

# O Foco do Desenvolvedor vs. a Magia do Framework

Uma visão geral do CAP Service SDK para Node.js

**Com o CAP, você se concentra na lógica de domínio. O resto é amplamente gerenciado pelo framework em tempo de execução.**

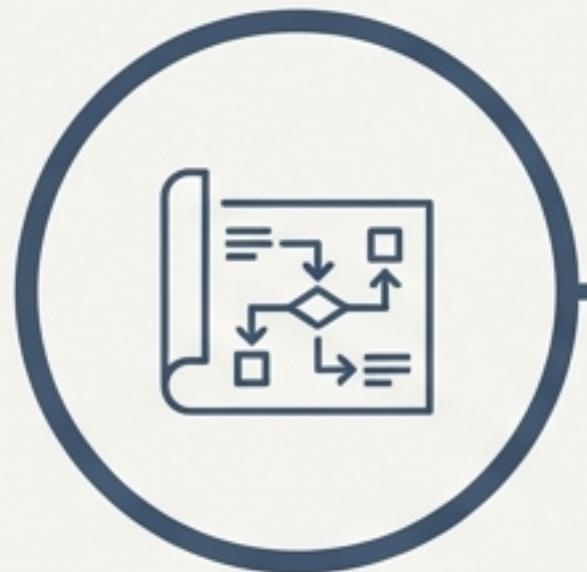
## Suas Responsabilidades Principais

-  1. Definir serviços em CDS (`cds.Service`)
-  2. Implementar handlers de eventos (`srv.on, before, after`)
-  3. Executar lógica de dados (`srv.run + cds ql`)
-  4. Interagir com outros serviços (`cds.RemoteService, cds.MessagingService`)

## Gerenciado pelo CAP Runtime

-  Bootstrapping do servidor (`cds.server`)
  -  Recursos genéricos de serviço (`cds.ApplicationService`)
  -  Gerenciamento de transações
  -  Isolamento de tenant
- E muito mais...

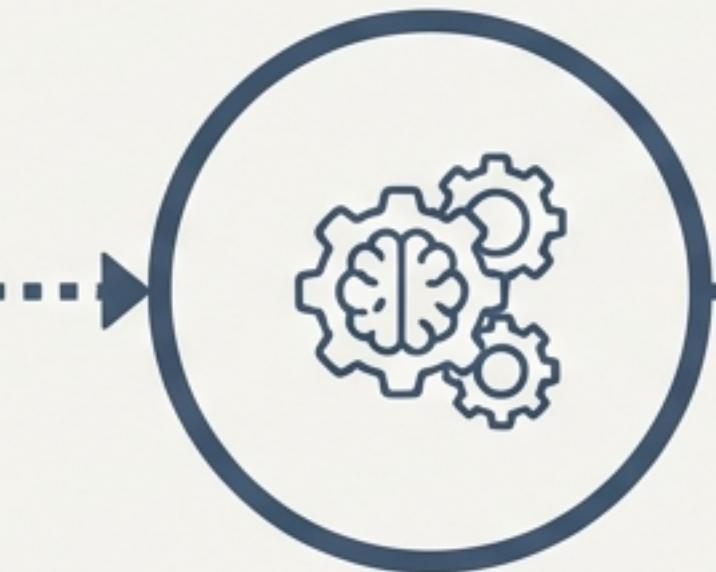
# A Jornada do Desenvolvedor: Do Modelo à Execução



## 1. Modelar & Compilar: A Fundação

Transformando arquivos `.cds` em artefatos executáveis e de persistência. A base de toda aplicação CAP.

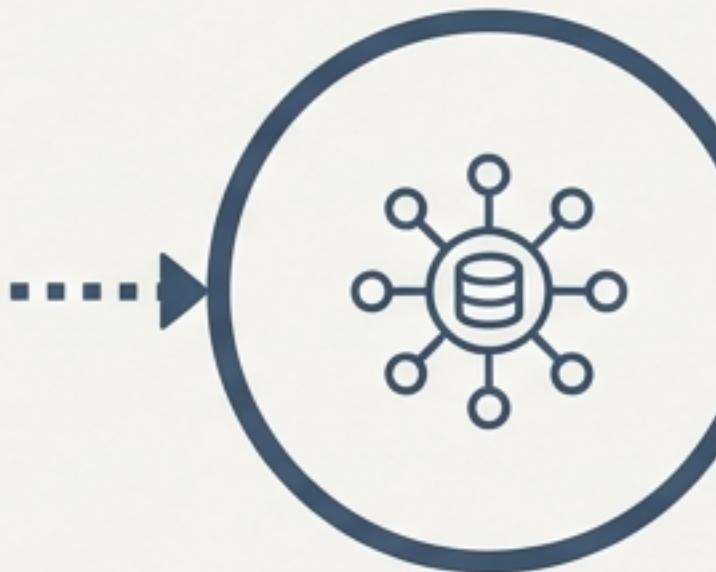
APIs em foco:  
`'cds.compile'`, `'cds.parse'`,  
`'cds.load'`



## 2. Implementar: Dando Vida à Lógica

Adicionando a lógica de negócios customizada aos seus serviços, interceptando e tratando eventos.

APIs em foco:  
`'cds.Service'`, `'srv.on'`,  
`'srv.before'`, `'srv.after'`



## 3. Interagir: Conectando o Ecossistema

Consumindo bancos de dados, serviços remotos e sistemas de mensageria de forma unificada.

APIs em foco:  
`'cds.ql'`,  
`'cds.RemoteService'`,  
`'cds.MessagingService'`



## 4. Operar: Garantindo a Robustez

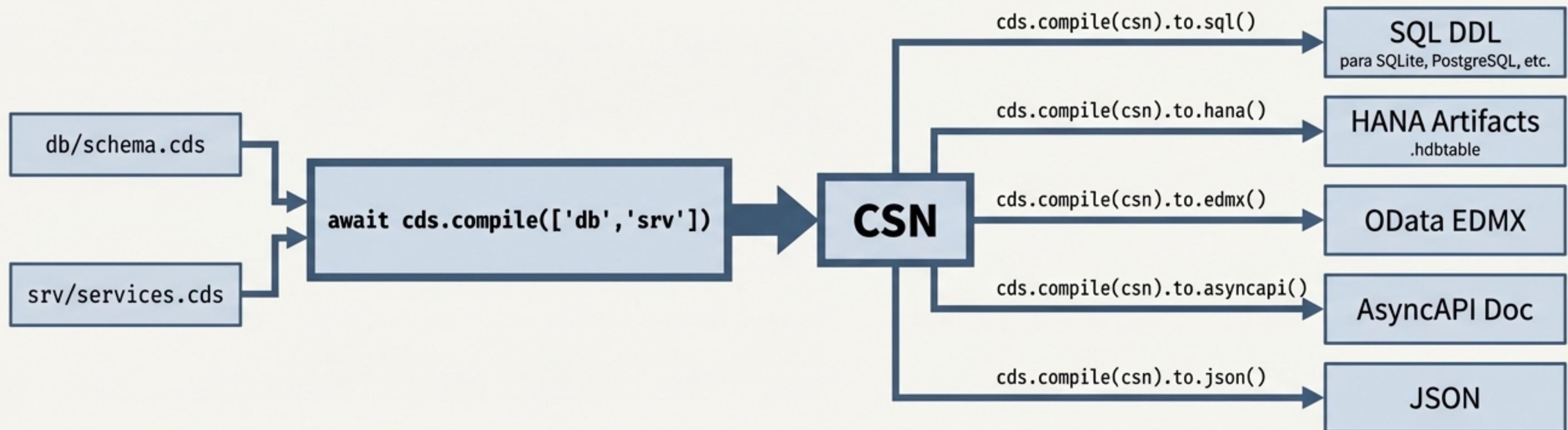
Gerenciando transações, contexto, configuração e logging para aplicações prontas para produção.

APIs em foco:  
`'cds.tx'`, `'cds.context'`,  
`'cds.env'`, `'cds.log'`

# Fase 1: Do Código CDS ao CSN e Além

## A função central `cds.compile`

`cds.compile` é a função central para compilar modelos de arquivos ou fontes em memória para o formato CSN (Core Schema Notation). O CSN é a representação universal do seu modelo dentro do CAP.



### Opções de Compilação

flavor	'inferred' (padrão, modelo efetivo com tudo aplicado) vs. 'parsed' (apenas parsing dos modelos individuais).
min	Aplica <code>cds.minify()</code> para remover definições não utilizadas.
docs	Inclui comentários de documentação ( <code>/** ... */</code> ) no CSN.

# A Caixa de Ferramentas de Compilação e Parsing

O CAP oferece um conjunto granular de APIs para interagir com seus modelos em diferentes níveis de abstração.

## Carregamento e Minificação

### **cds.load(files)**

Um atalho para `cds.compile([ ... ])` que também emite o evento `cds 'loaded'`. Resolve os nomes de arquivo usando `cds.resolve()`. É recomendado omitir sufixos para carregar arquivos CSN pré-compilados.

```
// Carrega um modelo de múltiplas fontes
const csn = await cds.load(['db','srv']);
```

### **cds.minify(csn)**

Remove tipos, aspectos e entidades não utilizadas do CSN, especialmente relevante ao usar modelos reutilizáveis como `@sap/cds/common`.

## Parsing Granular com `cds.parse`

`cds.parse` é uma fachada de API para analisar modelos CDL inteiros, consultas CQL individuais ou expressões. Suporta strings normais e *tagged template strings*.

### **cds.parse.cdl()**

Analisa uma string de sintaxe CDL e retorna um modelo CSN com flavor: 'parsed'.

```
let csn = cds.parse.cdl `entity Foo{}`;
```

### **cds.parse.cql()**

Analisa uma string de sintaxe CQL e retorna uma query CQN.

```
let cqn = cds.parse.cql `SELECT * from Foo`;
```

### **cds.parse.expr()**

Analisa uma expressão CQL e retorna um objeto CXN.

```
let cxn = cds.parse.expr `foo.bar > 9`;
// Retorna: {xpr:[{ref:['foo', 'bar']}, '>', {val:9}] }
[dev] cds repl
> let cxn = cds.parse.expr `foo.bar > 9`;
[{ref:['foo', 'bar']}, '>', {val:9} ]
```

## Eventos do Ciclo de Vida da Compilação

O CAP emite eventos que permitem estender o processo de compilação. Registre handlers com `cds.on(...)`.

- `compile.for.runtime`: Antes de compilar para o runtime Node.js/Java.
- `compile.to.dbx`: Antes de gerar artefatos de banco de dados (ex: SQL DDL).
- `compile.to.edmx`: Antes de compilar para EDMX (ex: adicionar anotações Fiori).

# Fase 2: O Universo de Serviços CAP

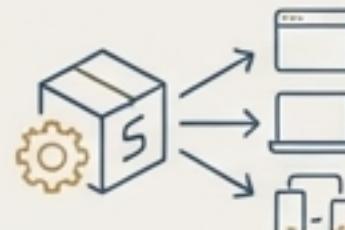
Uma aplicação CAP consiste principalmente nos serviços que ela provê e consome. O framework automatiza a criação, exposição e conexão desses serviços.

## Serviços Providos

- **O quê:** Serviços declarados em CDS que expõem a lógica de negócios da sua aplicação para clientes.
- **Como:** Definidos em arquivos `cds`. O CAP automaticamente cria instâncias de `cds.ApplicationService` e as serve em endpoints correspondentes via `cds.serve()`.

```
using { sap.capire.bookshop as my } from '../db/schema';

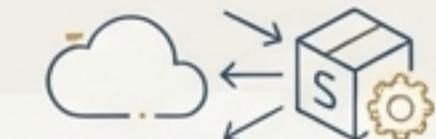
service CatalogService {
    entity Books {
        key ID : UUID;
        title : String;
        author : Association to my.Authors;
    }
    action submitOrder ( book: UUID, quantity: Integer );
    event OrderedBook: { book: UUID; quantity: Integer; buyer: User }
}
```



## Serviços Requeridos

- **O quê:** Serviços externos que sua aplicação consome, como o banco de dados primário (cds.db), outros microserviços ou message brokers.
- **Como:** Configurados na seção `cds.requires` do `package.json`. O CAP usa `cds.connect.to('ServiceName')` para estabelecer a conexão.

```
"cds": {
    "requires": {
        "ReviewsService": { "kind": "odata", "model": "@capire/reviews" },
        "db": { "kind": "sqlite", "credentials": { "url": "db.sqlite" } }
    }
}
```



```
const ReviewsService = await cds.connect.to('ReviewsService');
const db = await cds.connect.to('db');
```

# Implementando a Lógica de Negócio com Handlers de Evento

Para cada serviço definido em .cds, você pode fornecer um arquivo .js correspondente para implementar a lógica de negócios customizada, como validações, enriquecimento de dados e ações.



## Estrutura de Implementação (`cat-service.js`)

A implementação é feita estendendo cds.ApplicationService e registrando handlers de eventos no método init().

```
const cds = require('@sap/cds');
class CatalogService extends cds.ApplicationService {
  init() {
    const { Books, Authors } = this.entities;
    // Handler que executa ANTES de uma leitura de Authors
    this.before('READ', Authors, req => { /* Validação de entrada */ });
    // Handler que executa DEPOIS de uma leitura de Books
    this.after('READ', Books, each => {
      if (each.stock > 111) each.title += ' (11% off!)';
    });
    // Handler que implementa a ação customizada 'submitOrder'
    this.on('submitOrder', async req => {
      const { book, quantity } = req.data;
      // ... lógica para processar o pedido ...
    });
    return super.init();
  }
}
module.exports = CatalogService;
```

### Boas Práticas

- Use `before` para validações de dados de entrada.
- Use `after` para enriquecer os dados de saída.
- Use `after` para enriquecer os dados de saída.
- Use `on` para implementar a lógica principal de ações, funções e, em casos raros, para sobrepor o comportamento CRUD padrão.

# O Ciclo de Vida de uma Requisição: `before`, `on`, `after`

Cada requisição a um serviço passa por três fases principais, permitindo a inserção de lógica em diferentes momentos do processamento.



## `srv.before(event, entity, handler)`

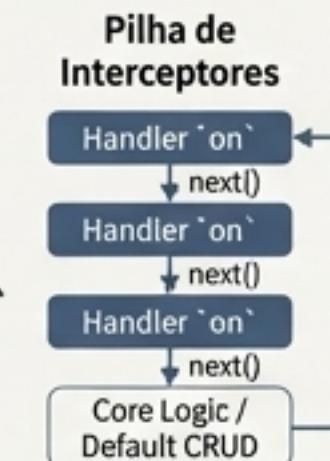
Executa antes do handler principal. Ideal para validação de dados de entrada.

```
this.before('CREATE', 'Books', req => {
  if (req.data.stock < 0) {
    req.error(400, 'Stock cannot be negative');
  }
});
```

## `srv.on(event, entity, handler)`

Implementa a lógica central da requisição. Para CRUD, o `cds.ApplicationService` já fornece uma implementação genérica. Você implementa `on` principalmente para ações e funções customizadas.

```
this.on('*', function authorize(req, next) {
  if (!req.user.is('authenticated-user')) {
    return req.reject(403, 'FORBIDDEN');
  }
  return next(); // Passa para o próximo handler
});
```



## `srv.after(event, entity, handler)`

Executa após o handler `on`. Ideal para enriquecer dados antes de enviá-los ao cliente.

Conveniência `each`: `srv.after('each', 'Books', ...)` é um atalho para um handler que executa para cada item individual em um resultado de `READ`.

# Fase 3: Consumindo Outros Serviços

Após obter uma instância de serviço com `cds.connect.to('MyService')`, você pode enviar requisições, queries ou mensagens de evento. O CAP oferece múltiplas APIs para diferentes casos de uso.

## APIs Estilo REST (`srv.send`)

Mapeamento direto para métodos HTTP. Tende a ser específico do protocolo (ex: usando query options de OData como `'\$filter'`).

```
`srv.send(method, path, data)`
```

```
const srv = await cds.connect.to  
('BooksService');  
await srv.send('POST', '/Books', {  
  title: 'Catweazle' });  
await srv.send('GET', '/Books/206');
```

Conveniência: `srv.get()`, `srv.post()`, `srv.put()`, etc.

## Prefira APIs Agnósticas de Plataforma

Enquanto `srv.send()` é útil, as APIs baseadas em `cds.ql`(`srv.run()`) são amplamente agnósticas de plataforma. Elas permitem que seu código de lógica lógica de negócios permaneça o mesmo, independentemente se o serviço consumido é um banco de dados local ou uma API OData remota.

## APIs Estilo CRUD com `cds.ql` (`srv.run`)

A abordagem preferencial e agnóstica de plataforma. As queries `cds.ql` são traduzidas para chamadas de API locais, requisições OData/GraphQL remotas ou SQL, OData/GraphQL remotas ou SQL, dependendo do serviço alvo.

```
`srv.run(CQN_QUERY)`
```

```
const { Books } = srv.entities;  
await srv.run(  
  INSERT.into(Books).entries({  
    title: 'Wuthering Heights' }) );  
await srv.run( SELECT.from(Books, 201) );
```

Conveniência: `srv.read()`, `srv.create()`, `srv.update()`, `srv.delete()` que constroem e executam queries `cds.ql`.

# `cds.ql`: A Linguagem Universal para Consultas de Dados

O módulo `cds.ql` fornece uma API para construir consultas em Core Query Notation (CQN) de forma segura e expressiva. Queries CQN são objetos JavaScript que podem ser executados em qualquer serviço CAP, seja ele um banco de dados, um serviço OData ou outro.

## Sabores do `cds.ql`

### 1. API Fluente (Fluent API)

Constrói a query passo a passo.

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

### 2. Tagged Template Literals (TTL)

Sintaxe concisa e familiar, parecida com SQL.

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

### 3. API Mista (Fluent + TTL)

Combine o melhor dos dois mundos para maior legibilidade.

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

## Execução de Queries

### Explícita

```
let results = await cds.db.run(query);
```

### Implícita

```
let results = await SELECT.from('Books').where({ID: 201});
  (Executa no `cds.db` por padrão).
```

## Segurança em Primeiro Lugar

`cds.ql` previne SQL Injection por padrão, tratando todos os valores interpolados em *tagged templates* como parâmetros de consulta.

 NUNCA FAÇA ISSO

```
SELECT.from('Books').where('ID=' + userInput)
```

 FAÇA ISSO

```
SELECT.from('Books').where(`ID = ${userInput}`)
```

# Comunicação Assíncrona via `cds.MessagingService`

Para comunicação desacoplada e baseada em eventos, o CAP utiliza o `cds.MessagingService`. Ele se integra a diversos message brokers (SAP Event Mesh, Redis, etc.) para troca de mensagens confiável.

## 1. Declarar Eventos (no CDS)

Modele eventos dentro das definições de serviço. A anotação `@topic` pode ser usada para mapear para um tópico customizado no broker.

```
service OwnService {  
    @topic: 'my.custom.topic'  
    event OwnEvent { ID: UUID; name: String; }  
}
```

## 2. Emitir Eventos (`srv.emit`)

Use o método `emit` em uma instância de serviço para enviar uma mensagem. As mensagens são enviadas após o commit da transação atual.

```
module.exports = async srv => {  
    const messaging = await cds.connect.to('messaging');  
    srv.after('CREATE', 'Reviews', async (_, req) => {  
        // ...  
        await messaging.emit('cap/review/reviewed', { subj  
    });  
}
```

## 3. Receber Eventos (`srv.on`)

Use o método `on` para se inscrever em um tópico e processar mensagens recebidas.

```
const messaging = await cds.connect.to('messaging');  
  
messaging.on('cap/review/reviewed', async msg => {  
    const { subject, rating } = msg.data;  
    await UPDATE(Books, subject).with({ rating });  
});
```

Nota sobre Contexto: Handlers de eventos assíncronos rodam com um usuário técnico (`cds.User.privileged`), portanto, quaisquer checagens de autorização devem ser feitas explicitamente.

## Protocolo CloudEvents

Para interoperabilidade, o CAP suporta nativamente o formato CloudEvents. Ative-o na configuração:

```
"messaging": {  
    "kind": "enterprise-messaging-shared",  
    "format": "cloudevent"  
}
```

# Fase 4: Gerenciamento de Transações — O Piloto Automático

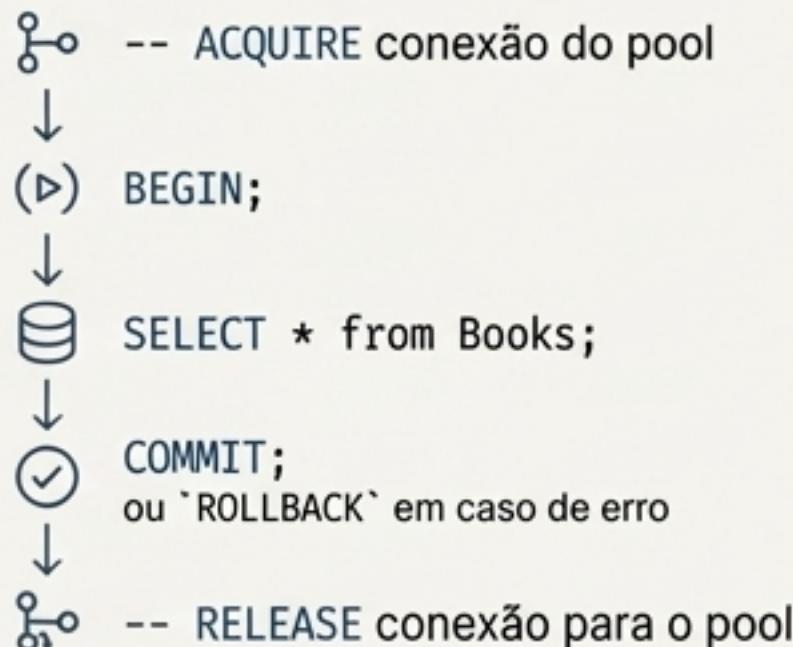
O CAP gerencia automaticamente as transações de banco de dados (ACID) para você. Cada requisição de serviço de alto nível é envolvida em uma transação.

## Transações Automáticas

### Seu Código

```
await db.read('Books')
```

### O que o CAP Faz



## Bloco de Transação (Recomendado)

O CAP gerencia `commit`/`rollback` automaticamente.

```
await cds.tx(async (tx) => {  
  // 'tx' é um objeto de serviço transacional  
  const author = await tx.create(Authors, { name: 'Emily Brontë' });  
  await tx.create(Books, { title: 'Wuthering Heights', author });  
}); // Commit ou rollback automático aqui
```

## Transações Aninhadas

Chamadas para outros serviços dentro de um handler de evento participam da transação raiz. Elas são commitadas ou revertidas atomicamente com a transação principal.

Uma ação `transfer` que atualiza duas contas e grava um log. Todas as três operações (`update`, `update`, `insert`) acontecem na mesma transação lógica.

## Controle Manual com `cds.tx()`

Para cenários que exigem controle explícito (por exemplo, fora de um handler de serviço ou para agrupar múltiplas operações), use `cds.tx()`.

### Bloco de Transação (Recomendado)

O CAP gerencia `commit`/`rollback` automaticamente.

```
const tx = db.tx(), {  
  await tx.run(...);  
  return run(...);  
};
```

### Gerenciamento Totalmente Manual

```
const tx = db.tx();  
try {  
  await tx.run(...);  
  await tx.commit();  
} catch (e) {  
  await tx.rollback(e);  
}
```

# O Contexto da Execução: `cds.context`

Para gerenciar transações, isolamento de tenants e propagação de principal, o CAP precisa de acesso ao contexto da invocação. `cds.context` fornece acesso a essas informações em qualquer lugar do seu código.



## O que há no Contexto?

`cds.context` é uma instância de `cds.EventContext` que contém:

- `user`: O usuário atual, instância de `cds.User`.
- `tenant`: O ID do tenant atual em um cenário multitenant.
- `locale`: O locale do usuário.
- `id`: Um ID de correlação único para a requisição.
- `timestamp`: Um timestamp constante para a requisição atual.
- `http`: Acesso aos objetos `req` e `res` do Express, se aplicável.

## Como Funciona: Continuation-Local Storage

`cds.context` utiliza o `AsyncLocalStorage` do Node.js. Isso significa que o contexto é vinculado à 'continuação' assíncrona da requisição, funcionando como uma 'thread-local variable' para o mundo assíncrono.

## Acesso ao Contexto

### Dentro de Handlers (Preferencial)

Acesse através do objeto da requisição (`req`).

```
this.on('READ', 'Books', req => {
  let { tenant, user } = req; // req é uma instância de EventContext
  // ...
});
```

### Fora de Handlers

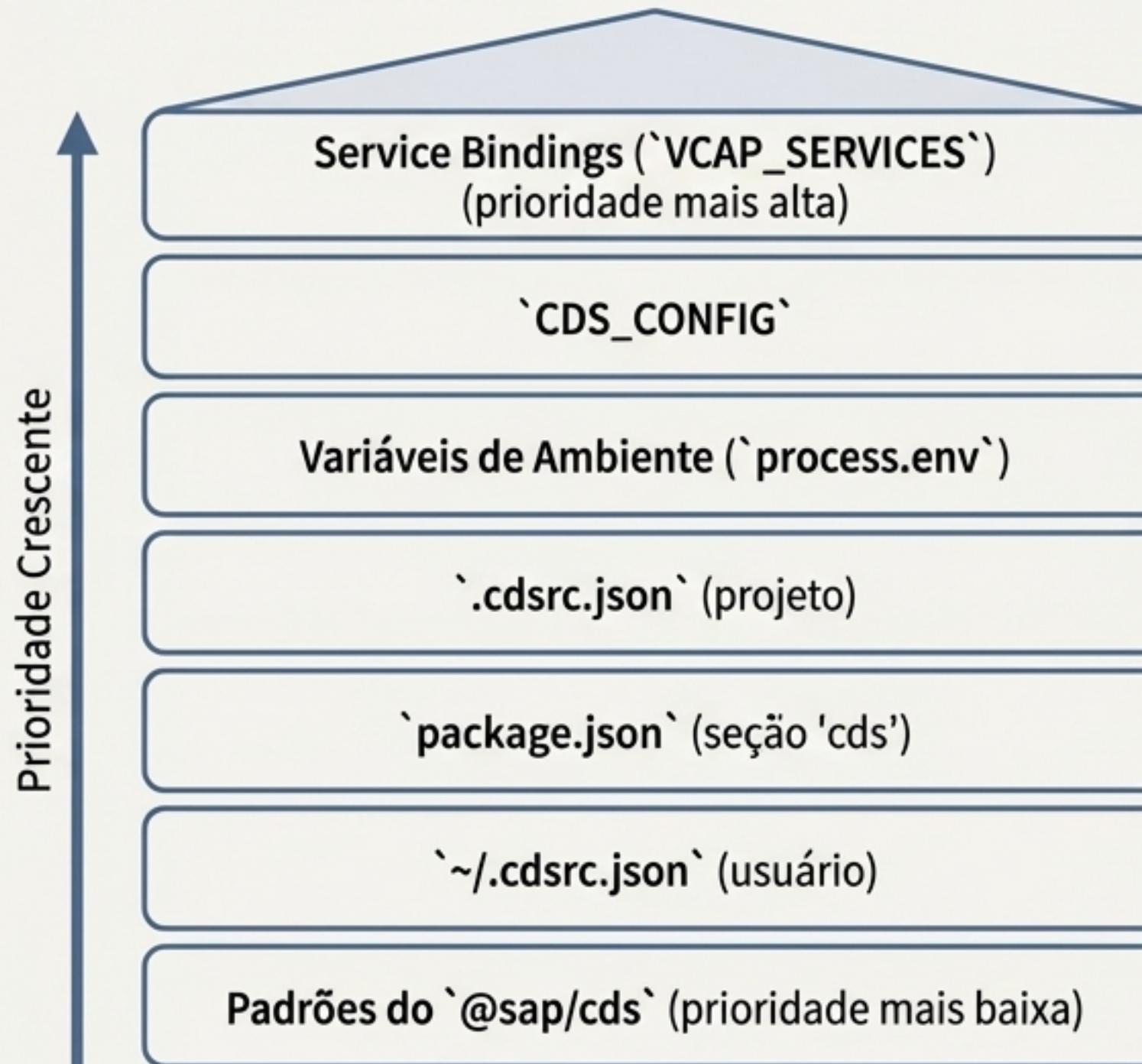
Acesse diretamente via `cds.context`.

```
// Em qualquer módulo da sua aplicação
const { tenant, user } = cds.context;
if (user.is('admin')) { /* ... */ }
```

**Dica:** Prefira usar `req` dentro de handlers, pois o acesso direto a `cds.context` tem um pequeno *overhead*.

# Configuração Unificada e Orientada a Perfis com `cds.env`

`cds.env` fornece acesso transparente à configuração efetiva da sua aplicação, consolidada a partir de diversas fontes.



## Acesso Programático

Use `cds.env` para ler configurações em seu código.

```
const cds = require('@sap/cds');
const dbConfig = cds.env.requires.db;
console.log(dbConfig.kind); //> 'sqlite' ou 'hana'
```

## Perfis de Configuração (`Profiles`)

Defina configurações específicas para diferentes ambientes (ex: `development`, `production`) dentro do seu `package.json`.

```
"cds": {
  "requires": {
    "db": {
      "[development)": {
        "kind": "sqlite",
        "credentials": { "url": "db.sqlite" }
      },
      "[production)": {
        "kind": "hana"
      }
    }
  }
}
```

Nota: O perfil é determinado automaticamente por `NODE\_ENV`, pelo argumento `--profile` ou pela variável `CDS\_ENV`. Se nada for especificado, o perfil `development` é ativado.

# Logging e Observabilidade com `cds.log`

O CAP fornece uma fachada de logging minimalista (`cds.log()`) que se integra facilmente com ferramentas padrão e frameworks avançados como o `winston`.

## Seção 1: Uso Básico e Configuração

### Uso Básico

O uso é semelhante ao `console`, mas com níveis e namespaces.

```
const LOG = cds.log('sql'); // Obtém um logger para o módulo 'sql'  
LOG.info('Executando query...');  
  
if (LOG._debug) { // Verifique o nível para evitar sobrecarga  
  LOG.debug('Detalhes da query:', query);  
}
```

Níveis de Log: `ERROR`, `WARN`, `INFO`, `DEBUG`, `TRACE`.

### Configuração (`package.json`)

Controle os níveis de log por módulo.

```
"cds": {  
  "log": {  
    "levels": {  
      "sql": "debug",  
      "cds": "info"  
    }  
  }  
}
```

## Seção 2: Desenvolvimento vs. Produção

O formato do log se adapta automaticamente ao ambiente.

### Em Desenvolvimento

Formato: `plain` (legível por humanos).

```
[sql] - SELECT * from Books;  
[cds] - ERROR: I will provoke a test error to be thrown.  
  at ODataRequest.reject (/path/to/node_modules/@sap/cds/lib/req/...)  
  at TravelService.<anonymous> (/path/to/srv/travel-service.js:14:9)  
...
```

<IMAGE 0>

### Em Produção (ex: SAP BTP)

Formato: `json` (automático).



Pronto para ingestão por serviços como SAP Cloud Logging ou SAP Application Logging, permitindo busca e análise avançada em dashboards (Kibana).

# Resumo da Jornada: Foco na Lógica, Poder no Framework

Percorremos a jornada completa de desenvolvimento de uma aplicação com o CAP Node.js SDK, desde a concepção do modelo de dados até sua operação robusta em produção.



## Modelar & Compilar (cds.compile)

A definição declarativa é a sua única fonte da verdade, gerando artefatos para persistidos para persistência e serviços.



## Implementar (cds.Service)

A lógica de negócios é adicionada de forma limpa e reativa através de handlers de eventos (before, on, after).



## Interagir (cds.ql)

A comunicação com outros serviços é unificada através de uma poderosa API de consulta agnóstica de plataforma.



## Operar (cds.context, cds.tx)

O framework gerencia automaticamente a complexidade de transações, multitenancy e configuração, garantindo a robustez da aplicação.

**O CAP Service SDK não é apenas uma biblioteca, é um ecossistema projetado para produtividade. Ao internalizar os padrões comuns de aplicações empresariais, ele libera o desenvolvedor para focar exclusivamente na criação de valor para o negócio. O resultado é um código mais limpo, mais consistente e mais rápido de entregar.**