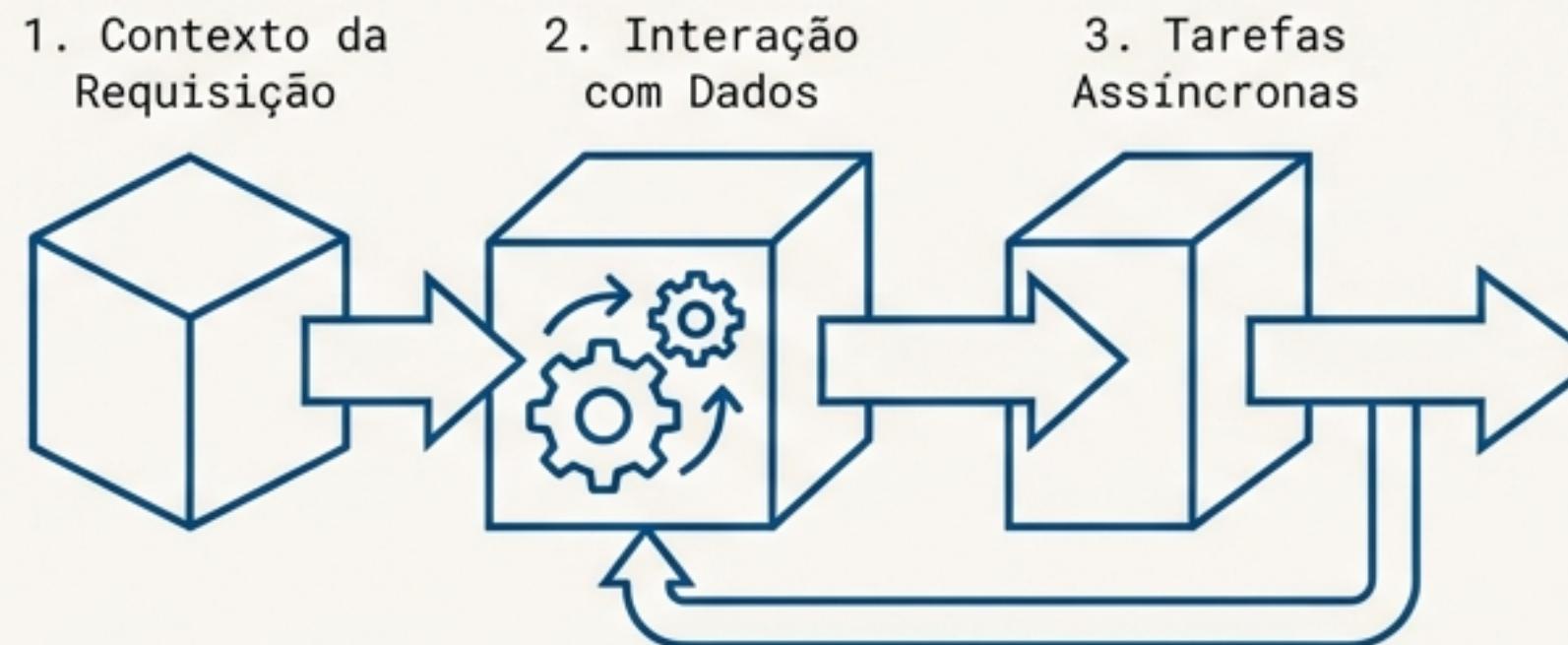


Dominando o Ciclo de Vida de Requisições no CAP Node.js

Um Guia Aprofundado sobre Contexto, Consultas e Processamento Assíncrono



Esta apresentação é uma jornada pelo fluxo de uma requisição no CAP. Vamos explorar como o framework gerencia o contexto, interage com o banco de dados de forma segura e orquestra operações em segundo plano para construir serviços robustos e eficientes.

O Ponto de Partida: Entendendo o Contexto da Requisição

`cds.context` - O Contexto Global

Fornece acesso estático ao `cds.EventContext` atual (tenant, usuário, locale) de qualquer lugar do seu código, utilizando `AsyncLocalStorage` do Node.js.

```
// Acesso global ao contexto
let { tenant, user } = cds.context;
```

`req` - O Contexto Local

Dentro de um handler, o objeto `req` herda do `cds.EventContext` e contém todas as informações da requisição atual.

```
this.on ('*', req => {
  let { tenant, user } = req; // Acesso local e mais eficiente
  ...
});
```



Boa Prática

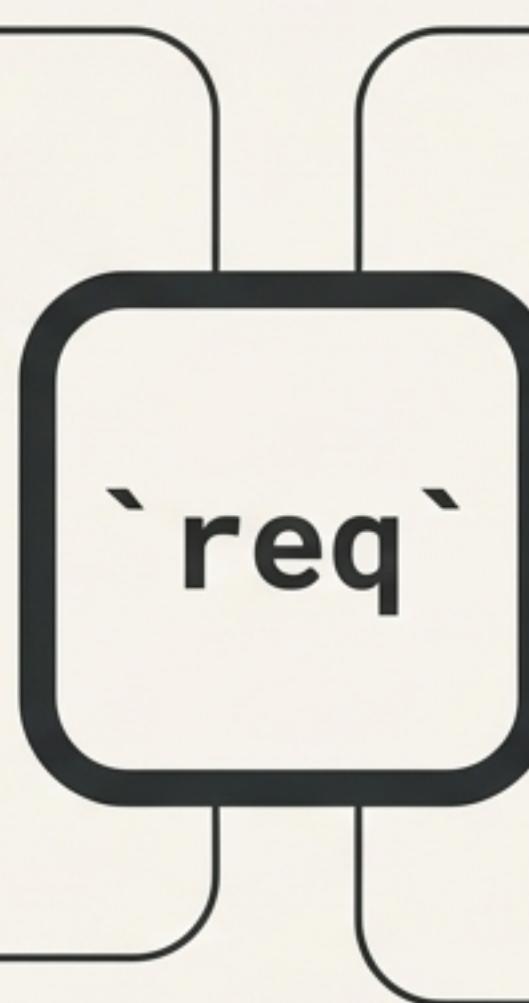
Prefira usar o objeto local `req` dentro dos seus handlers. Cada acesso a `cds.context` invoca `AsyncLocalStorage.getStore()`, o que adiciona um pequeno overhead de performance.

A Anatomia do Objeto `cds.Request`



Identidade e Contexto

- `req.user`: O usuário autenticado (`cds.User`).
- `req.tenant`: O identificador do tenant atual.
- `req.locale`: O locale do usuário, derivado do header `Accept-Language`.
- `req.id`: ID único para correlação de logs (`x-correlation-id`).



Intenção da Requisição

- `req.event`/`req.method`: O evento ou método CRUD (`CREATE`, `READ`, `POST`, `GET`).
- `req.target`: A definição da entidade de destino (do modelo CSN).
- `req.path`: O caminho canônico completo da requisição, incluindo navegações.
- `req.params`: Parâmetros da URL (ex: `Authors(101)/books(...)`).



Carga Útil (Payload)

- `req.data`: Os dados da requisição (ex: corpo de um `CREATE` ou `UPDATE`).
- `req.query`: A requisição de entrada capturada como um objeto CQN (`SELECT`, `INSERT`, etc.).



Ponteiro de Alvo

- `req.subject`: Uma referência CQN para a instância única visada pela requisição (útil em `READ`, `UPDATE`, `DELETE` de um único registro).

Finalizando a Interação: Sucesso, Erros e Notificações

`req.reply()`

Envia os resultados de volta ao cliente. Um `return` no handler tem o mesmo efeito.

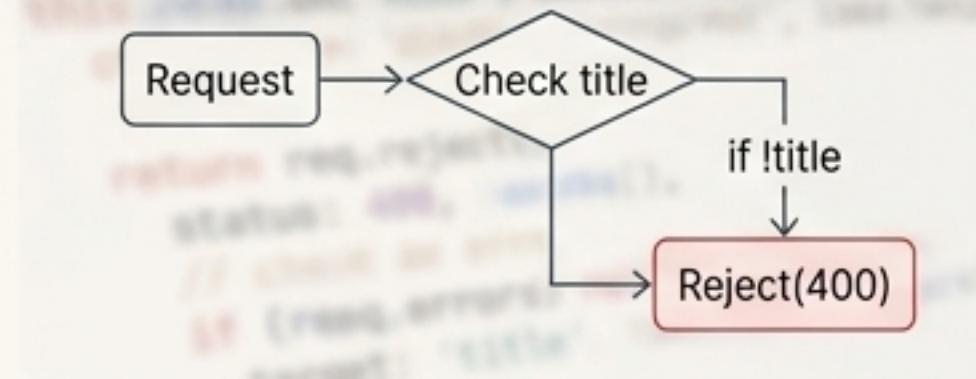
```
this.on('READ', Books, req => {
  // Ambas as formas são
  // equivalentes
  // req.reply([...]);
  return [
    { ID: 1, title: 'Wuthering
      Heights' },
    { ID: 2, title: 'Catweazle' }
  ];
});
```



`req.reject()`

Constrói e lança um erro, interrompendo o fluxo e enviando uma resposta de erro.

```
return req.reject({
  status: 400,
  code: 'MISSING_INPUT',
  message: 'Input is required',
  target: 'title'
});
```



`req.error()`

Coleta erros em `req.errors` sem interromper o fluxo imediatamente. O framework rejeita a requisição ao final da fase.



```
if (!title) req.error(400, 'Title
is missing', 'title');
if (!author) req.error(400,
'Author is missing', 'author');
if (req.errors) return; // Erros
foram registrados
```



Use `req.warn()`, `req.info()`, e `req.notify()` para enviar mensagens não-críticas que acompanham uma resposta de sucesso.

A Conversa com os Dados: Apresentando `cds.ql`

Com o contexto da requisição em mãos, a tarefa principal é interagir com a camada de persistência. O `cds.ql` é a API fluente e segura do CAP para construir e executar consultas, abstraindo a complexidade do SQL.



Segurança em Primeiro Lugar: Projetado para prevenir SQL Injection por padrão.



Flexibilidade: As consultas (`CQN`) são objetos de primeira classe. Elas podem ser construídas, modificadas e passadas entre funções.



Independente de Banco: Embora se pareça com SQL, o `cds.ql` pode ser executado contra diferentes fontes de dados, incluindo serviços remotos OData.

```
// Constrói a consulta
const query = SELECT.from(Books).where({ ID: 201 });

// Executa contra o banco de dados primário
const book = await cds.db.run(query);

// Ou simplesmente...
const book_alt = await SELECT.from(Books).where({ ID: 201 });
```

🛡 A Regra de Ouro: `cds.ql` e a Prevenção de SQL Injection

Concatenação de Strings ✗

Nunca construa consultas concatenando diretamente a entrada do usuário. Isso abre uma brecha de segurança grave.

```
// PERIGO: VULNERÁVEL A SQL INJECTION
let input = "0; DELETE from Books; --";
let books = await SELECT.from('Books').where('ID=' + input);
```

Um input malicioso como `0; DELETE from Books; --` poderia destruir seus dados.

Tagged Template Literals ✓

Use `tagged template literals` ou a API fluente. O `cds.ql` trata os valores como parâmetros de ligação (`binding parameters`), não como parte da string SQL.

```
// SEGURO: A entrada do usuário é tratada como um valor
let input = "0; DELETE from Books; --";
let books = await SELECT.from('Books').where(`ID = ${input}`);
```

Como Funciona

1. A consulta é capturada como um objeto CQN: `'{ where: [{ref:'ID'}, '=', {val: ...}] }` 
2. É traduzida para SQL com placeholders: `SELECT * from Books where ID = ?` 
3. Os valores são passados de forma **segura** como **parâmetros**, neutralizando qualquer código malicioso. 

A Flexibilidade do `cds.ql`: Escolha seu Estilo

O `cds.ql` oferece múltiplas sintaxes para construir consultas, desde uma API fluente até a construção manual de objetos CQN.

API Fluente

Ideal para construção dinâmica e legibilidade em código complexo.

```
let q = SELECT.from('Books').where({ ID: 201 }).orderBy({ title: 1 });
```

Tagged Template Literals (TTL)

Sintaxe concisa e familiar, muito parecida com SQL, mas segura.

```
let q = cds.ql`SELECT from Books where ID=${201} order by title`;
```

API Fluente Mista com TTL

O melhor dos dois mundos, combinando a estrutura da API com a concisão dos TTLs para cláusulas.

```
let q = SELECT.from`Books`.where`ID=${201}`.orderBy`title`;
```

Construção Manual de CQN

Máximo controle para cenários avançados ou geração de consultas programáticas.

```
let q = { SELECT: { from: {ref:['Books']}, where: [{ref:'ID'}, '=', {val:201}] } };
```

Dominando `SELECT`: Projeções, Expansões e Transações

Retornando um Único Registro

Use ` `.one` para indicar que você espera no máximo um resultado. A query retornará um objeto em vez de um array, ou ` `undefined` se nada for encontrado.

```
const one = await SELECT.one.from(Authors).where({ ID: 101 });
```

Projeções com Funções e Expansões

As funções de projeção são a forma recomendada de especificar colunas, permitindo `expands` (projeções aninhadas) de forma natural e tipada.

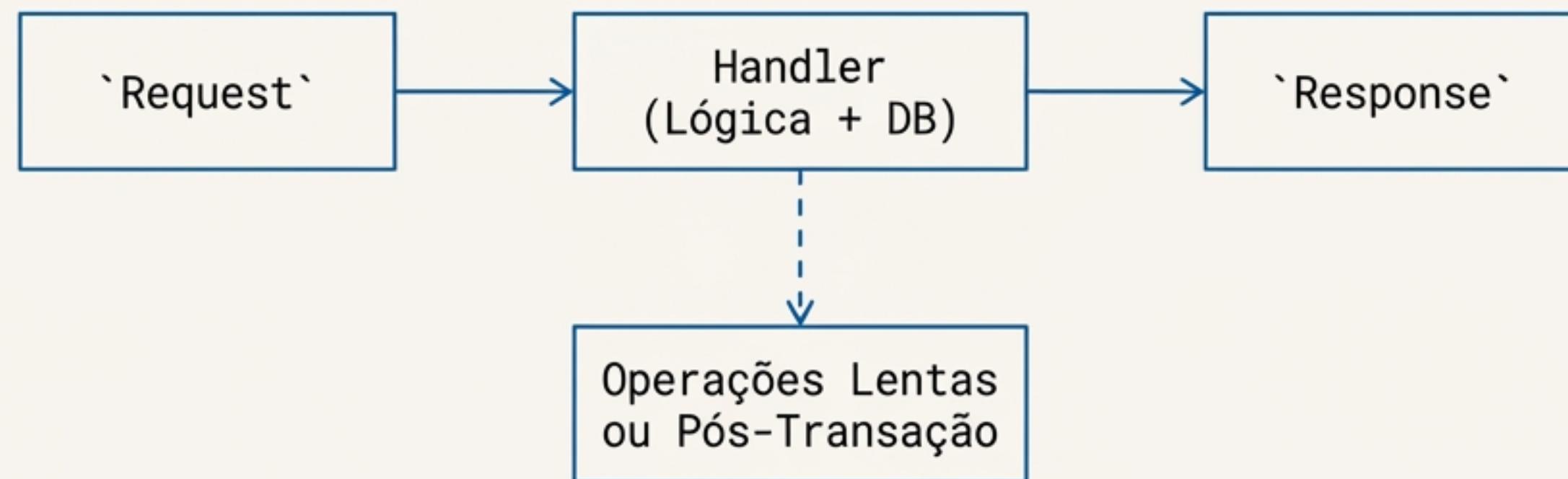
```
// SELECT from Authors a { a.name, a.books { title } }
SELECT.from('Authors', a => {
  a.name,
  a.books(b => {
    b.title
  })
});
```

Bloqueio para Atualizações (Locking)

Use ` `.forUpdate()` para aplicar um bloqueio exclusivo nas linhas selecionadas, prevenindo que outras transações as modifiquem até o `commit` ou `rollback`.

```
try {
  // Bloqueia o livro com ID 201 para esta transação
  let book = await SELECT.from(Books, 201).forUpdate();
  await UPDATE(Books, 201).with({ stock: {'-=': 1} });
} catch (e) {
  // Falha ao obter o lock (ex: timeout)
}
```

Além da Resposta Imediata: A Necessidade de Tarefas Assíncronas



O Desafio

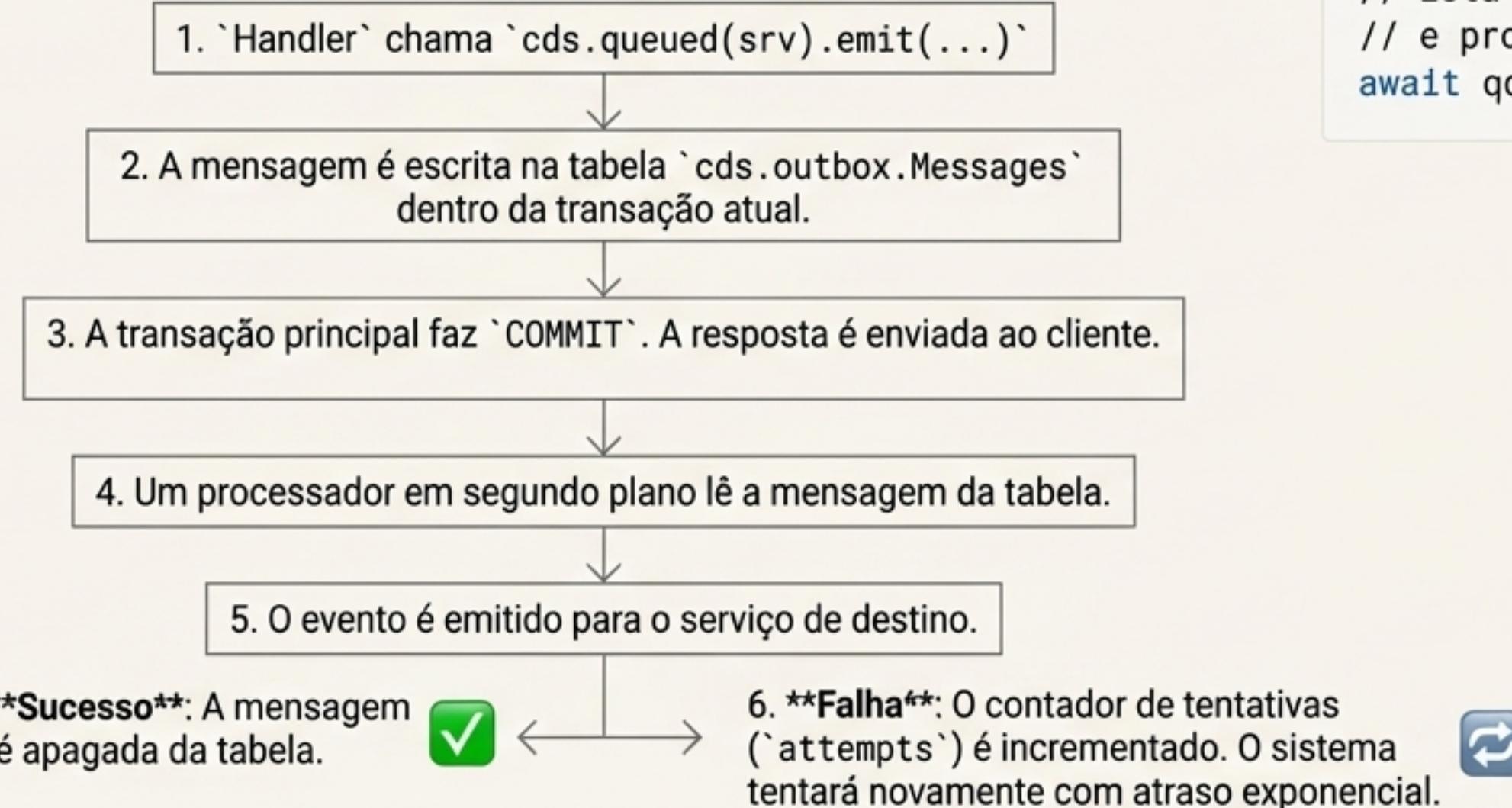
- Como lidar com operações que são lentas e não deveriam atrasar a resposta ao usuário (ex: chamar uma API externa)?
- Como garantir que uma ação só ocorra **depois** que a transação principal for confirmada com sucesso (padrão `outbox`)?
- Como construir um sistema resiliente que tenta novamente em caso de falha temporária?

A Solução CAP: `cds.queued`

O `cds.queued` implementa uma fila de tarefas que permite adiar o processamento de eventos. Em vez de emitir um evento diretamente, ele é salvo na fila e processado em segundo plano, de forma assíncrona.

O Mecanismo Padrão: A Fila Persistente (`Persistent Queue`)

Por padrão, o `cds.queued` usa uma tabela no banco de dados (`cds.outbox.Messages`) para persistir as tarefas. Isso garante que nenhuma tarefa seja perdida, mesmo que a aplicação reinicie.



```
const srv = await cds.connect.to('yourService');
const qd_srv = cds.queued(srv);

// Esta operação é assíncrona. A mensagem é salva no DB
// e processada após o commit da transação.
await qd_srv.emit('someEvent', { some: 'message' });
```

Ponto de Atenção

Handlers de eventos assíncronos rodam em modo privilegiado (sem o contexto de roles do usuário original), pois o contexto do usuário não é persistido. Apenas o `user.id` é mantido.



Gerenciando Falhas: A `Dead Letter Queue`

O Problema: Após um número máximo de tentativas (`maxAttempts`), uma mensagem para de ser processada e permanece na tabela `cds.outbox.Messages`, tornando-se uma "dead letter".

A Solução: Exponha a entidade `cds.outbox.Messages` através de um serviço customizado para permitir o monitoramento e a intervenção manual.

Passo 1: Definir o Serviço (`.cds`)

```
using from '@sap/cds/srv/outbox';

@requires: 'internal-user'
service OutboxDeadLetterQueueService {
  @readonly
  entity DeadOutboxMessages as projection on cds.outbox.Messages; });
  action revive();
  action delete();
}
```

Passo 2: Implementar as Ações (`.js`)

```
// Filtra para mostrar apenas mensagens que excederam as
// tentativas
this.before('READ', 'DeadOutboxMessages', req => {
  const { maxAttempts } = cds.env.requires.outbox;
  req.query.where('attempts >= ', maxAttempts);

// Ação para 'reviver' a mensagem, resetando as tentativas
this.on('revive', 'DeadOutboxMessages', async req => {
  await UPDATE(req.subject).set({ attempts: 0 });
});
```

Recursos Avançados: Agendamento e Outros Tipos de Fila

Recurso 1: Agendamento de Tarefas (`schedule`)

Um atalho para `cds.queued(srv).send()` que permite definir quando uma tarefa deve ser executada.

Recurso 2: Fila em Memória (`in-memory-queue`)

Uma alternativa leve que não usa o banco de dados. As mensagens são mantidas em memória até o commit da transação.

package.json

```
"requires": { "queue": { "kind": "in-memory-queue" } }
```



Não há mecanismo de `retry`. Se a emissão do evento falhar, a mensagem é perdida. Use apenas para tarefas não-críticas.

```
const srv = await cds.connect.to('yourService');

// Executa imediatamente (após o commit)
await srv.schedule('someEvent', { ... });

// Executa após 1 hora
await srv.schedule('someEvent', { ... }).after('1h');

// Executa a cada hora
await srv.schedule('someEvent', { ... }).every('1h');
```

Recurso 3: Emissão Imediata (`outboxed: false`)

Desabilita o comportamento de fila para um serviço específico, fazendo com que os eventos sejam emitidos de forma síncrona.

A Jornada Completa: Construindo Serviços Resilientes



Takeaway Principal

Dominar o fluxo `Request -> QL -> Queued` é fundamental. Essa arquitetura permite separar a lógica de negócio síncrona e crítica das operações assíncronas e de longa duração, resultando em aplicações mais rápidas, robustas e escaláveis no SAP CAP.