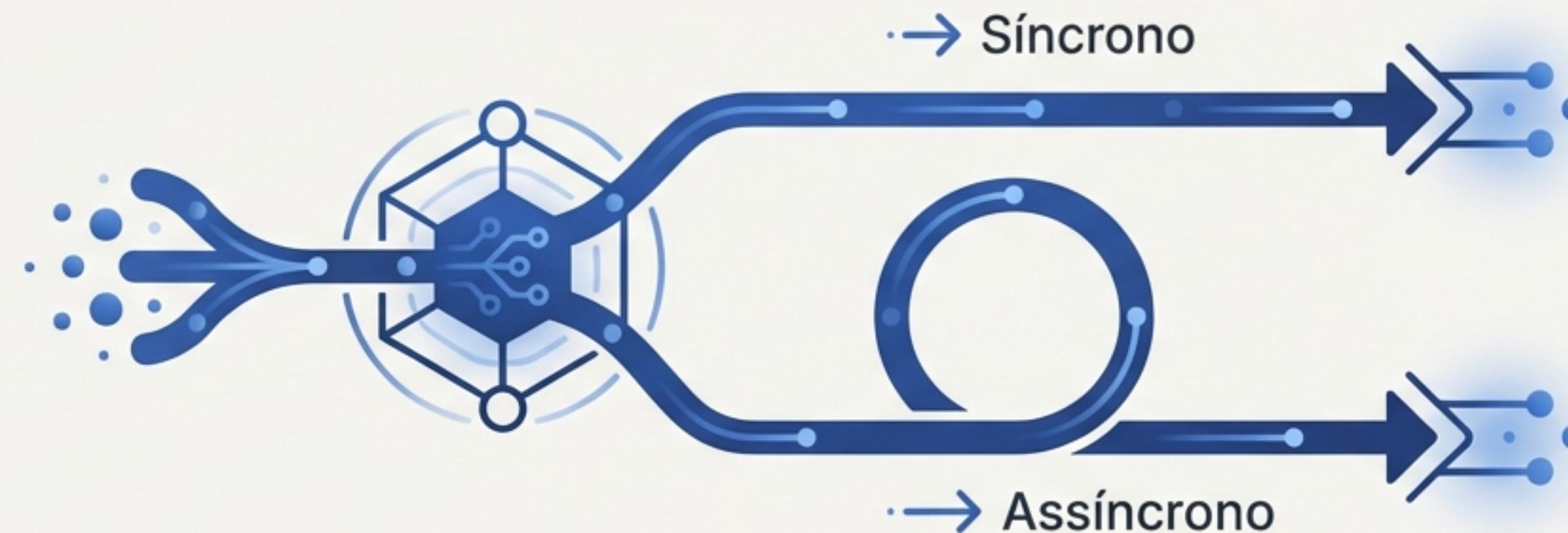


Dominando o Fluxo de Requisições no CAP Node.js

Do Síncrono ao Assíncrono: Um Guia Conceitual para Desenvolvedores

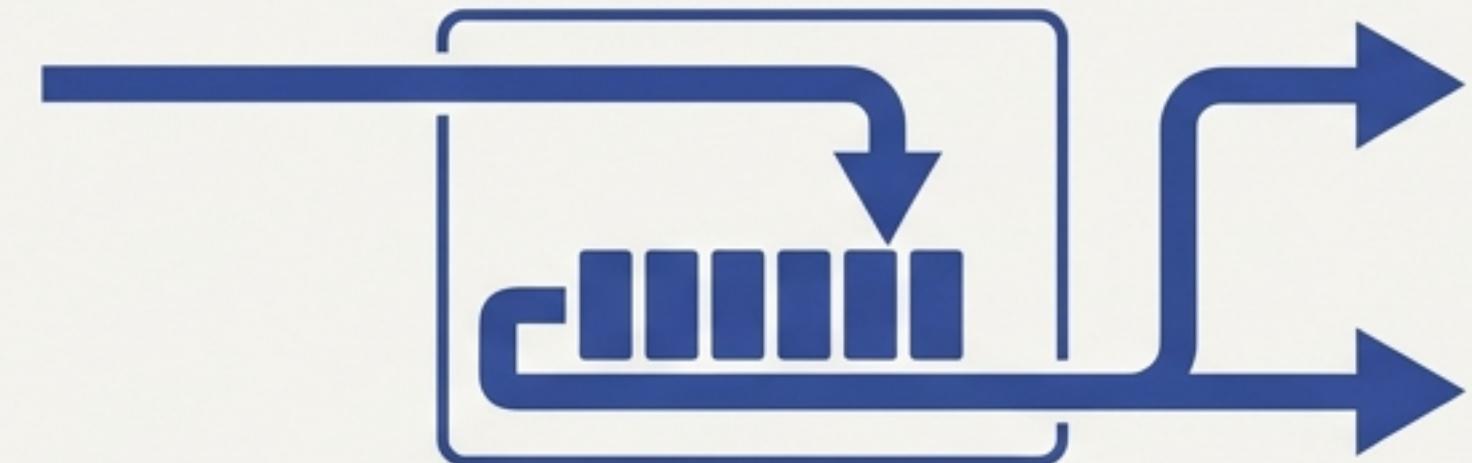


A Jornada de uma Requisição: Dois Mundos



O Mundo Síncrono: Anatomia de uma Requisição Imediata

Seguiremos o ciclo de vida de uma chamada padrão. Da chegada da requisição, passando pelo tratamento no seu serviço, até a resposta imediata. É o fundamento para construir APIs robustas e reativas.



O Mundo Assíncrono: O Poder das Filas para Tarefas Deferidas

Vamos explorar como e por que adiar o trabalho. Aprenda a usar o recurso de filas do CAP para criar sistemas resilientes e escaláveis que lidam com tarefas longas ou comunicação com sistemas externos.

O Ponto de Partida de Tudo: cds.context

Conceito Chave

`cds.context` oferece acesso estático ao `cds.EventContext` atual em qualquer parte do seu código. É a sua fonte para informações como `tenant`, `user` e `locale`.

Como Funciona

- É implementado usando a técnica de *async local storage* do Node.js.
- Normalmente, o contexto é preenchido por um middleware de entrada.

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

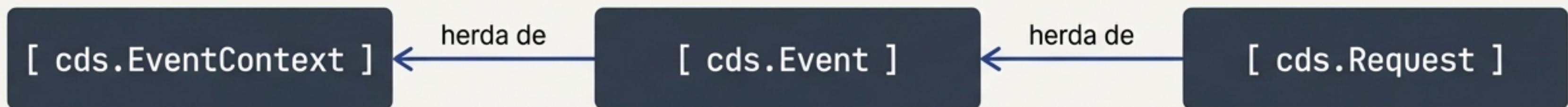
// Também funciona como um setter que instancia um EventContext
cds.context = { tenant: 't1', user: 'u2' };
const ctx = cds.context; // ctx é uma instância de cds.EventContext
```



Performance Importa!

Prefira usar o objeto `req` local nos seus handlers para acessar as propriedades de contexto. Cada acesso a `cds.context` passa por `AsyncLocalStorage.getStore()`, `e()`, o que induz um pequeno overhead.

A Anatomia de um Evento: `EventContext` e `Event`



`cds.EventContext` - O Contexto da Invocação

- 👤 .user: O usuário autenticado (instância de `cds.User`).
- 🏢 .tenant: O identificador único do tenant (se multitenancy estiver ativa).
- 🌐 .locale: O locale preferido do usuário (do header `Accept-Language`).
- 👤 .id: Um ID único para correlação de requisições (`x-correlation-id`).
- ⌚ .timestamp: Um timestamp constante para a requisição (`\$now`).
- 🌐 .http: Acesso aos objetos `req` e `res` do Express (se a origem for HTTP).

`cds.Event` - A Mensagem em Si

- 📄 .event: O nome do evento ('CREATE', 'READ', 'OrderedBook').
- 📦 .data: O payload do evento (o corpo da requisição em um `CREATE` ou `UPDATE`).
- Headers .headers: Os cabeçalhos da requisição ou mensagem.

`cds.Request`: Sua Ferramenta Principal no Handler

`cds.Request` estende `cds.Event` com propriedades e métodos para lidar com requisições síncronas a serviços.

📄 `req.query`

A requisição de entrada capturada como uma query CQN.

```
GET .../Books` se torna  
{SELECT:{from:{ref:['Books']}}}
```

📍 `req.target`

A definição da entidade de destino da requisição (refletida do modelo CSN).

```
GET .../Books(201)/author → req.target  
refere-se a `AdminService.Authors`
```

→ `req.path`

O caminho canônico completo da requisição, incluindo navegações.

```
GET .../Books(201)/author → req.path  
é `AdminService.Books/author`
```

☰ `req.params`

Acesso aos parâmetros na URL de forma iterável.

```
Authors(101)/books(title='X') →  
const [ author, book ] = req.params;
```

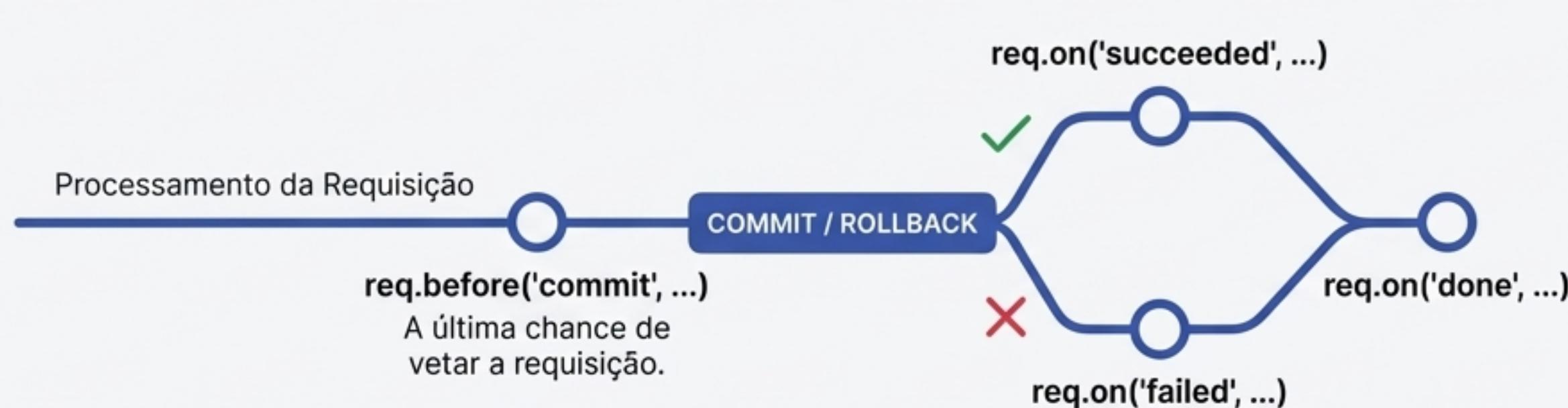
👤 `req.subject`

Um ponteiro para a instância específica visada pela requisição (para `READ`, `UPDATE`, `DELETE` em um único registro).

Permite operações como:
`SELECT.from(req.subject)` ou `DELETE.from(req.subject)`

Orquestrando o Final da Transação

Você pode registrar handlers para serem executados quando todo o processamento da requisição de nível superior estiver concluído.



```
// Exemplo: Atualizar estatísticas após uma operação bem-sucedida.  
req.on('done', async () => {  
    await cds.tx(async () => {  
        await UPDATE `Stats` .set `views = views + 1` .where `book_ID = ${book.ID}`;  
    });  
});
```

⚠️ Fora da Transação!

Os eventos `succeeded`, `failed` e `done` são emitidos *depois* que a transação é finalizada. Portanto, handlers executados aqui não podem mais vetar o commit. Para executar operações de banco de dados dentro deles, use `cds.spawn()` ou inicie uma nova transação manual com `cds.tx()`.

Concluindo com Sucesso: Enviando a Resposta

Para finalizar uma requisição com sucesso, você envia os resultados de volta ao cliente. O CAP renderiza esses dados no formato específico do protocolo (ex: JSON para OData).

A Forma Explícita com `req.reply()`

Use `req.reply(results)` para definir explicitamente os dados da resposta. O controle do fluxo continua.

```
this.on('READ', Books, req => {
  req.reply([
    { ID: 1, title: 'Wuthering Heights' },
    { ID: 2, title: 'Catweazle' }
  ]);
});
```

A Forma Implícita com `return`

Simplesmente retorne um valor do seu handler. O framework o utiliza automaticamente como a resposta. É mais conciso e comum.

```
this.on('READ', Books, req => {
  return [
    { ID: 1, title: 'Wuthering Heights' },
    { ID: 2, title: 'Catweazle' }
  ];
});
```

Lidando com Falhas: `req.reject()` vs `req.error()`

`req.reject()` - Rejeição Imediata



Caso de Uso

Quando um erro impeditivo é encontrado e o processamento deve parar imediatamente.

Como Funciona

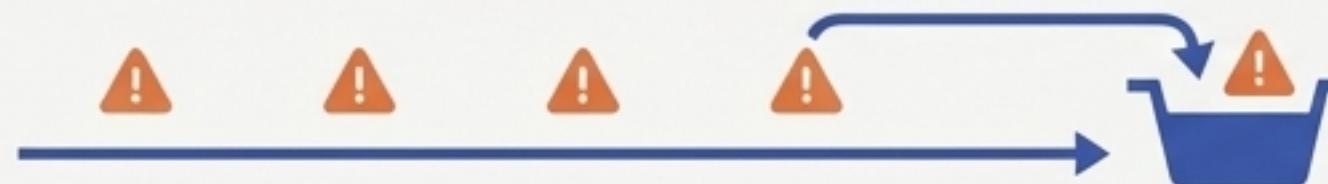
Constrói e lança uma exceção, que é capturada pelo framework e enviada como resposta de erro.

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

Sintaxe Alternativa

```
req.reject(400, 'MISSING_INPUT', 'title');
```

`req.error()` - Acumulando Erros



Caso de Uso

Para validações que podem encontrar múltiplos problemas. Você quer reportar todos de uma vez.

Como Funciona

Adiciona um erro ao array `req.errors` sem interromper o fluxo. O framework verifica `req.errors` ao final de cada fase e, se não estiver vazio, rejeita a requisição com todos os erros coletados.

```
if (!req.data.title) {  
  req.error(400, 'Missing input', 'title');  
}  
if (!req.data.author) {  
  req.error(400, 'Missing input', 'author');  
}  
// if (req.errors) ... -> verificação
```

Além dos Erros: Mensagens de Aviso e Informação

Às vezes, você precisa enviar mensagens de volta ao cliente junto com uma resposta de sucesso, não como um erro.

- `req.warn()`, `req.info()`, `req.notify()`
- Funcionam de forma similar ao `req.error()`, aceitando os mesmos argumentos.
- As mensagens são coletadas no array `req.messages` (não em `req.errors`).
- Não causam a rejeição da requisição.
- São retornadas em um cabeçalho HTTP (ex: `sap-messages`), não no corpo da resposta.

```
this.on('CREATE', Books, req => {
  if (req.data.stock < 10) {
    req.warn('Low stock for this new book');
  }
  // ... lógica de criação do livro
  return book;
});
```



Cuidado com Injeção de Dados!

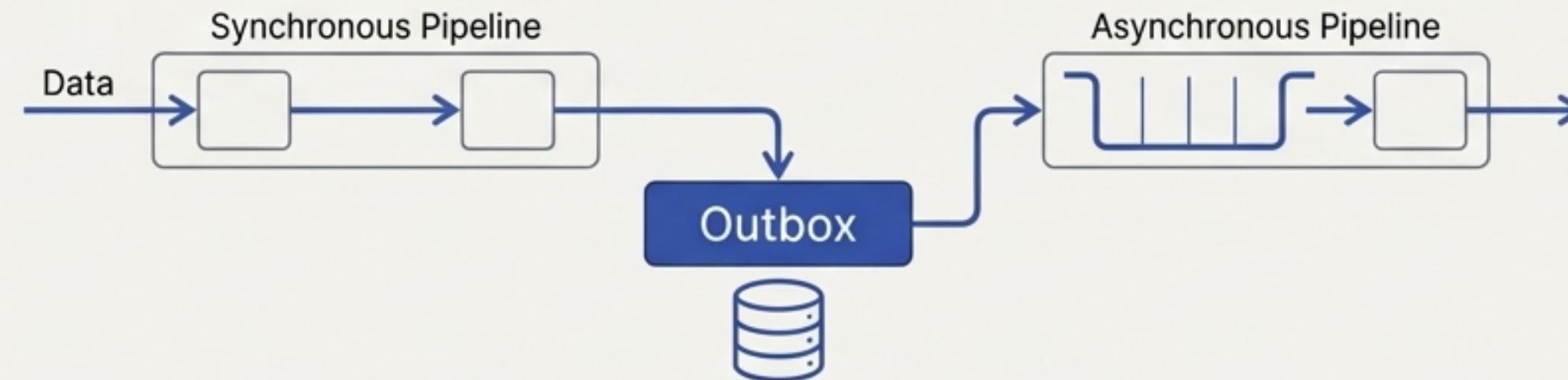
Garanta a validação adequada do texto da mensagem se ele contiver valores provenientes de entradas do usuário.

Quando a Resposta Imediata Não é Suficiente

Cenários Problemáticos

-  Como lidar com uma operação que leva vários segundos (ou minutos) para ser concluída?
-  O que acontece se você precisa chamar um serviço externo que pode estar temporariamente indisponível?
-  Como garantir que uma ação (ex: enviar um e-mail) ocorra **apenas se** a transação principal for confirmada com sucesso?

A Solução: O Padrão Outbox



Conceito

Em vez de executar a tarefa imediatamente, gravamos uma 'intenção' de executá-la em um local persistente (uma tabela no banco de dados). Um processo separado então lê essa tabela e executa a tarefa de forma assíncrona.

Benefícios

- Resiliência, desacoplamento e garantia de execução.

O CAP fornece uma implementação poderosa desse padrão através do recurso de filas.

`cds.queued`: A Porta de Entrada para o Mundo Assíncrono

Conceito Chave

Qualquer serviço CAP (que não seja de banco de dados) pode ser 'enfileirado', o que significa que o despacho de seus eventos se torna assíncrono.

Como Usar

- A função `cds.queued(srv)` retorna uma versão 'enfileirada' do seu serviço.
- Métodos como ` `.emit()` e ` `.send()`` nesta versão enfileirada não executam a lógica imediatamente.

```
// Obtenha a instância do serviço original
const srv = await cds.connect.to('yourService');

// Obtenha a versão enfileirada (assíncrona)
const qd_srv = cds.queued(srv);

// Esta chamada agora é assíncrona. A mensagem é enfileirada para
// processamento posterior.
await qd_srv.emit('someEvent', { some: 'message' });
```



‘await’ ainda é necessário!

Você ainda precisa usar 'await'. A operação é assíncrona porque, no caso da fila persistente (padrão), a mensagem precisa ser gravada no banco de dados dentro da transação atual.

Para obter o serviço original de volta, use `cds.unqueued(qd_srv)`.

Estratégias de Fila: Persistente vs. Em Memória

Fila Persistente (Padrão)



Como Funciona:

A mensagem a ser emitida é armazenada em uma tabela de banco de dados (`cds.outbox.Messages`). Após o `commit` da transação, um processo lê a tabela e despacha a mensagem.

Vantagens:

- Transacional e Resiliente:** A mensagem só é processada se a transação principal tiver sucesso.
- Mecanismo de Retentativa:** Se o processamento falhar, o sistema tenta novamente com atrasos exponenciais.
- Dead Letter Queue:** Após o número máximo de tentativas, a mensagem permanece no banco para análise.

Configuração:

Habilitada por padrão.

Fila Em Memória



Como Funciona:

As mensagens são mantidas em memória até que a transação principal seja confirmada com sucesso. É um análogo a `cds.context.on('succeeded', () => this.emit(msg))`.

Desvantagens:

- Sem Retentativas:** Se a emissão falhar, a mensagem é perdida.
- Não Persistente:** Se a aplicação reiniciar antes do `commit`, a mensagem é perdida.

Configuração (`package.json`):

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

Um Olhar Detalhado na Fila Persistente

A Entidade Central: `cds.outbox.Messages`

Seu modelo de dados é automaticamente estendido com esta entidade.

```
entity Messages {  
    key ID: UUID;  
    timestamp: Timestamp;  
    target: String;          // O serviço de destino  
    msg: LargeString;        // O payload  
    attempts: Integer;       // Contador de tentativas  
    lastError: LargeString;  
    status: String(23);  
}
```

Limitações a Conhecer



Sem Contexto de Roles

O serviço que lida com o evento enfileirado não deve depender de papéis ou atributos de usuário, pois eles não são armazenados com a mensagem. Tarefas assíncronas são sempre executadas em modo privilegiado.



Apenas User ID

O ID do usuário é armazenado para recriar o contexto básico, mas não suas permissões.

Erros Não Recuperáveis

Você pode marcar seus próprios erros como irrecuperáveis para impedir novas tentativas, definindo `error.unrecoverable = true`. O sistema então definirá `attempts` como `maxAttempts`.

Gerenciando a 'Dead Letter Queue'

O Problema : Após o número máximo de tentativas (`maxAttempts`), uma mensagem permanece na tabela `cds.outbox.Messages`. Como lidar com ela?

A Solução em 3 Passos : Implemente um serviço para expor, reviver ou excluir essas mensagens.

1 Definir o Serviço

Crie uma projeção somente leitura sobre `cds.outbox.Messages` e defina ações vinculadas.

```
.cds
using from '@sap/cds/srv/outbox';
service OutboxDeadLetterQueueService {
  @readonly
  entity DeadOutboxMessages as projection on
    cds.outbox.Messages;
  actions { action revive(); action delete(); }
}
```

2 Filtrar Programaticamente

No handler `before('READ')`, adicione a condição de filtro dinamicamente.

```
.js
this.before('READ', 'DeadOutboxMessages', function(req)
  const { maxAttempts } = cds.env.requires.outbox;
  req.query.where('attempts >= ', maxAttempts);
})
```

3 Implementar as Ações

`revive`: Zera o contador de tentativas (`SET attempts = 0`). `delete`: Remove a mensagem.

```
.js
this.on('revive', 'DeadOutboxMessages', async req =>
  await UPDATE(req.subject).set({ attempts: 0 });
  await UPDATE(req.subject).set({ attempts: 0 });
});
```

APIs Avançadas e Pontos-Chave

APIs Adicionais para Filas

Agendamento de Tarefas (`srv.schedule`)

Um atalho para `cds.queued(srv).send()` com opções de agendamento.

```
await srv.schedule(...).after('1h')
await srv.schedule(...).every('1h')
```

Processamento Manual (`.flush`)

Para acionar manualmente o processamento de mensagens da fila.

```
cds.queued(srv).flush()
```

Callbacks de Tarefas

Registre handlers para eventos de sucesso ou falha.

```
srv.after('<event>/#succeeded', ...)
srv.after('<event>/#failed', ...)
```

Resumo: Suas Diretrizes para o Fluxo de Requisições



Síncrono (`req`)

Use para controle total sobre a resposta imediata. Ideal para validações, leituras e operações rápidas onde o cliente precisa de um resultado instantâneo.



Assíncrono (`cds.queued`)

Essencial para resiliência. Use para operações demoradas, comunicação com sistemas externos ou tarefas que devem ocorrer de forma confiável após a transação principal. A fila persistente deve ser seu padrão.



Contexto é Rei

Dominar `cds.context` e o objeto `req` é fundamental para acessar informações cruciais e controlar o ciclo de vida da requisição.



Erros são Dados

Trate os erros de forma estruturada com `req.reject` e `req.error` para fornecer feedback claro e açãoável aos clientes da sua API.