

# Dominando o Controle Transacional em CAP Java

Um Guia Prático para `ChangeSet Contexts`

# O Que é um ChangeSet Context?

No coração do CAP Java, o `ChangeSetContext` é uma abstração leve para transações.

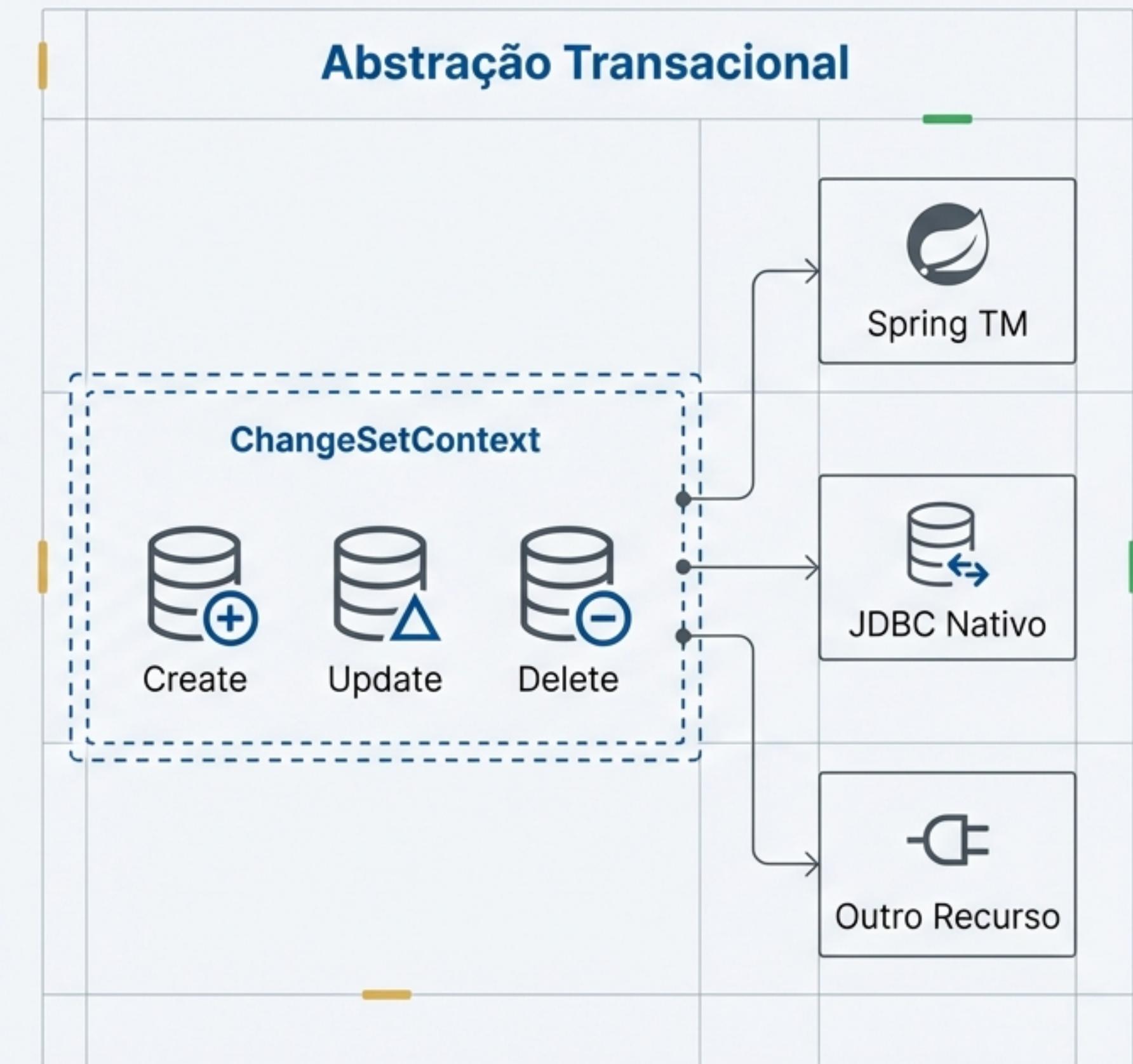
- Ele define **limites transacionais**, mas não a implementação de como uma transação é iniciada, comitada ou revertida (commit/`rollback`).
- Essa separação permite a integração com diversos tipos de gerenciadores de transações e recursos transacionais.

É a base para um sistema flexível.

**Como acessar o contexto ativo a partir do Event Context:**

```
context.getChangeSetContext();
```

## Abstração Transacional



# O Padrão: A Magia Automática do CAP

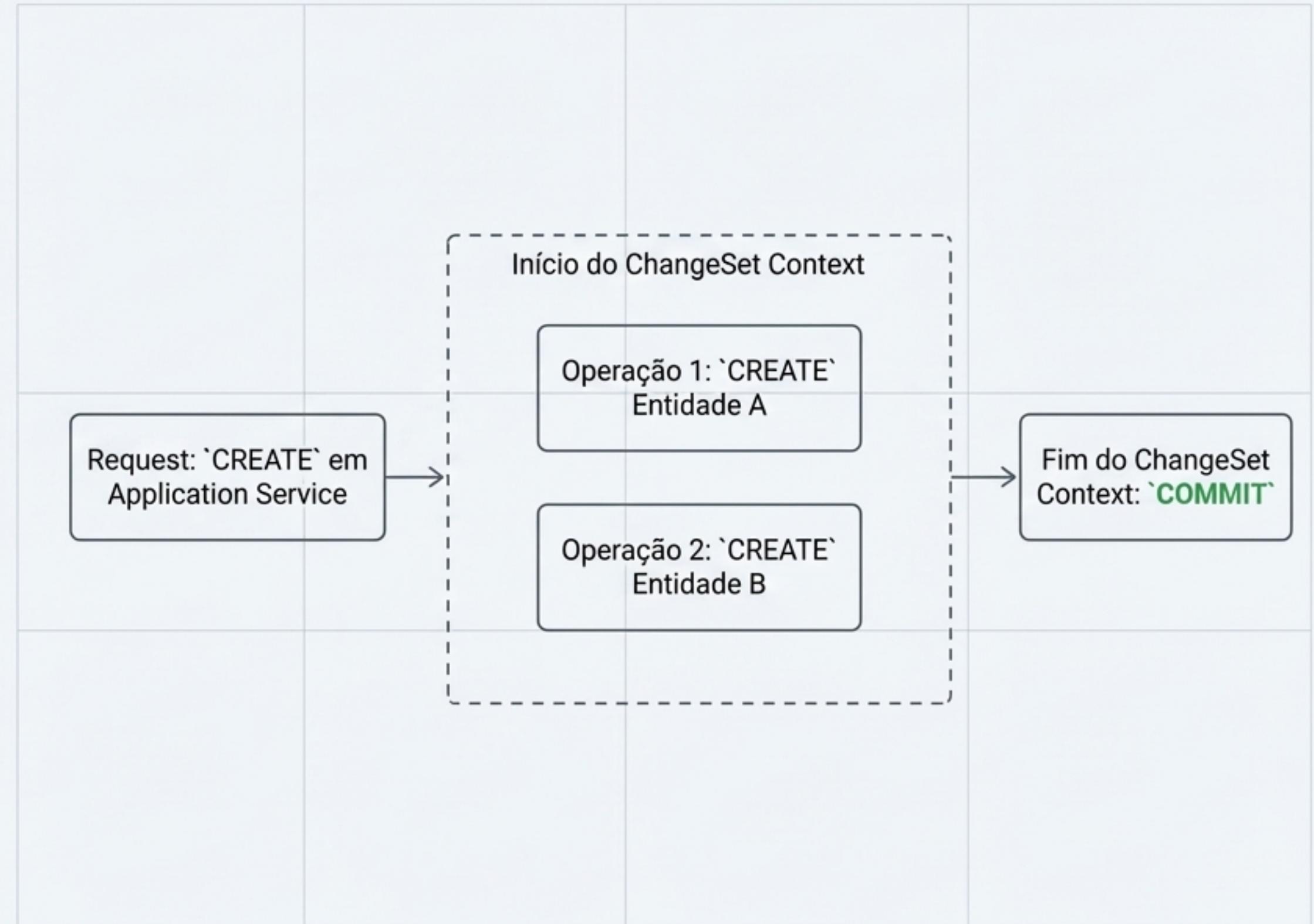
Por padrão, você não precisa se preocupar com transações. O CAP cuida disso.

Se nenhum `ChangeSetContext` estiver ativo, o CAP abre um novo para cada evento de "nível superior" ("top-level event"). Isso garante que cada evento principal seja executado dentro de seus próprios limites transacionais, de forma atômica.

## Exemplo Prático:

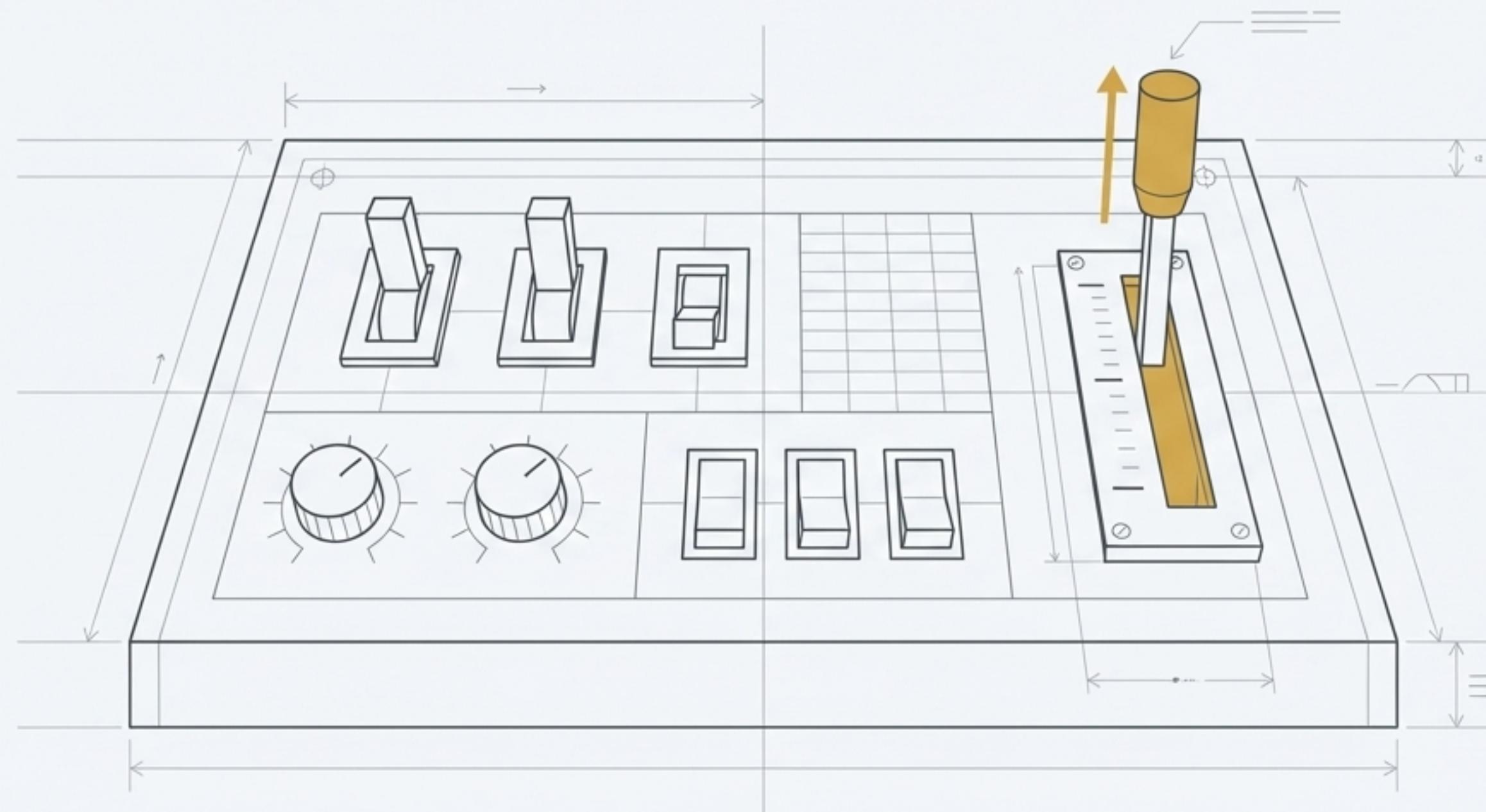
Imagine um evento `CREATE` em um *Application Service*. Esse evento pode se desdobrar em múltiplos `CREATEs` para diferentes entidades no *Persistence Service*. O CAP garante que um único `ChangeSetContext` envolva todo o processo.

Todas as interações com o banco de dados acontecem em uma única transação, que é comitada ao final do evento principal.



# Assumindo o Controle Explícito

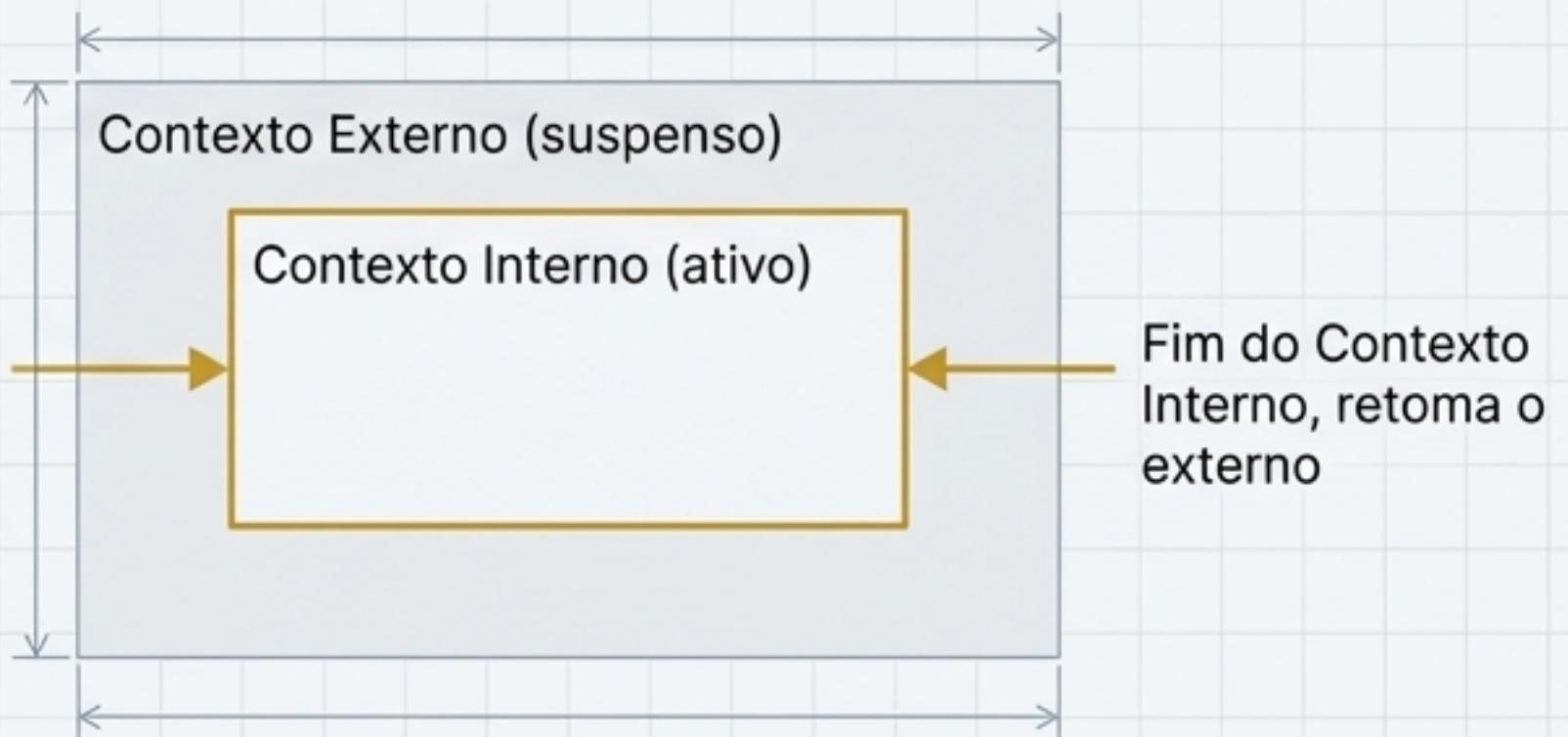
Quando o comportamento padrão não é suficiente, o CAP Java oferece as ferramentas para você gerenciar os limites transacionais.



# Criando seus Próprios Limites Transacionais

A CdsRuntime permite definir um ChangeSetContext dedicado para um bloco de código específico.

- O código executado dentro da Function ou Consumer passada para o método run() é executado em um novo contexto.
- Isso permite aninhar contextos. Um novo contexto suspende a transação anterior até que o contexto interno seja concluído.



```
// Use runtime.changeSetContext().run() para
// definir um escopo transacional.

runtime.changeSetContext().run(context → {

    // Este código executa dentro de um
    // ChangeSet Context dedicado.

    // Se já existia um contexto, ele é
    // suspenso e retomado depois.

});
```

# Reagindo ao Ciclo de Vida da Transação

É possível registrar listeners para executar ações em momentos cruciais do ChangeSetContext, como antes do commit ou após o commit/rollback.

- Use a interface **ChangeSetListener** para isso.
- **beforeClose()**: Executado momentos antes do fechamento do ChangeSet. Ideal para validações finais.
- **afterClose(boolean completed)**: Executado após o fechamento. O parâmetro **completed** indica o resultado: **true** para sucesso (commit), **false** para falha (rollback).

```
ChangeSetContext changeSet = context.getChangeSetContext();

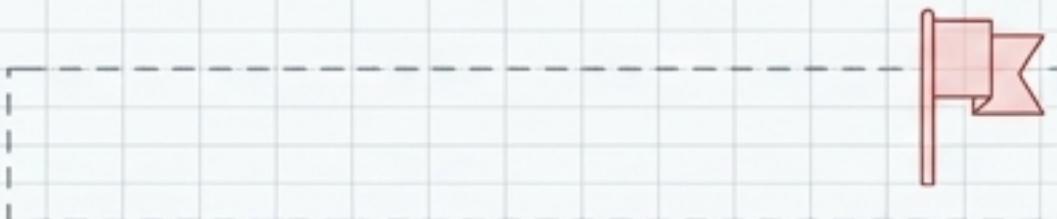
changeSet.register(new ChangeSetListener() {
    @Override
    public void beforeClose() {
        // Lógica a ser executada antes do commit.
    }

    @Override
    public void afterClose(boolean completed) {
        // Se (completed), a transação foi comitada.
        // Se (!completed), a transação sofreu rollback.
    }
});
```

# Cancelando um ChangeSet sem Exceções

Você pode marcar um `ChangeSet` para `rollback` sem precisar lançar uma exceção.

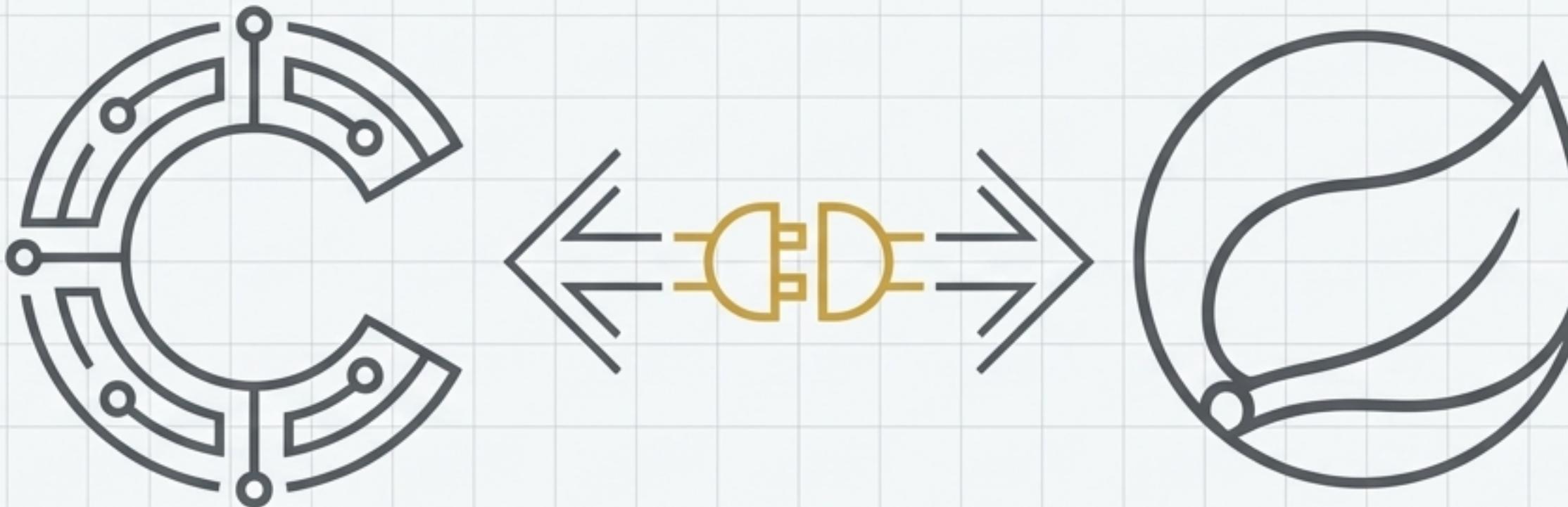
- \* Isso permite que todos os eventos dentro do `ChangeSet` sejam processados, mas garante que a transação será revertida no final.
- \* A marcação de cancelamento pode ser feita inclusive de dentro de um listener `beforeClose()`.



```
ChangeSetContext changeSet = context.getChangeSetContext();  
  
// Marca o changeset para ser revertido (rollback) no final do  
processo,  
// mas não interrompe o fluxo de eventos imediatamente.  
changeSet.markForCancel();
```

# Integração Perfeita com o Mundo Spring

Combine o poder dos `ChangeSet Contexts` com o robusto gerenciamento de transações do Spring Boot para máxima flexibilidade.



# Usando `@Transactional` do Spring em Event Handlers

Em um ambiente Spring Boot, o CAP Java se integra completamente ao gerenciamento de transações do Spring. Você pode usar a anotação `@Transactional` como uma alternativa para definir limites transacionais.

- ↪ Ao anotar um *event handler* com `@Transactional`, o Spring garante que uma transação seja inicializada.
- ↪ O CAP integra essa transação a um `ChangeSetContext` existente (se a transação ainda não havia sido iniciada) ou cria um novo `ChangeSetContext` se necessário.
- ↪ Com a propagação `REQUIRES_NEW`, o Spring e o CAP criam uma nova transação e um novo `ChangeSetContext`, suspendendo os anteriores.

**\*\*Ponto Chave\*\*:** Isso permite um controle mais “ansioso” (`'eager'`) da inicialização da transação em comparação ao comportamento `'lazy'` padrão do CAP.

# Caso de Uso: Conexões JDBC Diretas

A integração com o Spring é especialmente útil quando você precisa de acesso direto ao JDBC, por exemplo, para chamar procedures do SAP HANA ou consultar tabelas não modeladas no CDS.

- \* A anotação `@Transactional` garante que uma conexão transacional esteja disponível para ser utilizada pelo `JdbcTemplate` do Spring ou obtida via `DataSourceUtils`.

```
@Autowired  
private JdbcTemplate jdbc;  
  
@Autowired  
private DataSource ds;  
  
@Before(event = CqnService.EVENT_CREATE, entity = Books_.CDS_NAME)  
@Transactional // Garante que a transação JDBC está ativa  
public void beforeCreateBooks(List<Books> books) {  
    // Exemplo com JdbcTemplate  
    jdbc.queryForList("SELECT 1 FROM DUMMY");  
  
    // Exemplo obtendo a conexão nativa  
    Connection conn = DataSourceUtils.getConnection(ds);  
    conn.prepareCall("SELECT 1 FROM DUMMY").executeQuery();  
}
```

# Padrão Avançado: Definindo Variáveis de Sessão

Combine listeners de `ChangeSet` e a integração com o Spring para gerenciar o estado da conexão JDBC durante uma transação.

## \*\*Lógica do Padrão:\*\*

1. Um \*event handler genérico (@Before)\* intercepta as operações.
2. Ele verifica se a transação (`ChangeSetContext`) já foi 'manuseada' para evitar trabalho duplicado.
3. Define uma variável de sessão na conexão JDBC (`setSessionContextVariable`).
4. Registra um `ChangeSetListener` para limpar a variável de sessão no `beforeClose()`, garantindo que a conexão volte ao pool em seu estado original.

```
@Component  
@ServiceName(value = "*", type = PersistenceService.class)  
public class SessionContextHandler implements EventHandler {  
    private final static Set<ChangeSetContext> handled = ...;  
    @Autowired private DataSource dataSource;  
  
    @Before  
    protected void setSessionContextVariables(EventContext context) {  
        ChangeSetContext changeSet = context.getChangeSetContext();  
        if(handled.add(changeSet)) { ← 1. Executa apenas uma vez por transação  
            setSessionContextVariable("foo", "bar"); ← 3. Define uma variável  
            changeSet.register(new ChangeSetListener(){  
                @Override  
                public void beforeClose() {  
                    setSessionContextVariable("foo", null); // Limpeza  
                    handled.remove(changeSet);  
                }  
            });  
        }  
    }  
    private void setSessionContextVariable(...) { /* ... Lógica JDBC ... */ }  
}
```

← 1. Executa apenas uma vez por transação

← 3. Define uma variável de sessão

← 4. Limpa a variável de sessão no beforeClose()

# Otimização Avançada de Performance

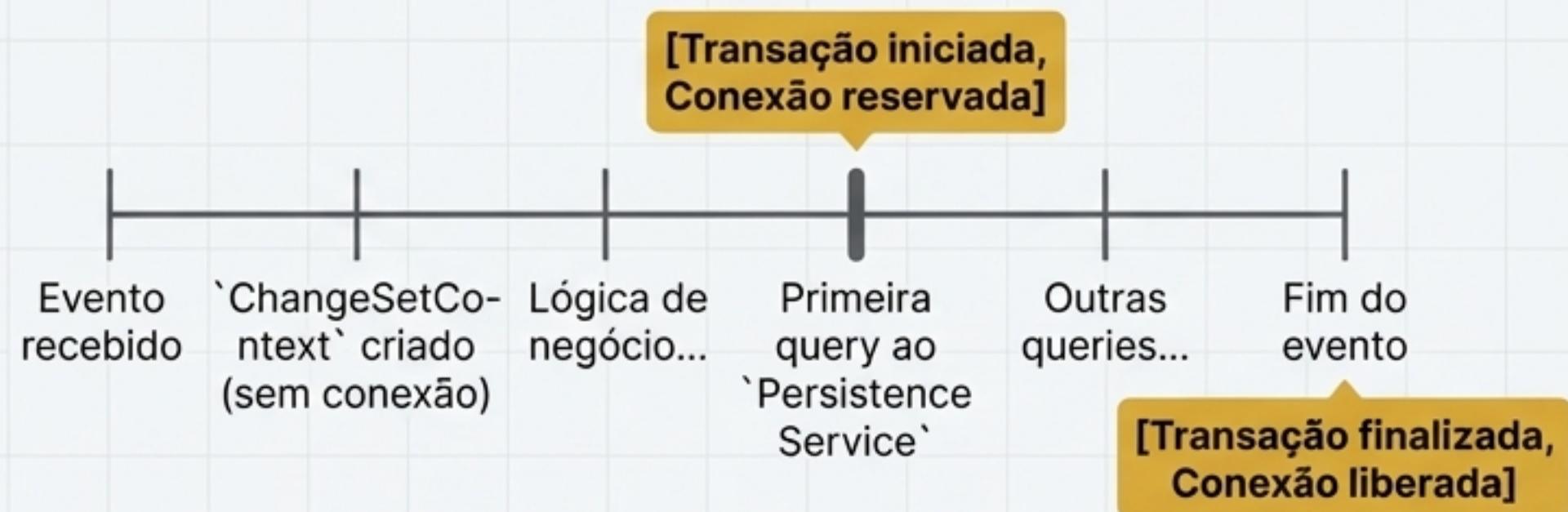
Entenda o comportamento ‘lazy’ das transações do CAP e aprenda como otimizar as operações de leitura para aumentar a concorrência.



# O Início `Lazy` das Transações no CAP

Por padrão, o CAP otimiza o uso de conexões de banco de dados.

- Um `ChangeSetContext` é sempre criado para envolver a interação com um serviço.
- **Porém, a transação no banco de dados não é iniciada imediatamente.**
- A transação só começa (e uma conexão é reservada do *pool*) **na primeira interação com o `Persistence Service`**.
- Essa conexão fica retida até o `commit` ou `rollback` final do `ChangeSetContext`.



# Aumentando a Concorrência: Evitando Transações para `SELECTs`

A maioria das consultas de leitura (`SELECT`) simples não necessita de uma transação. Reter uma conexão do \*pool\* para elas pode limitar a concorrência.

`**Fira.persistence.changeSet.enforceTransactions = false`

**A Solução:** Configure a seguinte propriedade para otimizar esse cenário:

```
cds.persistence.changeSet.enforceTransactional = false
```

## O Efeito

- A maioria das queries `SELECT` não iniciará mais uma transação.
- Uma conexão é **obtida do \*pool**, a query é executada, e a **conexão é imediatamente devolvida**.
- Isso aumenta **drasticamente** o *throughput* da aplicação, pois as conexões ficam disponíveis mais rapidamente para requisições concorrentes.
- Uma transação será iniciada normalmente assim que uma operação de modificação (`INSERT`, `UPDATE`, `DELETE`) for executada.



**Nota Importante:** Este comportamento é transparente quando se usa o nível de isolamento padrão 'Read Committed'.

# Controle Total: Exceções e Overrides da Otimização

Mesmo com a otimização ativa, o CAP é inteligente o suficiente para iniciar transações em `SELECTs` que as exigem:

- **Queries com 'lock'**: `SELECT ... FOR UPDATE` são tratadas como operações de modificação e iniciarão uma transação.  

- **Leitura de Mídia ('streams')**: Uma transação é iniciada para evitar que o `InputStream` seja corrompido caso a conexão seja devolvida ao *pool* prematuramente.  


## Forçando uma Transação Manualmente

- Se você precisar garantir que um bloco de código seja transacional, mesmo com a otimização ativa, você pode marcá-lo explicitamente.
- Use o método `markTransactional()` no `ChangeSetContext` ou `ChangeSetContextRunner`.

**Resumo:** Você tem o poder de otimizar o comportamento padrão e a flexibilidade para forçar transações quando a lógica de negócio exigir. É o melhor dos dois mundos.