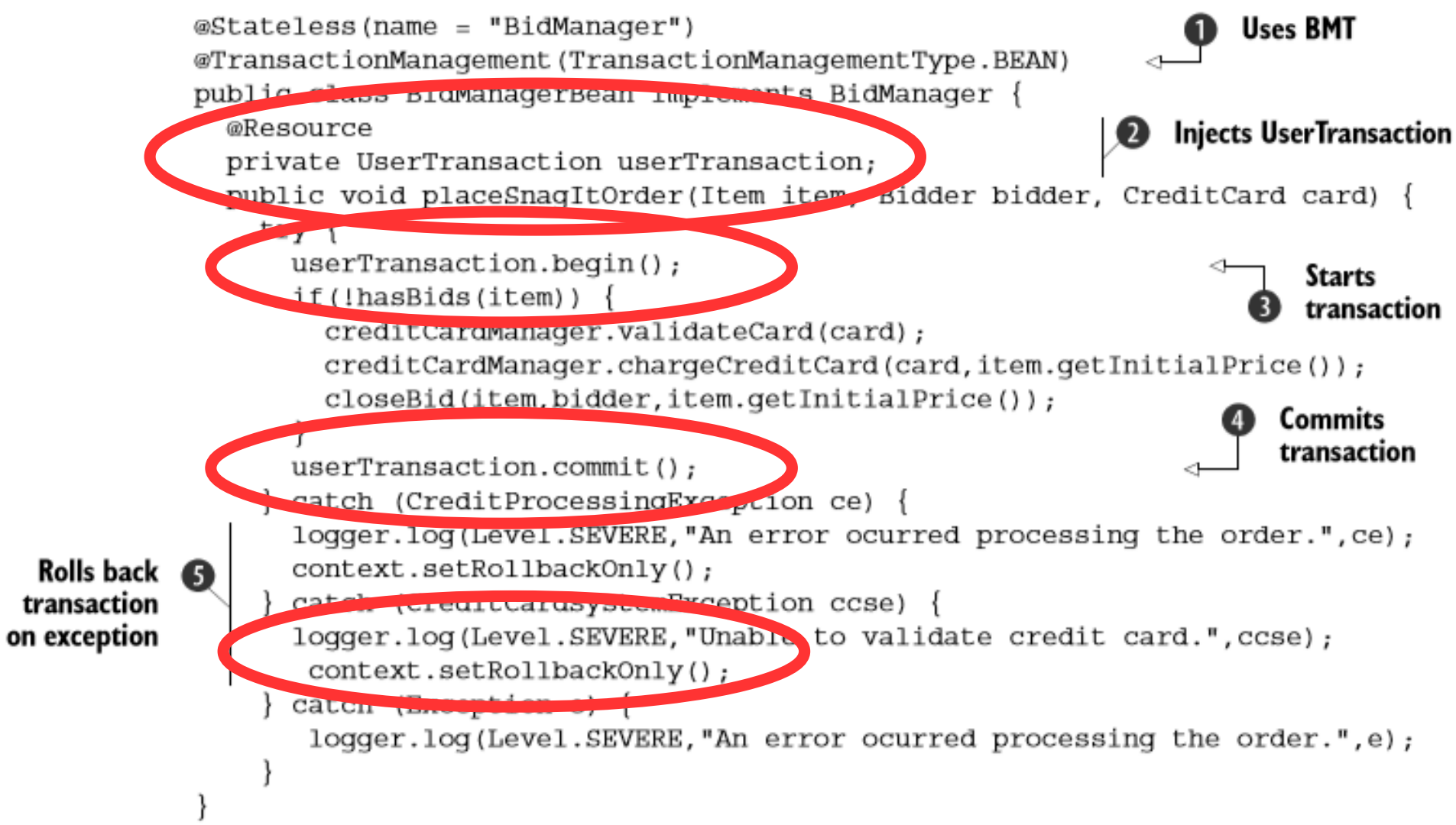
```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void placeSnagItOrder(Item item, Bidder bidder, CreditCard card)
  throws CreditProcessingException, CreditCardSystemException  {
  if(!hasBids(item)) {
    creditCardManager.validateCard(card);
    creditCardManager.chargeCreditCard(card,item.getInitialPrice());
    closeBid(item, bidder, item.getInitialPrice());
  }
}
…
@ApplicationException(rollback=true)
public class CreditProcessingException extends Exception {
…
@ApplicationException(rollback=true)
public class CreditCardSystemException extends RuntimeException {
```

2 Throws exceptions from method bodies

3 Specifies ApplicationException

4 Marks RuntimeException as ApplicationException

Transações BMT



Exercício

- Crie um EJB Stateless que seja capaz de armazenar em uma tabela de log, todos as chamadas executadas para outro EJB Stateless (utilize algum dos exercícios anteriores). Este EJB deve ser capaz de gravar este arquivo, independente do EJB ter tido sucesso na execução.

Acesso ao Ambiente de Execução

Ambiente de Execução

- Idealmente EJB's deveriam conter apenas lógica de negócio, mas nem sempre é possível. Eventualmente pode ser necessário acessar serviços do Container diretamente;
- Isso é feito via recursos especiais injetados via @Resource
  - EJBContext
    - SessionContext
    - MessageDrivenContext
  - DataSource

EJBContext

- Acesso programático a serviços de segurança, transações, timer e lookup JNDI do container;
  - Injetado via @Resource EJBContext ejbContext;
  - Métodos
    - getCallerPrincipal(): identifica o usuário autenticado
    - get/setRollbackOnly(): Marca/Consulta transação para rollback quando usando CMT
    - getUserTransaction() : Acesso a transação gerenciada pelo usuário;
    - getTimerService(): Acesso ao serviço de Timer;
    - lookup(String jndiName): lookup de beans

SessionContext

- Extensão de EJBContext com métodos específicos de session beans;
  - Injetado via @Resource SessionContext sessionContext;
  - Métodos adicionados
    - getBusinessObject(Class<?> businessInterface): usado para obter um objeto capaz de executar este bean através de sua interface de negócio;
    - getInvokedBusinessInterface(): Obtém a interface pela qual o bean foi chamado()
    - outros

MessageDrivenContext

- Extensão de EJBContext com comportamentos específicos de message driven beans;
  - Injetado via @Resource MessageDrivenContext mdContext;
  - Métodos adicionados: nenhum
  - Métodos invalidados:
    - getCallerPrincipal()
    - IsCallerInRole()

E o InitialContext?

- Este é a visão do cliente de um EJB. O InitialContext é usado para fazer lookup de um recurso ou bean;
  - BusinessInterface bi = new InitialLookup().lookup(name)

DataSource

- Usado para ter acesso direto a um datasource configurado no Container;
  - Injetado via @Resource (name=”jndiName”) DataSource ds;
  - A partir do ds pode-se obter acesso a uma Connection;
  - Mas lembre-se: não faça commit, rollback na connection. Deixe que o container gerencie a transação.
  - Se precisar indicar que a transação deve ser abortada, use o EJBContext;

DataSource no JBoss

- Configuração
  - jndi-name : Nome JNDI sobre o qual o wrapper do Datasource será vinculado. Note que este nome é relativo ao “java:/context, a menos que utilize “use-java-context” seja marcado para falso.
  - use-java-context : Se marcado como falso o datasource será vinculado ao contexto jndi global ao inves do ocntexto java
  - Exemplo: jndi-name=”TreinamentoDS”
- Uso
  - @Resource(mappedName = “java:/TreinamentoDS”) ou
  - @Resource(mappedName=”TreinamentoDS”) se use-java-context=false

DataSource no JBoss

- Uso
  - @Resource(name = “TreinamentoDS”)
    - Mapeia para:
      - "java:comp/env/nomeDaClasse/nomeDoAtributo

Concorrência

Concorrência

- Até a versão JEE 6, as únicas formas de concorrência eram implementando MDBS, ou disparando EJB´s assíncronos
- JEE 7 suporta a execução da concorrência com o uso de threads gerenciadas pelo container

```
@Resource
ManagedExecutorService managedExecutorService;
```

Concorrência

- As tarefas concorrentes devem implementar Runnable ou Callable
- Com Callable é possível aguardar pelo processamento da thread e utilizar o resultado do processamento
- Cada thread é executada fora de uma transação. Caso seja necessário uma transação, instancie o UserTransaction através do JNDI

```
public run() {
   InitialContext ctx = new InitialContext();
   UserTransaction ut = (UserTransaction)
ctx.lookup("java:comp/UserTransaction");
   ut.begin();

   // Perform transactional business logic

   ut.commit();
 }
```

Concorrência (Exemplo)

```
@Stateless
public class FaturamentoService {

	@Resource(name = "DefaultManagedExecutorService")
	private ManagedExecutorService mes;

	public BigDecimal calculaValorTotalFaturadoAnual(int ano) {

		// cria lista de tarefas
		List<Faturamento> faturamentoTotal = new ArrayList<Faturamento>();

		// inicia a criação das tarefas para faturar agrupados por mes
		for (int i = 1; i <= 12; i++) {
			faturamentoTotal.add(new Faturamento(i));
		}

		try {
			// dispara a execucao do faturamento
			List<Future<BigDecimal>> resultado = mes
					.invokeAll(faturamentoTotal);

			// consolida o resultado
			return resultado.stream().map(mes -> {
				try {
					return mes.get(); // Aguarda até a thread retornar o
					                  // resultado
				} catch (Exception e) {
					e.printStackTrace();
					return BigDecimal.ZERO;
				}
			}).reduce((mes1, mes2) -> mes1.add(mes2)).get();

		} catch (InterruptedException e) {
			System.out.println("Erro no processamento");
		}

		throw new IllegalStateException();

	}
```

Concorrência (Exemplo)

```
private class Faturamento implements Callable<BigDecimal> {
		private final int mes;

		public Faturamento(int mes) {
			super();
			this.mes = mes;
		}

		@Override
		public BigDecimal call() throws Exception {
			// recupera faturas do mes
			// processa a soma das faturas
			return BigDecimal.ZERO;
		}
	}
```

Exercício

- Crie um EJB Stateless que, dado uma lista de vendas, calcule o total de vendas por mês e retorne o resultado consolidado em um objeto com o valor de cada mês.
- Dois métodos devem ser criados: o cálculo de um deve ser executado de forma linear e o outro deve ser executado em 20 threads em paralelo.
- Crie casos de teste que exercitem os dois métodos para 1 milhão de registros.
- Intrumente o código para contabilizar o tempo gasto

REST (JAX-RS)

JAX-RS

- É um conjunto de API´s para criar web services seguindo a arquitetura REST
- REST fornece uma abordagem para a troca de informações entre aplicações diferentes e/ou distribuídas tirando proveito das características do protocolo HTTP

JAX-RS

```java
@Path("/log")
@Consumes("application/json")
@Produces("application/json")
public class LogRest {

	@EJB
	private LogRepository repository;

	@GET
	public List<Log> consultarLogs() {
		return repository.getAll();
	}

	@POST
	public Response salvar(Log log) {
		repository.save(log);
		return Response.ok().build();
	}

	@DELETE
	@Path("/{id}")
	public Response remove(@PathParam("id") Long id) {
		repository.remove(id);
		return Response.ok().build();
	}

}
```