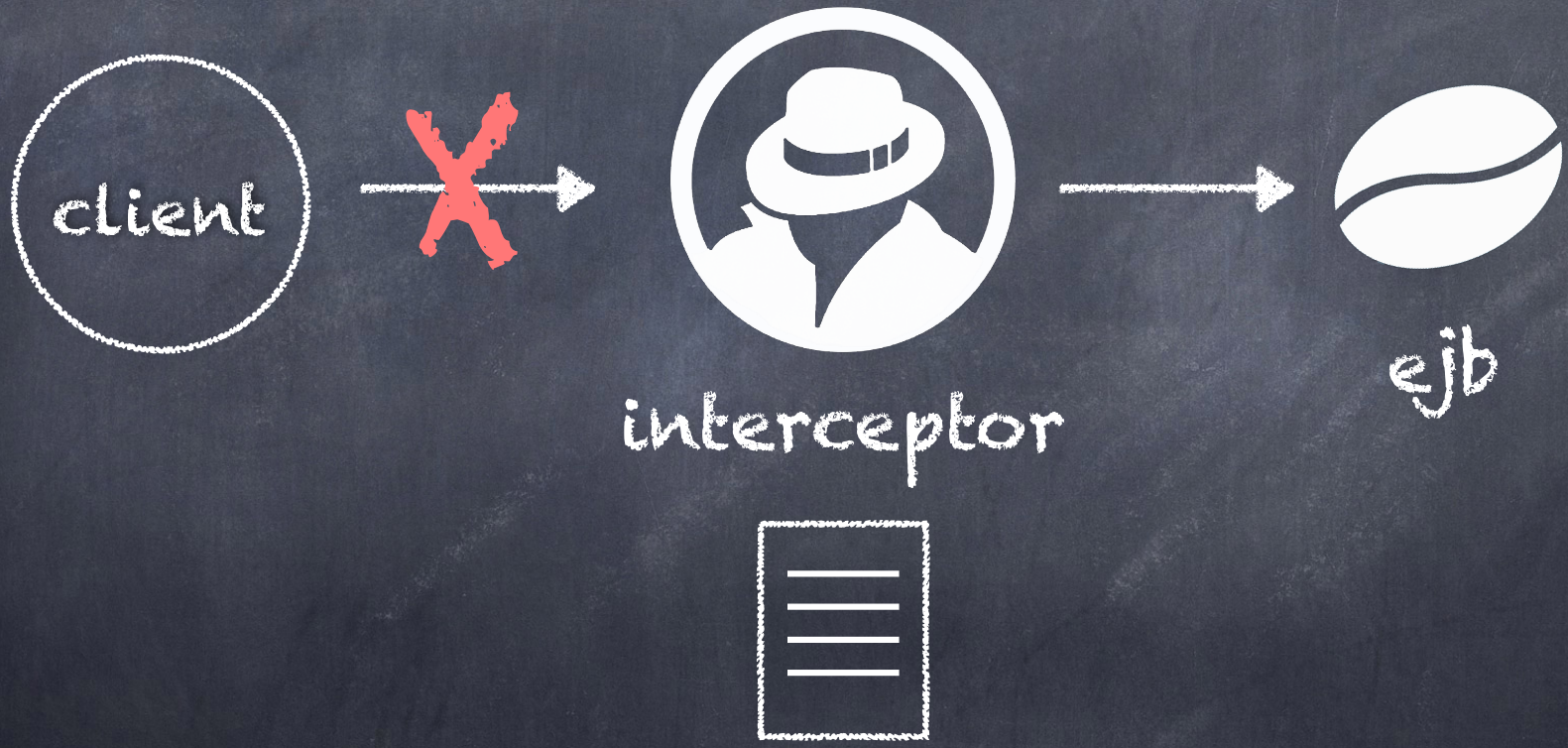
Interceptors

O uso de interceptors é uma maneira de adicionar funcionalidade a um método existente de um componente EJB sem que seja necessário alterá-lo.

Usos comuns para interceptors:

* Logging
* Auditoria
* Profiling
* Segurança


client
interceptor
ejb

```
public class LogInterceptor {

    @AroundInvoke
    public Object log (InvocationContext ctx){
        Object[] args = ctx.getParameters();
        Method method = ctx.getMethod();
        log(method, args);
        return ctx.proceed();
    }

}
```

Onde aplicar?

* Toda a aplicação (xml)
* Classe
* Método

```
@Interceptors({LogInterceptor.class})
@Stateless (name="ShoppingCart")
public class ShoppingCartImpl
                implements ShoppingCart {

    @Interceptors({SecurityInterceptor.class})
    public void process(Order order){
        entityManager.persist(order);
    }

}
```

Um interceptor padrão é declarado no ejb-jar.xml:

```
<interceptors>
   <interceptor>br.gov.fa7.Interceptor</interceptor>
<interceptors>
<assembly-descriptor>
    <interceptor-binding>
       <ejb-name> * </ejb-name>
       <interceptor-class>br.gov.fa7.Interceptor</interceptor-class>
    </interceptor-binding>
</assembly-descriptor>
```

- Interceptors possuem as mesmas callbacks de ciclo de vida de um EJB:
  - PostConstruct
  - PreDestroy
  - PrePassivate
  - PostActivate

Exercícios (4)

- Crie um interceptor padrão capaz de registrar no saída do console as seguintes informações

[data] [entrada] NomeDaClasse.nomeDoMetodo

[data] [saída] NomeDaClasse.nomeDoMetodo [tempo: xx ms]

Gerenciamento de Transações

Uma transação é um conjunto de ações que precisam ser concluídas atomicamente. Ou todas concluem, ou todas falham – deixando o estado da aplicação inalterado

Atomicidade

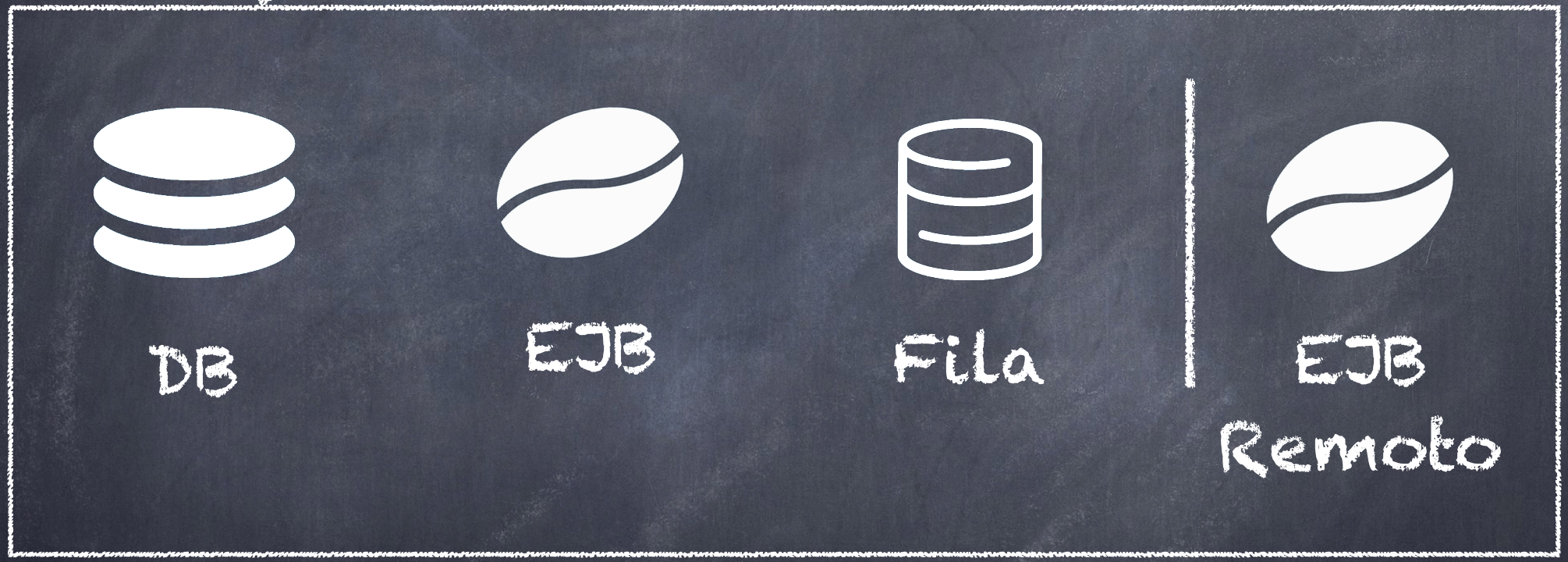
Consistência

Isolamento

Durabilidade

Atomicidade
Consistência
Isolamento
Durabilidade

transação lógica



Dois tipos de Transações

CMT -
Container Managed Transactions

BMT -
Bean Managed Transactions

```
@Stateless (name="ShoppingCart")
public class ShoppingCartImpl
                implements ShoppingCart {
    @PersistenceContext
    private EntityManager entityManager;
begin
-------------------------------------------------
    public void process(Order order){
        entityManager.persist(order);
    }
commit / rollback
-------------------------------------------------
}
```

```
@Stateless (name="ShoppingCart")
public class ShoppingCartImpl
                implements ShoppingCart {


  @TransactionAttribute (REQUIRED)
  public Item find (Long itemId){
      return entityManager.find (..., ...);
  }

}
```

```
@Stateless (name="ShoppingCart")
public class ShoppingCartImpl
             implements ShoppingCart {


  @TransactionAttribute (NOT_SUPPORTED)
  public Item find (Long itemId){
      return entityManager.find (..., ...);
  }


}
```

```
@Stateless (name="ShoppingCart")
public class ShoppingCartImpl
             implements ShoppingCart {


  @TransactionAttribute (REQUIRES_NEW)
  public Order save (Order order){
      return entityManager.merge (order);
  }

}
```

Outras opções menos usadas

* SUPPORTS

* MANDATORY

* NEVER

JPA e Transações

```
@Stateless (name="ShoppingCart")
public class ShoppingCartImpl
                implements ShoppingCart {

    @PersistenceContext
    private EntityManager entityManager;

    @TransactionAttribute (NOT_SUPPORTED)
    public void process(Order order){
        entityManager.persist(order);
    }
}
```

```
@Stateless (name="EmployeeService")
public class EmployeeServiceImpl
                implements EmployeeService {

  public void update(Employee emp){
      Employee e = entityManager.find(..., ...);
      e.setName(emp.getName());
      e.setUpdated(new Date());
  }
}
```

O que está faltando?

Controlando programaticamente as transações

* Marcar a transação para rollback

* Lançar uma exceção que provoque o rollback

```
@Stateless (name="EmployeeService")
public class EmployeeServiceImpl
                implements EmployeeService {
    @Resource
    private SessionContext context;

    public void save(Employee order){
        if (shouldRollback()) {
            context.setRollbackOnly();
        }
    }
}
```

```
@Stateless (name="EmployeeService")
public class EmployeeServiceImpl
                implements EmployeeService {

    public void save(Employee emp){
        if (emp == null) {
            throw new
                IllegalArgumentException (...);
        }
    }
}
```

> ROLLBACK

```
@Stateless (name="EmployeeService")
public class EmployeeServiceImpl
                implements EmployeeService {

    public void save(Employee emp)
                        throws ParseException{
        //erro de parse
        emp.setUpdated(parse("06/06/aaaa"));
        ...
    }
}
```

> COMMIT

```
@ApplicationException (rollback = false)
public class EmployeeNotFoundException
                extends RuntimeException {

        ...

}
```

```
@Stateless (name="EmployeeService")
public class EmployeeServiceImpl
                implements EmployeeService {

    public void find(Long id) {
        Employee e = ...;
        if (e == null) {
            throw new EmployeeNotFound...();
        } return e;
    }
}
```

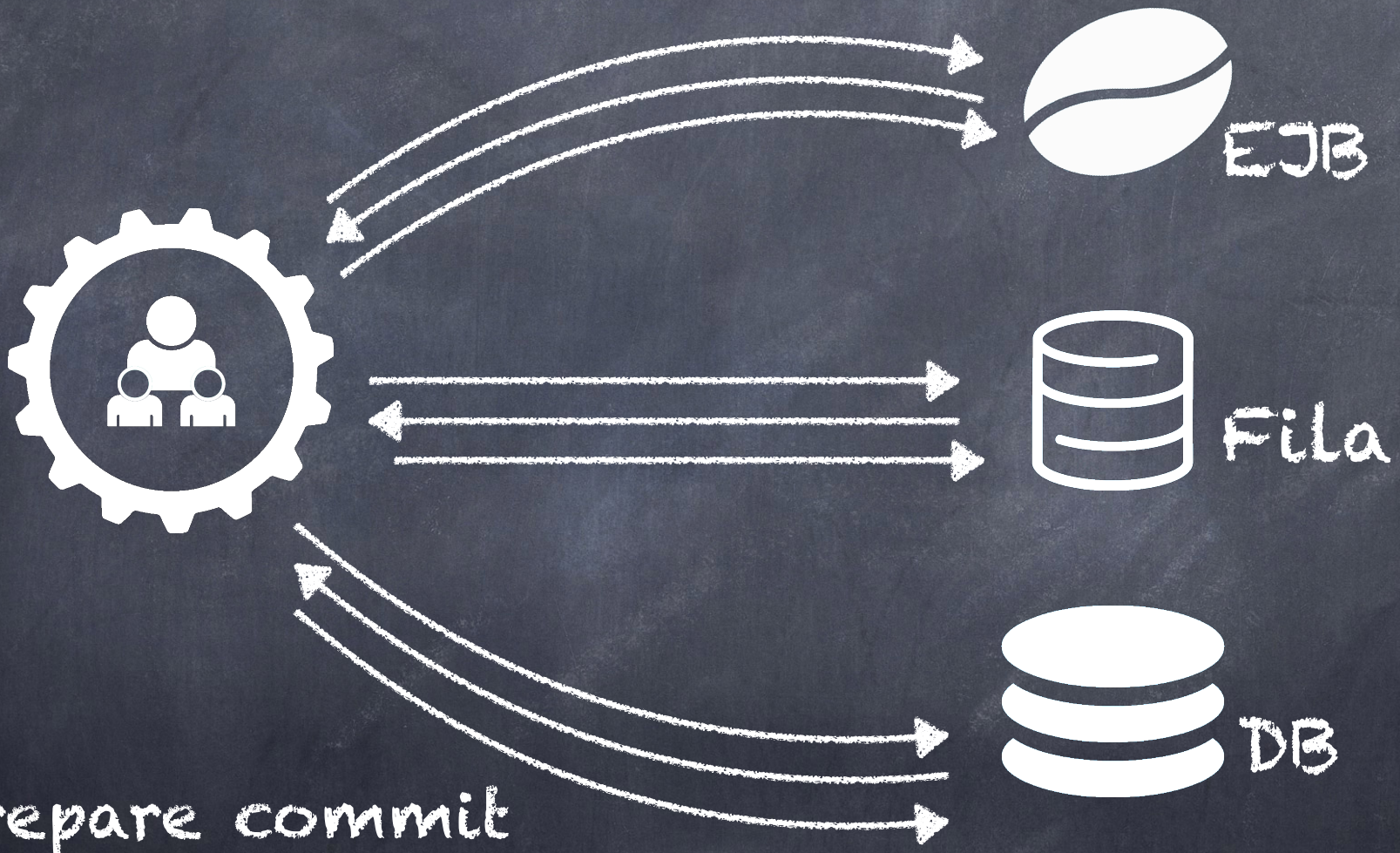
> COMMIT

É possível configurar uma exceção que não faça parte da aplicação como exceção de aplicação?

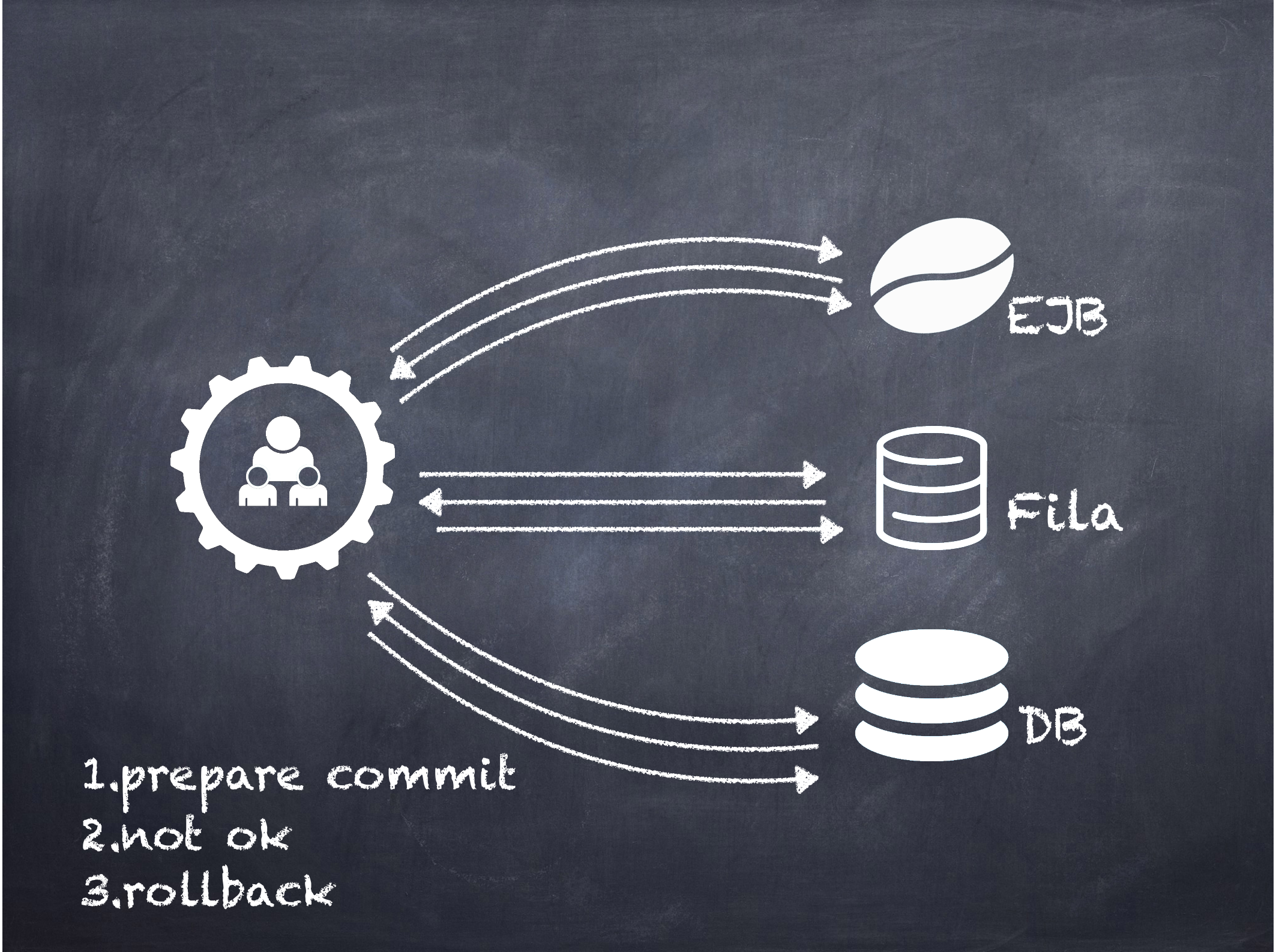
Sim, no ejb-jar.xml

```
<assembly-descriptor>
   <application-exception>
        <exception-class>java.lang.IndexOutOfBoundsException</exception-class>
        <rollback>true<rollback>
   </application-exception>
</assembly-descriptor>
```

Two-Phase Commit



1.prepare commit
2.ok
3.commit


EJB
Fila
DB
1.prepare commit
2.not ok
3.rollback

Exercícios

- Crie uma exceção chamada InvalidOperationException e lance sempre que uma operação não implementada for requisitada
  - A exceção deve herdar de RuntimeException
    - Deve ser uma exceção de aplicação.
    - A exceção é o sucesso do teste
- Dispare uma ArithmeticException caso seja solicitada uma divisão por zero.
  - Declare como exceção de aplicação
- Ambas deve realizar o rollback