

# **Dominando o Gerenciamento de Transações**

## Do Autopilot ao Controle Manual no CAP Node.js

Uma apresentação para desenvolvedores de aplicações SAP CAP.

# A Filosofia do CAP: Foco no que Importa

O CAP gerencia a complexidade por você. Como desenvolvedor de aplicações, você não precisa se preocupar com transações (ACID), propagação de contexto (usuário/tenant) ou isolamento de tenants. O runtime do CAP cuida disso disso automaticamente. A sua tarefa é focar na lógica de negócios.



*"Somente em casos rares, você precisa ir além desse nível."*

# O Autopilot em Ação: Transações Automáticas

Cada requisição de serviço opera dentro de uma transação garantida pelo framework.

## Código da Aplicação - JS

```
// Dentro de um event handler...
await db.read('Books')
```

Este simples comando desencadeia um ciclo transacional completo.

## Fluxo no Banco de Dados - SQL

```
-- ADQUIRIR conexão do pool
CONNECT; -- (se não houver uma disponível)
BEGIN;
SELECT * from Books;
COMMIT;
-- LIBERAR conexão para o pool
```

O CAP gerencia o início (BEGIN), a finalização (COMMIT) e o pool de conexões.

## DICA

Sempre que uma operação de serviço é executada, o framework garante que ela se junte a uma transação existente ou crie uma nova transação raiz. Dentro dos event handlers, seu serviço está sempre em uma transação.

# Orquestração de Múltiplos Serviços: Transações Aninhadas

**Contexto:** Serviços frequentemente processam requisições em event handlers que, por sua vez, enviam requisições a outros serviços.

## Exemplo de Código (Lógica de Transferência Bancária):

```
const log = cds.connect.to('log')
const db = cds.connect.to('db')

BankingService.on ('transfer', async req => {
  let { from, to, amount } = req.data
  await db.update('BankAccount',from).set('balance -=', amount)
  await db.update('BankAccount',to).set('balance +=', amount)
  await log.insert ({ kind:'Transfer', from, to, amount })
})
```

**Explicação:** O framework CAP orquestra três transações:

1. Uma **transação raiz** para `BankingService.transfer`.
2. Uma **transação aninhada** para as chamadas ao serviço `db`.
3. Uma **transação aninhada** para as chamadas ao serviço `log`.



**ALERTA CRÍTICO:** Não são transações distribuídas. As transações aninhadas são sincronizadas para um commit/rollback final, mas não como uma transação atômica distribuída. O commit de uma pode ter sucesso enquanto o de outra falha.

# O Ponto de Inflexão: Quando Assumir o Controle Manual?

O Autopilot é poderoso, mas cenários avançados exigem controle manual para garantir a atomicidade e o contexto corretos.



## Cenários Principais



**Múltiplas Operações:** Executar uma série de queries como uma única unidade de trabalho, especialmente fora de um event handler padrão.



**Tarefas em Background:** Executar jobs desacoplados da requisição atual (ex: envio de e-mails, processamento de filas), que não devem bloquear a resposta ao usuário.



**Contexto Específico:** Operar com um contexto de usuário ou tenant diferente do atual, como em tarefas administrativas que exigem privilégios elevados.

# A Ferramenta Principal: cds.tx() com Função

**cds.tx()** é o seu ponto de entrada para transações manuais. A forma mais segura e recomendada é passar uma função assíncrona como argumento.

```
// Ideal para executar múltiplas queries em uma única transação.  
await cds.tx(async () => {  
  const [ Emily ] = await db.insert(Authors, {name: 'Emily Brontë'})  
  await db.insert(Books, { title: 'Wuthering Heights', author: Emily })  
})
```

Mecanismo "3 em 1"



**Cria** uma nova transação raiz.



**Executa** todas as operações aninhadas dentro desta transação.



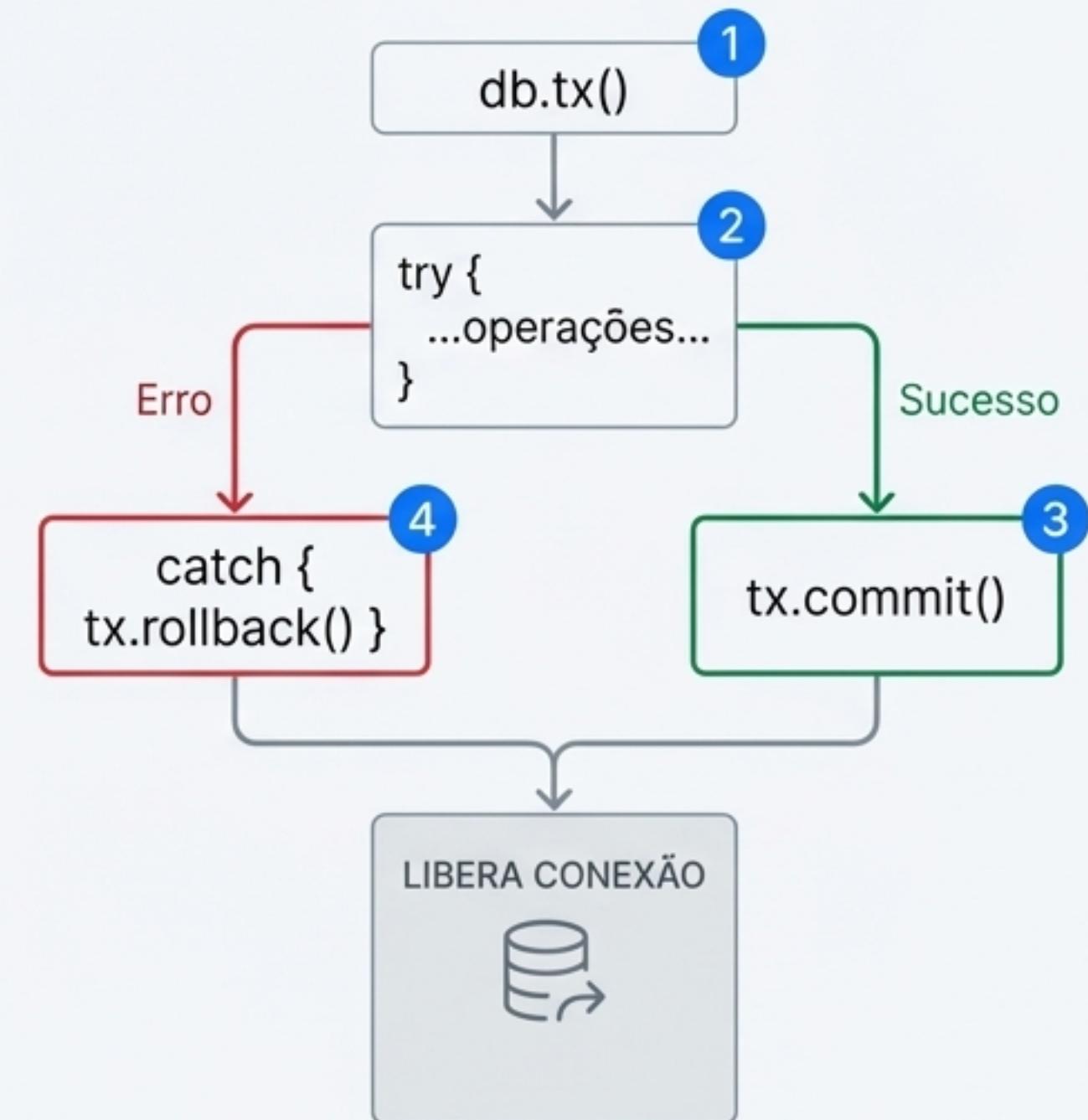
**Finaliza** a transação com `commit` em caso de sucesso ou `rollback` em caso de erro.

**Nota Importante:** Você só precisa disso em ambientes não gerenciados. Dentro de um event handler, o CAP já criou uma transação para você.

# Controle Total: O Padrão de Objeto `tx`

Para controle granular máximo, você pode obter o objeto de transação e gerenciar seu ciclo de vida manualmente.

```
let db = await cds.connect.to('db')
let tx = db.tx() // 1 Inicia a transação
try {
  await tx.run (SELECT.from(Foo)) // 2 Executa operações
  await tx.create (Foo, ...)
  await tx.commit() // 3 Efetiva em caso de sucesso
} catch(e) {
  await tx.rollback(e) // 4 Desfaz em caso de erro
}
```



**ALERTA CRÍTICO:** Você é responsável por finalizar a transação com `'commit'` ou `'rollback'`. Falhar em fazer isso resultará em transações abertas e vazamento de conexões do pool do banco de dados.

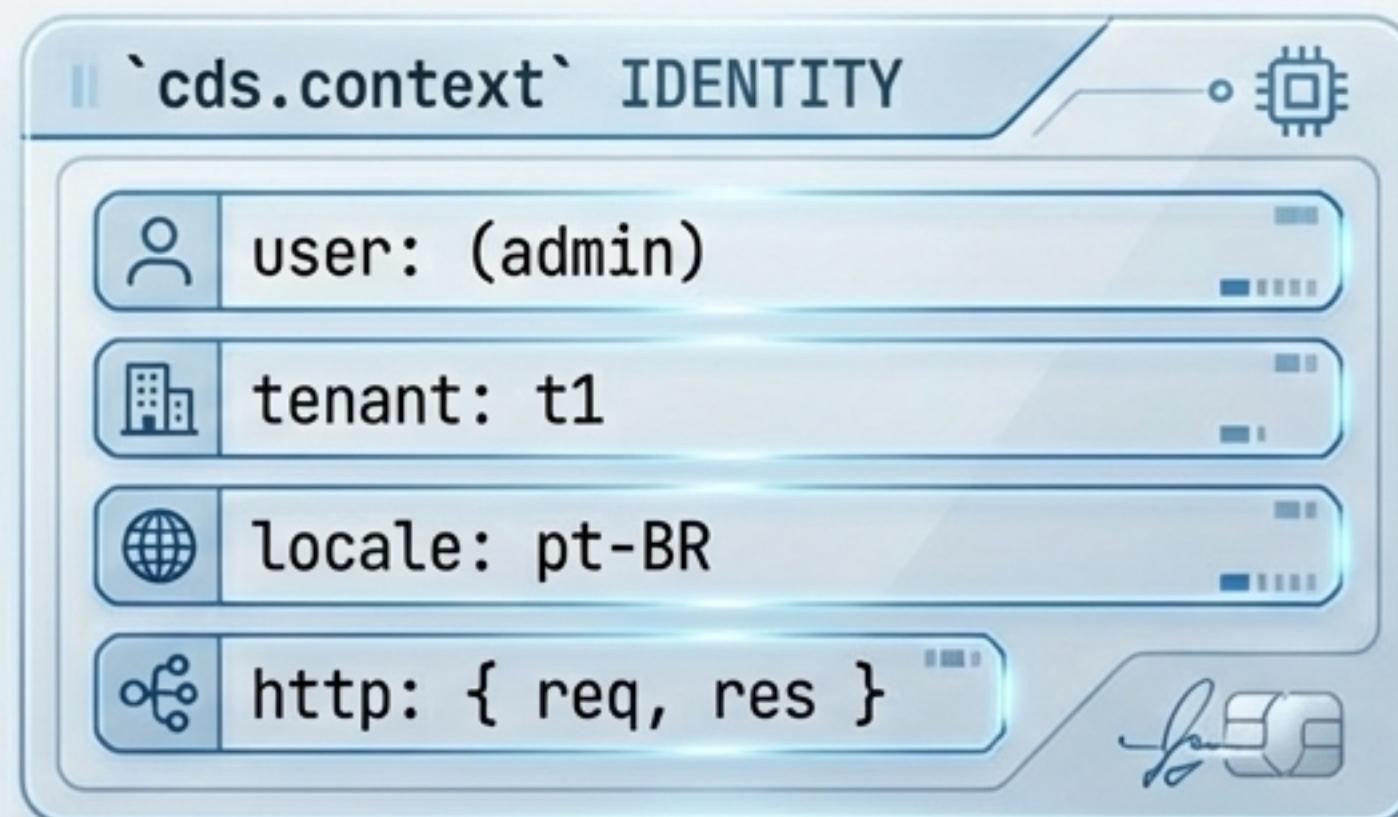
# O Painel de Controle: Entendendo `cds.context`

## Definição

O gerenciamento automático de transações precisa de acesso a propriedades do contexto da invocação. `cds.context` é o objeto que carrega essa informação.

## O que ele contém?

- O usuário e o tenant atuais.
- O locale da requisição.
- Em contextos HTTP, os objetos `req` e `res`.



## Exemplo de Acesso

```
// Acessando o usuário atual  
const { user } = cds.context  
  
if (user.is('admin')) { ... }
```

```
// Acessando objetos HTTP  
const { req, res } = cds.context.http  
if (!req.is('application/json')) res.send(415)
```

# Propagação de Contexto: A Mágica do 'Continuation-Local Storage'

## Como Funciona?

`cds.context` é implementado como uma variável *continuation-local*. Isso permite que o contexto (usuário, tenant, etc.) flua de forma transparente através de chamadas assíncronas no ambiente single-threaded do Node.js.

## Herança e Sobrescrita

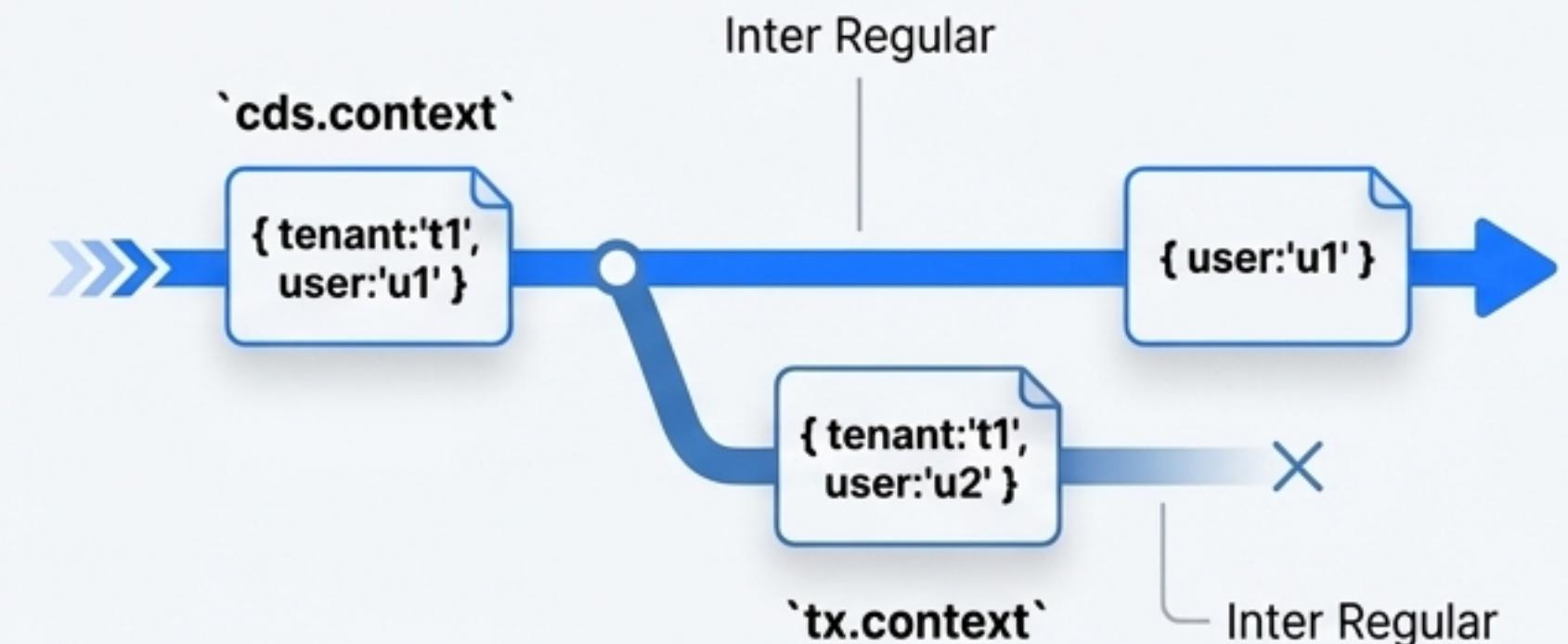
Ao criar uma nova transação com `cds.tx()`, as propriedades do contexto são herdadas do `cds.context` atual, a menos que sejam explicitamente sobrescritas.

```
// Contexto inicial
cds.context = { tenant:'t1', user:'u1' }
cds.context.user.id === 'u1' //> true

// Criando transação com usuário sobreescrito
let tx = cds.tx({ user:'u2' })

// A nova transação tem seu próprio contexto
tx.context !== cds.context //> true
tx.context.tenant === 't1' //> true (herdado)
tx.context.user.id === 'u2' //> true (sobreescrito)

// O contexto original permanece inalterado
cds.context.user.id === 'u1' //> true
```



# Operações Fora da Faixa: Jobs com `cds.spawn()`

## Definição

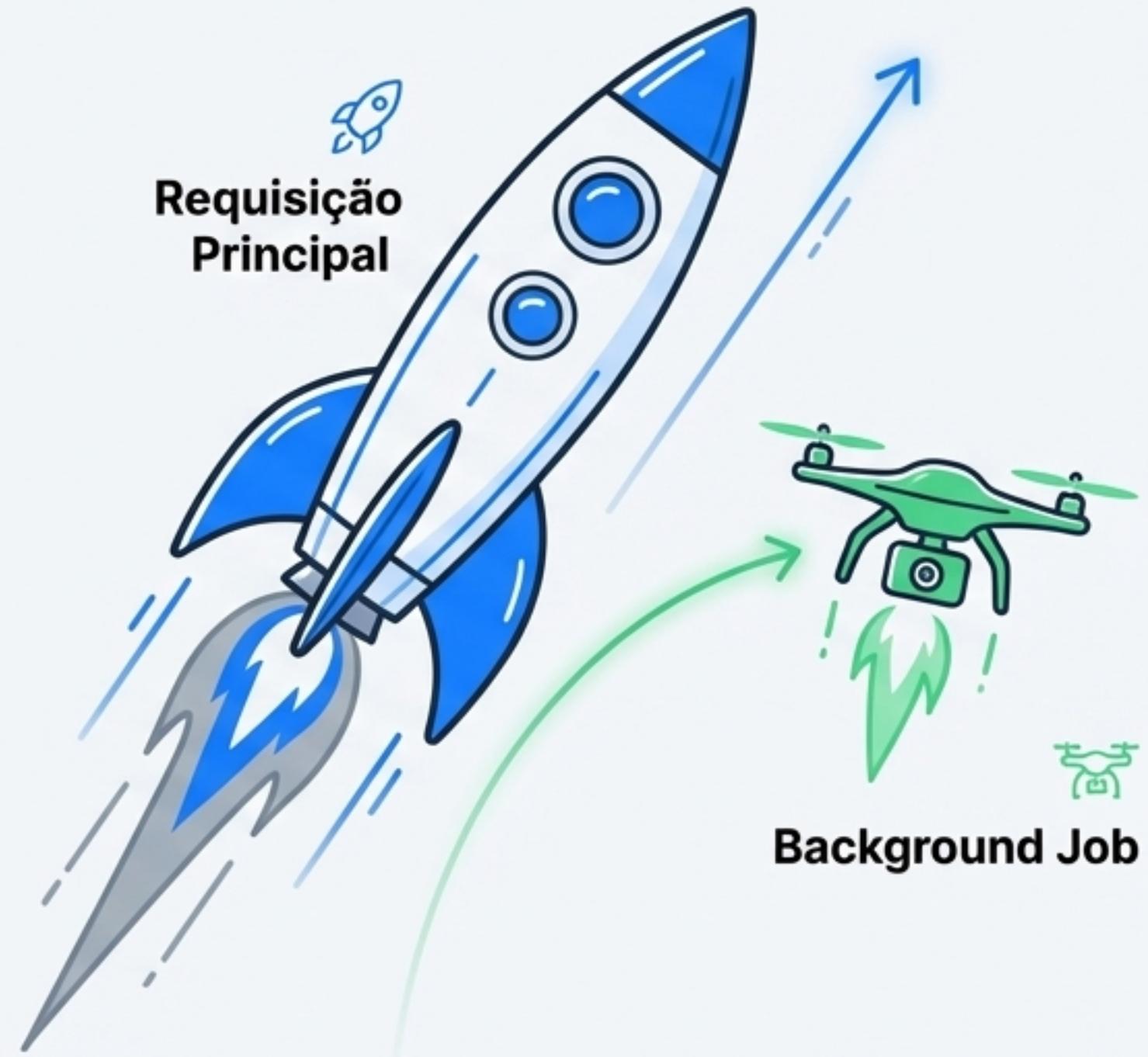
Use `cds.spawn()` para tarefas a serem executadas fora da transação atual, possivelmente com outros usuários e de forma repetida ou agendada.

## Casos de Uso Comuns

- Envio de newsletters.
- Processamento de itens em uma fila.
- Tarefas de limpeza periódicas.
- Qualquer operação que não deva bloquear a resposta da requisição principal.

## Exemplo de Código Conceitual

```
// Inicia um job que roda a cada segundo
cds.spawn ({ tenant:'t0', every: 1000 /* ms */ }, async (tx) => {
  // Lógica do job aqui...
  // Cada execução ocorre em sua própria transação 'tx'.
  const mails = await SELECT.from('Outbox')
  await MailServer.send(mails)
  await DELETE.from('Outbox').where(...)
})
```

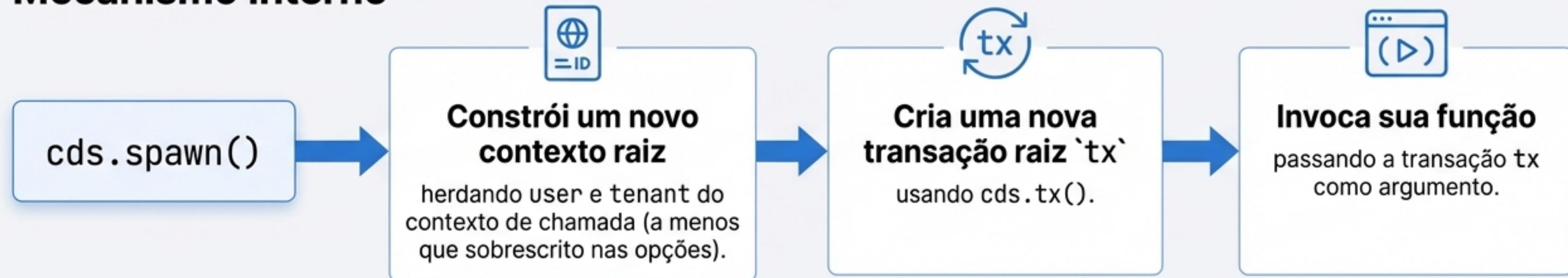


# Anatomia de um Background Job

## Garantias de Execução

`cds.spawn()` garante uma execução totalmente desacoplada da thread/continuação da requisição original.

## Mecanismo Interno



## Controle do Job

A função retorna um `EventEmitter` que permite registrar handlers para os eventos `succeeded`, `failed` e `done`. A propriedade `job.timer` pode ser usada para parar jobs periódicos com `clearInterval(job.timer)`.



**DICA:** Pense nisso como se cada execução acontecesse em sua própria thread com seu próprio contexto e gerenciamento automático de transações.

# Guia de Referência Rápida: Qual Usar e Quando?

| Cenário   | Solução Recomendada   | Nível de Controle  |
|---|---|--|
| Lógica de negócios padrão dentro de um event handler.                             | Transações Automáticas (Não fazer nada)                       |  Nenhum (Gerenciado pelo CAP) |
| Múltiplas operações atômicas fora de um event handler.                            | <code>cds.tx(async (tx) =&gt; { ... })</code>                 |  Seguro e Gerenciado         |
| Controle total do ciclo de vida (begin, commit, rollback) para lógicas complexas. | <code>const tx = db.tx();</code><br><code>try/catch...</code> |  Total e Manual             |
| Tarefas em background, agendadas ou de longa duração.                             | <code>cds.spawn({ ... }, async (tx) =&gt; { ... })</code>     |  Totalmente Desacoplado     |

# Olhando para Trás: APIs Obsoletas

## Contexto

Em bases de código mais antigas (anteriores à release 5), você pode encontrar um padrão diferente para garantir a propagação de contexto e o gerenciamento de transações. O padrão obsoleto era:  
\$srv.tx(req) → tx<srv>

```
this.on('READ', 'Books', req => {
  const tx = cus.tx(req)
  return tx.read(300)
})
```

**DEPRECATED**

## Status Atual

Este padrão ainda funciona para garantir a compatibilidade com versões anteriores, mas **não é mais necessário nem recomendado** para novos desenvolvimentos. Os mecanismos atuais de *Continuation-Local Storage* tornam isso obsoleto.

# Resumo: Você Agora Domina o Fluxo de Transações no CAP



## Confie na Automação

Para a maioria dos casos, o CAP gerencia tudo para você. Deixe o piloto automático fazer o trabalho.



## Controle com Segurança

Use `cds.tx(fn)` para agrupar operações manuais de forma segura, com gerenciamento automático de `'commit'`/`'rollback'`.



## Entenda o Contexto

Use `cds.context` para inspecionar o estado da requisição e `cds.tx({ user, tenant })` para operações com um contexto customizado.



## Isole Tarefas

Utilize `cds.spawn()` para executar tarefas em `background` de forma robusta e isolada, sem impactar a requisição principal.

# Recursos e Documentação Oficial

Para explorar todos os detalhes, exemplos avançados e as últimas atualizações sobre o Gerenciamento de Transações no CAP, consulte a documentação oficial do projeto Capire.

[cap.cloud.sap/docs/node.js/transactions](https://cap.cloud.sap/docs/node.js/transactions)

