

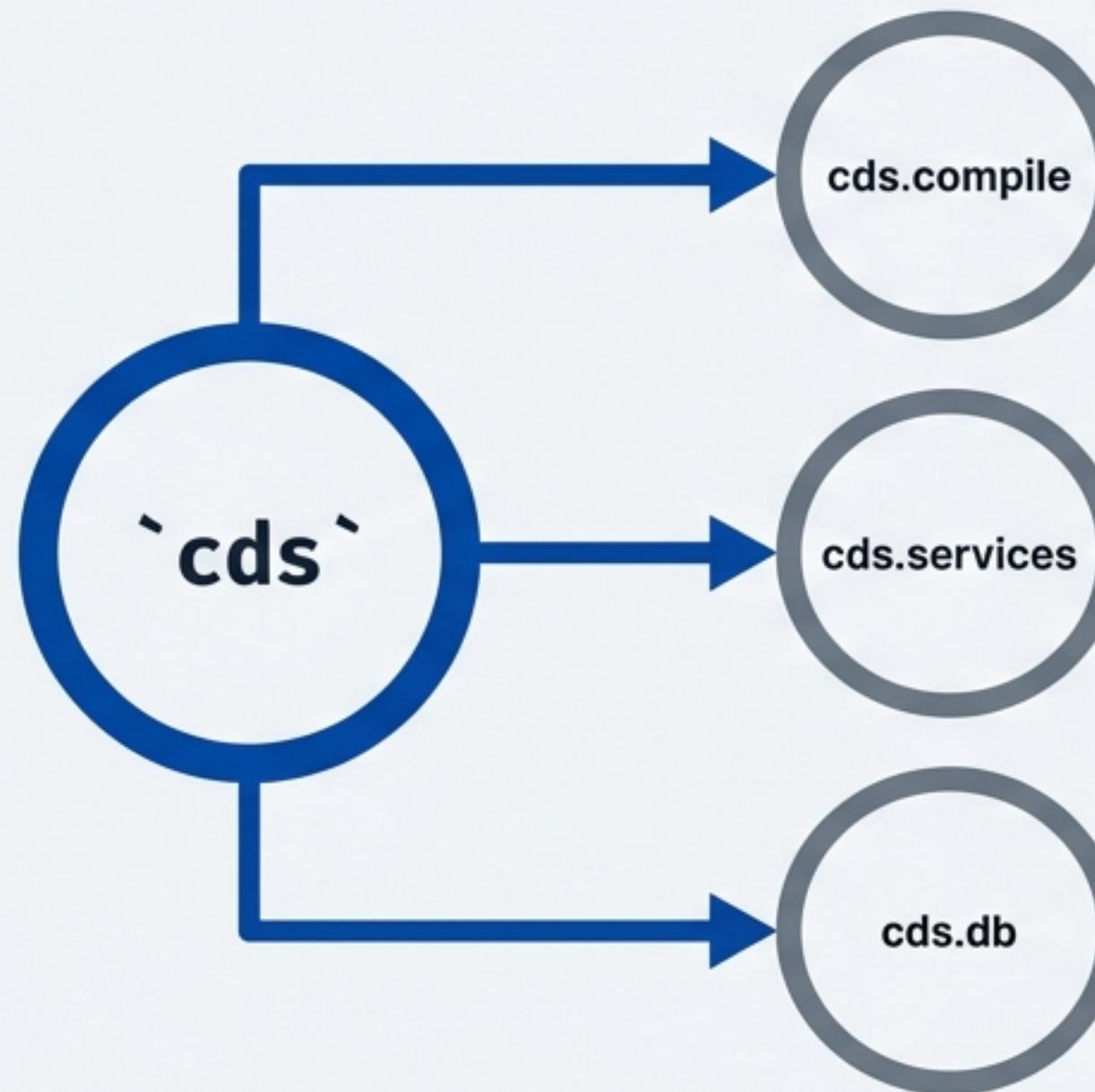


# Desvendando o Objeto `cds`

O Ponto de Acesso Central para Todas as APIs do CAP Node.js

# Um Único Ponto de Entrada. Inúmeras Possibilidades.

O objeto `cds` é a 'fachada' que unifica o acesso a todo o ecossistema CAP. Em vez de importar de múltiplos caminhos internos, que são sujeitos a **mudanças**, você **começa aqui**. Isso garante que seu **código seja mais robusto e fácil de manter**.



✓

```
const { Request } =  
require('@sap/cds')
```

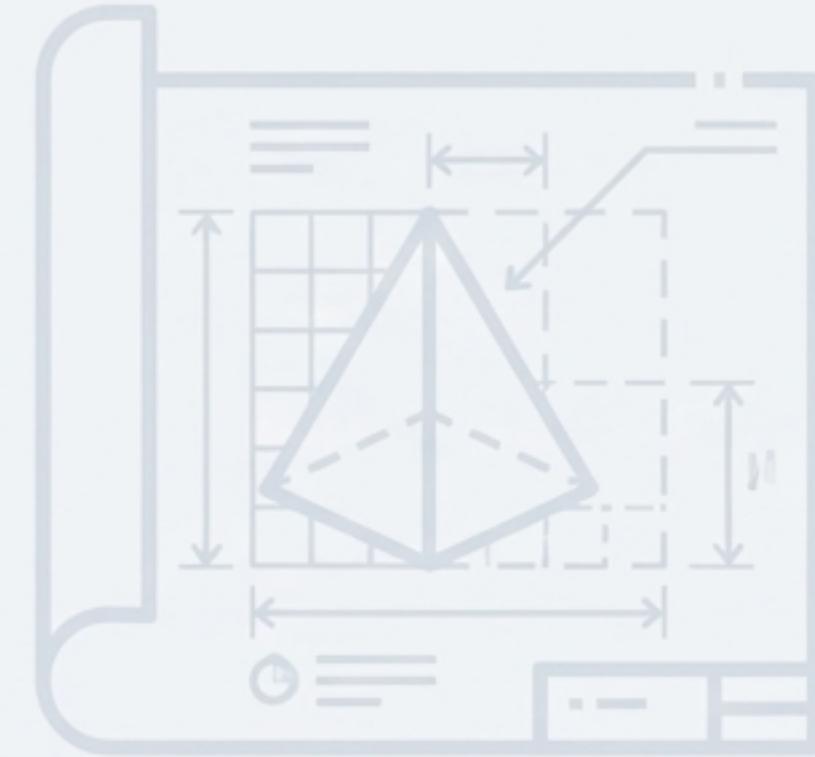
✗

```
const Request =  
require('@sap/  
cds/lib/..lib/...  
/Request')
```

Nunca crie código que dependa de caminhos internos de `@sap/cds`.

# Modelagem e Compilação

APIs para carregar, analisar, compilar e inspecionar suas definições de modelo (CDL/CSN).



## - **cds.compile()**

Transforma CDL em CSN.

```
let csn = cds.compile('entity Foo { key ID: UUID }')
```

## - **cds.parse()**

Analisa CDL para uma representação abstrata (CST).

## - **cds.linked()**

Acesso ao modelo com todas as associações resolvidas.

## - **cds.model**

Acesso ao modelo globalmente carregado após o bootstrap.

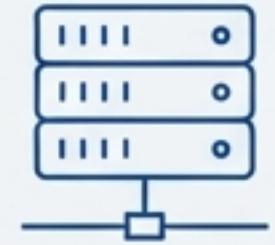
```
cds.model = await cds.load('*')
```

## - **cds.entities**

Atalho conveniente para `cds.model.entities`. Permite acessar as entidades do modelo de forma direta.

# Servidor e Serviços

APIs para inicializar o servidor, interagir com os serviços da aplicação e conectar-se a fontes de dados ou sistemas remotos.



- `cds.serve()`: O ponto de partida para executar seus serviços e iniciar o servidor.
- `cds.server`: Fornece a referência à instância do servidor subjacente (geralmente uma aplicação `express.js`).
- `cds.connect()`: Conecta-se a um serviço, especialmente o banco de dados.

```
cds.db = await cds.connect.to('db')
```

- `cds.services`: Um registro (dicionário) de todos os serviços instanciados. É um objeto iterável.

```
let { CatalogService, db } = cds.services;
for (let s of cds.services) { /* ... */ }
```

# Acesso a Dados e Transações



Apresenta a interface fluente e integrada para interagir com o banco de dados primário da aplicação de forma segura e consistente.

- **cds.db**  
Atalho direto para o serviço de banco de dados primário (cds.services.db).
- **cds.ql (SELECT, INSERT, UPDATE, DELETE)**  
Apresenta a API de consulta fluente.

```
let books = await SELECT.from(Books);
```

Explicação: Este código é um atalho para await cds.db.run(SELECT.from(Books)), demonstrando como as operações QL usam cds.db por padrão.

- **cds.tx()**  
Introduz o conceito de gerenciamento de transações, permitindo que múltiplas operações sejam executadas atomicamente.



# Configuração e Ambiente

**Conceito:** Como o CAP consolida configurações de diversas fontes (package.json, .cdsrc.json, variáveis de ambiente, service bindings) e as expõe de forma unificada.

## cds.env

O objeto que contém a configuração efetiva e consolidada.

```
cds.env.requires.auth
```

## cds.requires

Um "overlay" e atalho para cds.env.requires. A principal vantagem é que ele mapeia nomes de definições de serviço para suas configurações, mesmo que tenham nomes diferentes na configuração.

## Exemplo de Cenário

Definição do serviço: service ReviewsService {}

Configuração em json: { 'reviews': { 'service': 'ReviewsService' } }

### Comparação de acesso

#### cds.env

```
cds.env.requires.ReviewsService // > undefined
```

#### cds.requires

```
cds.requires.ReviewsService // > aponta para a config de 'reviews'
```

# Classes e Tipos Essenciais

Apresenta as classes fundamentais que representam os principais conceitos do CAP, disponíveis diretamente no objeto `cds` para facilitar o uso.



Conceitos de Serviço	Estruturas de Requisição	Conceitos de Modelo	Contexto de Usuário
<ul style="list-style-type: none"><li>• cds.Service</li><li>• cds.ApplicationService</li><li>• cds.RemoteService</li><li>• cds.DatabaseService</li><li>• cds.MessagingService</li></ul>	<ul style="list-style-type: none"><li>• cds.Request</li><li>• cds.Event</li><li>• cds.EventContext</li></ul>	<ul style="list-style-type: none"><li>• cds.Association</li><li>• cds.Composition</li></ul>	<ul style="list-style-type: none"><li>• cds.User</li></ul>

# Utilitários e Metadados



Propriedades úteis para obter informações sobre o ambiente de execução, versão do framework e estrutura do projeto.

**cds.version:** Retorna a versão do pacote @sap/cds.

```
const [major] = cds.version.split('.').map(Number);
if (major < 6) { /* código legado */ }
```

**cds.root:** O diretório raiz do projeto (o padrão é `process.cwd()`).

**cds.home:** O diretório de instalação do @sap/cds.

**cds.cli:** Fornece acesso aos argumentos da linha de comando com os quais o processo foi iniciado. Útil para plugins.  
Exemplo de estrutura:

```
{
  "command": "serve",
  "argv": [...],
  "options": { "with-mocks": true }
}
```

# Gerenciamento de Erros de Forma Elegante



O método `cds.error` é uma ferramenta flexível para construir e lançar erros com mensagens, status HTTP e códigos customizados.

## Instanciando vs. Lançando Immediatamente

```
// Cria o objeto de erro.  
let e = new cds.error('message', { code, ... });  
  
// Lança o erro se foo for falso.  
foo || cds.error('Expected foo to be truthy');
```

## Uso com Tagged Template Strings

(Permite interpolação de objetos com formatação rica via `util.format()`)

```
foo || cds.errorExpected 'foo' to be truthy, but  
got: ${foo};
```

## Atalho `cds.error.expected`

(Uma forma mais conveniente para construir mensagens de erro comuns)

```
foo || cds.error.expected${foo}} to be truthy;
```

# O Ciclo de Vida da Aplicação e o `cds.on`

O objeto `cds` é um `EventEmitter`. Você pode registrar *listeners* para eventos chave do ciclo de vida da aplicação, permitindo a execução de lógica customizada em momentos específicos, como inicialização e desligamento.



- 'bootstrap': Emitido no início da inicialização.
- 'served': Emitido quando os serviços estão montados e prontos.
- 'listening': Emitido quando o servidor está online e aceitando requisições.
- 'shutdown': Emitido para permitir um desligamento gracioso (*graceful shutdown*). Os handlers podem ser assíncronos.

## \*\*Exemplo Prático\*\*

```
// Garante a limpeza de recursos antes de sair
cds.on('shutdown', async () => {
  await fs.promises.rm('some-file.json');
  console.log('shutdown');
});
```

`cds.exit()` inicia o processo de *graceful shutdown*, executando os *handlers* de `shutdown` antes de finalizar o processo.

# A Ferramenta Essencial: `cds repl`

O `cds repl` (Read-Eval-Print Loop) é seu playground interativo para o CAP. É a maneira mais rápida e eficaz de explorar o objeto `cds`, inspecionar modelos compilados, testar consultas e entender o comportamento das APIs em tempo real.

```
> cds.version
'7.3.0'

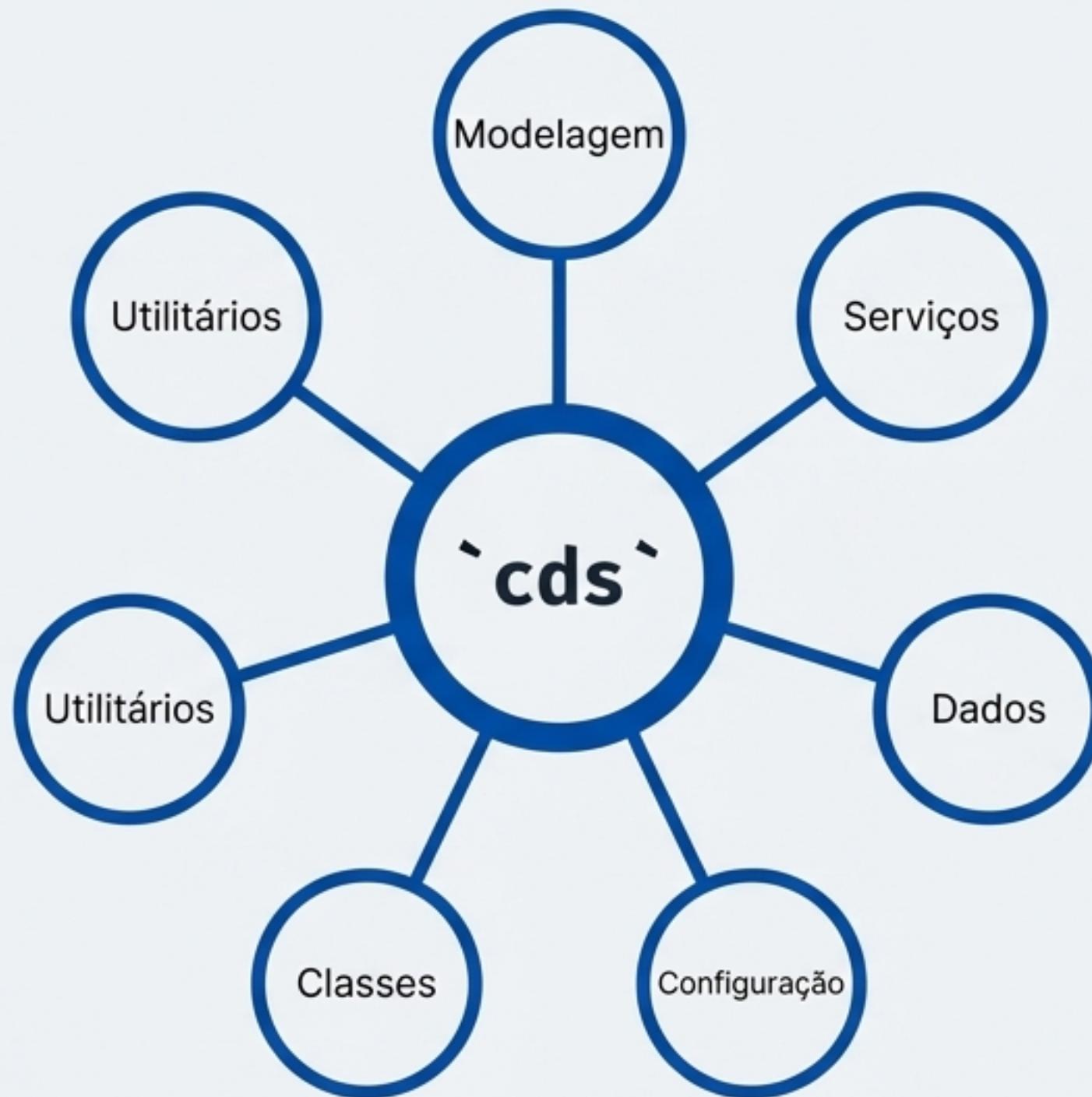
> cds.model.entities.Books
{ kind: 'entity', ... }

> cds.compile('entity Foo {}')
{ definitions: { Foo: { kind: 'entity', ... } } }

> cds.requires.db
{ kind: 'sqlite', ... }

> █
```

# Dominando o Objeto `cds`



## Pontos-Chave para Memorizar

- `cds` é sua **fachada única** para todas as APIs do CAP.
- **Sempre** importe via `const cds = require('@sap/cds')`. Evite caminhos internos.
- Pense em termos de **domínios funcionais**: Modelagem, Serviços, Dados, Configuração, Classes e Utilitários.
- Utilize `cds.ql` e `cds.db` para um acesso a dados fluente e integrado.
- Explore interativamente e valide suas ideias rapidamente com o `cds repl`.

# Próximos Passos e Recursos

O objeto `cds` é o seu ponto de partida. Aprofunde seu conhecimento explorando a documentação oficial e aplicando esses conceitos em seus projetos.

## Documentação Oficial (capire)

A fonte de verdade abrangente para todos os aspectos do CAP.



<https://cap.cloud.sap/docs/>

## API Reference

Documentação detalhada das APIs do Node.js para o objeto `cds`.

</> <https://cap.cloud.sap/docs/node.js/api>