

## De Blaupausen a Modelos Vivos: Dominando a Reflexão de Modelos com 'cds.linked'

Um modelo CDS (CSN) é a espinha dorsal de qualquer aplicação CAP. Por padrão, é uma estrutura de dados estática – um JSON detalhado, porém inerte. Navegar e interpretar essa estrutura diretamente no código pode ser complexo e propenso a erros. E se pudéssemos “dar vida” a esse modelo? Transformá-lo em um grafo de objetos rico, navegável e inteligente por ser complexo e propenso a erros. E se pudéssemos “dar vida” a esse modelo? Transformá-lo em um grafo de objetos rico, navegável e inteligente em tempo de execução?

**Essa é a função da API de Reflexão: ‘cds.linked’.**

# O Ponto de Partida: `cds.linked(csn)`

## O Conceito

O método `cds.linked` (ou seu alias `cds.reflect`) é a porta de entrada para a API de reflexão. Ele transforma um modelo CSN em uma instância da classe `LinkedCSN` e todas as suas definições em instâncias de `LinkedDefinition`, de forma recursiva.

## \*\*Características Chave\*\*

- **Modificação In-Place:** O método modifica o objeto CSN passado como argumento. O modelo retornado é o mesmo objeto.
- **Idempotente:** Chamar `cds.linked` em um modelo que já foi 'linkado' não tem custo adicional de processamento.

## O Código em Ação

```
function* cds.linked(csn: CSN | string)
=> LinkedCSN

// 1. Carregue o modelo
let csn = cds.load('some-model.cds');

// 2. Crie o modelo "linkado"
let linked = cds.linked(csn);

// linked === csn (retorna true)
```

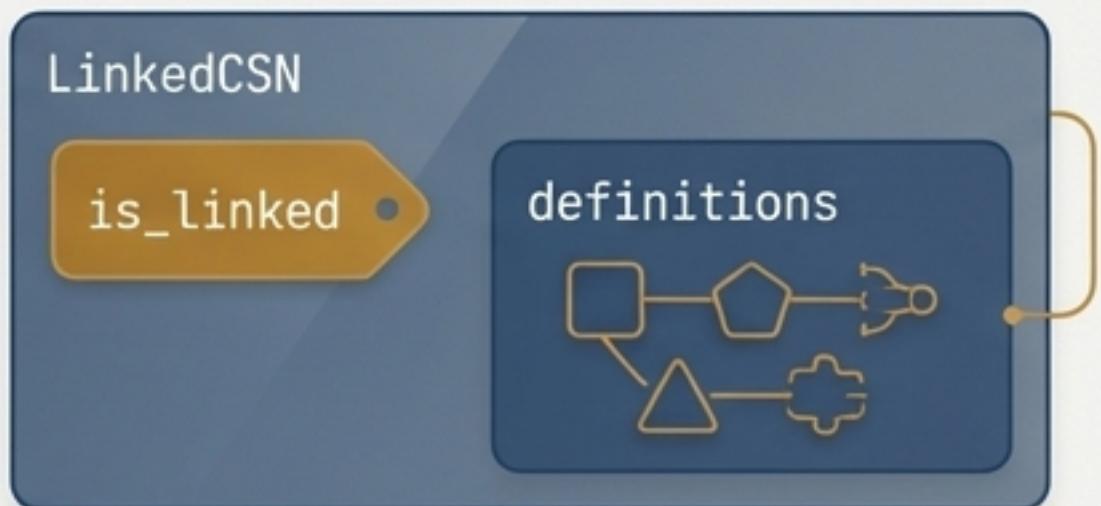
# Explorando o Mundo: A Classe `LinkedCSN`

## O Conceito

O objeto retornado por `cds.linked` é uma instância de `LinkedCSN`. Ele representa todo o seu modelo de forma navegável.

### \*\*Propriedades Fundamentais\*\*

- `.is_linked`: Uma propriedade de tag que é `true` para modelos processados pela API. Uma forma rápida de verificar o estado do modelo.
- `.definitions`: O coração do modelo. Um objeto contendo todas as definições (entidades, serviços, tipos, etc.) transformadas em instâncias de `LinkedDefinitions`.



## O Código em Ação

```
let linked = cds.linked
entity Books {
    key ID: UUID;
    title: String;
    author: Association to Authors;
}
entity Authors {
    key ID: UUID;
    name: String;
}
;

// Acessando todas as definições
for (const def of linked.definitions) {
    console.log(def.kind, def.name);
}

> entity my.bookshop.Books
> entity my.bookshop.Authors
```

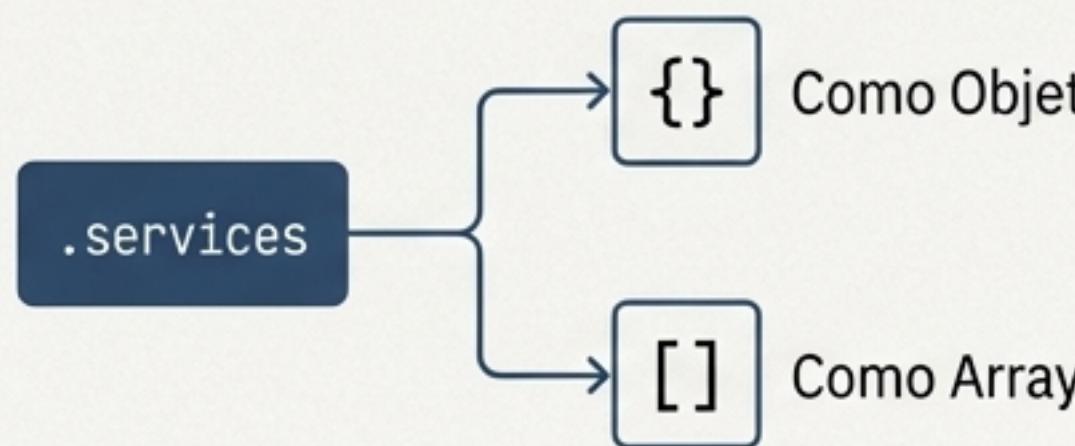
# Atalhos Inteligentes para Serviços e Entidades

## O Conceito

Para facilitar o acesso, `LinkedCSN` oferece atalhos para as coleções mais comuns. Essas propriedades (`.services`, `.entities`) têm uma natureza dupla:

- **Como Objeto:** Permitem acesso direto às definições pelo nome, ideal para desestruturação.
- **Como Array:** Permitem iteração padrão com `for...of`.

A propriedade ` `.entities` também atua como uma função para buscar entidades dentro de um namespace específico.



## O Código em Ação

Acesso a ` `.services`

```
let m = cds.linked`service S1 { ... } service S2 { ... }`;  
// Natureza de Objeto  
let { S1, S2 } = m.services;  
  
// Natureza de Array  
for (let s of m.services) { console.log(s.name); } // S1, S2
```

Acesso a ` `.entities`

```
let m = cds.linked`namespace my.app; entity E1{} entity E2{}`;  
// Natureza de Função (com namespace)  
let { E1, E2 } = m.entities('my.app');  
  
// Natureza de Objeto (namespace padrão)  
let { E1, E2 } = m.entities;
```

# Consultando o Modelo: `each`, `find` e `foreach`

## O Conceito

A API fornece métodos poderosos para consultar as definições do modelo.

- `m.each('kind')`: Retorna um *iterador* para todas as definições que correspondem ao filtro (por exemplo, `'entity'`, `'service'`). Suporta tipos derivados (ex: `'struct'` encontra `structs` e `entities`).
- `m.find('kind')`: Um atalho conveniente que retorna a *primeira* definição encontrada que corresponde ao filtro. Equivalente a `m.each(...)` e pegar o primeiro resultado.
- `m.foreach('kind', visitor)`: Executa uma função `visitor` para *cada* definição que corresponde ao filtro. Itera apenas no nível passado; não desce na hierarquia de elementos.

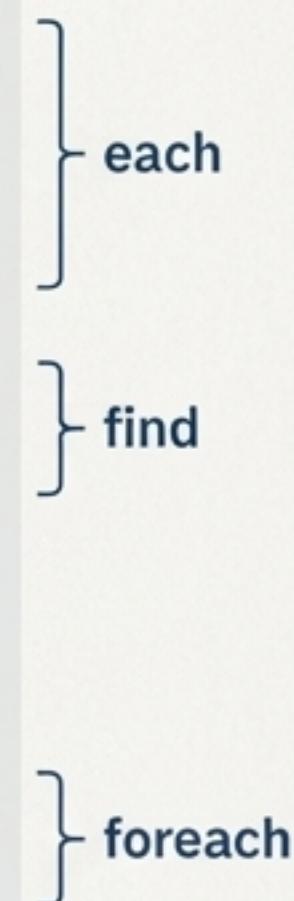
## O Código em Ação

```
let m = cds.linked(csn);

// Iterar sobre todas as entidades usando each()
for (let entity of m.each('entity')) {
    console.log('Found entity:', entity.name);
}

// Encontrar o primeiro serviço usando find()
let firstService = m.find('service');

// Imprimir o nome de todas as associações na
// entidade Books
let { Books } = m.entities;
m.foreach('Association', a =>
    console.log(a.name), Books.elements);
```



The diagram consists of three large curly braces on the right side of the code. The top brace groups the entire first code block under the label 'each'. The middle brace groups the second code block under the label 'find'. The bottom brace groups the third code block under the label 'foreach'.

# O Padrão Universal: A Classe `LinkedDefinitions`

## O Conceito

`LinkedDefinitions` é a classe base para **qualquer coleção** de definições em um modelo linkado.

Isso significa que o padrão de acesso duplo (objeto e array) que vimos em `m.entities` se aplica a muitos outros lugares:

- cds.service.entities
- cds.entity.elements
- cds.entity.keys
- cds.entity.associations
- cds.Association.foreignKeys

**Princípio de Design:** Consistência. Uma vez que você aprende a interagir com uma coleção, você sabe interagir com todas.

## O Código em Ação

```
let linked = cds.linked(model);  
  
// Acessando entidades do modelo (instância de  
// LinkedDefinitions)  
let { Books, Authors } = linked.entities; //  
// Acesso como objeto  
let [ bookDef, authorDef ] = linked.entities;  
// Acesso como array  
  
// Acessando elementos da entidade Books  
// (também LinkedDefinitions)  
let { ID, title, author } = Books.elements;  
// Acesso como objeto  
  
// Iterando sobre os elementos  
for (let element of Books.elements) {  
    console.log(element.name, ':', element.type);  
}
```

# O Bloco de Construção Fundamental: `LinkedDefinition`

## O Conceito

Cada entrada em uma `LinkedDefinitions` é uma `LinkedDefinition` (ou uma subclasse dela). É a classe base para tudo no modelo.

### \*\*Propriedades Essenciais\*\*

- `.name`: O nome totalmente qualificado da definição (ex: `my.bookshop.Books`).
- `.kind`: O tipo resolvido da definição (ex: `'entity'`, `'service'`, `'element'`).
- `.is_linked`: Propriedade de tag, sempre `true`.

Como todas as definições são instâncias de classes específicas, você pode usar o operador `instanceof` do JavaScript para checagens de tipo seguras e legíveis.

## O Código em Ação

```
let { Books } = cds.linked(csn).entities;  
  
console.log(Books.name);  
console.log(Books.kind);  
  
// Checagem de tipo em tempo de execução  
if (Books instanceof cds.entity) {  
    console.log("É uma entidade!");  
}  
  
if (Books.elements.author instanceof cds.Association) {  
    console.log("O autor é uma associação!");  
}  
  
> my.bookshop.Books  
> 'entity'  
> É uma entidade!  
> O autor é uma associação!
```

# Detalhando um `cds.service`

## O Conceito

Definições de serviço no modelo linkado são instâncias de `cds.service`. Esta classe fornece atalhos convenientes para acessar as definições expostas pelo serviço.

### **\*\*Atalhos da Classe\*\***

- `.entities`: Acesso a todas as entidades expostas.
- `.events`: Acesso aos eventos.
- `.actions`: Acesso às ações e funções.

Cada um desses atalhos retorna uma instância de `LinkedDefinitions`, permitindo o mesmo padrão de acesso duplo que já conhecemos.

## O Código em Ação

```
let m = cds.linked`  
  service CatalogService {  
    entity Products { key ID: UUID; name: String; }  
    entity Categories { key ID: UUID; descr: String; }  
    action updateStock(product: UUID, quantity: Integer);  
  }  
`;  
  
let { CatalogService } = m.services;  
  
// Acessando as entidades do serviço  
let { Products, Categories } = CatalogService.entities;  
console.log(Products.name); //> CatalogService.Products  
  
// Acessando as ações do serviço  
let { updateStock } = CatalogService.actions;  
console.log(updateStock.kind); //> 'action'
```

# A Anatomia de uma `cds.entity`

## O Conceito

`cds.entity` é uma subclasse de `cds.struct` e representa as entidades do seu domínio. A API de reflexão oferece acesso direto e estruturado aos componentes de uma entidade.

### **\*\*Propriedades Chave\*\***

- **.elements**: A coleção completa de todos os elementos declarados.
- **.keys**: Um atalho para os elementos que são parte da chave primária.
- **.associations**: Atalho para todos os elementos do tipo `Association`.
- **.compositions**: Atalho para todos os elementos do tipo `Composition`.
- **.texts**: Referência à entidade de textos, se houver elementos localizados.
- **.drafts**: Referência à entidade de rascunho, se o 'draft' estiver habilitado.

## O Código em Ação

```
let m = cds.linked`  
entity Books: cuid, managed {  
    title : localized String;  
    stock : Integer;  
    author : Association to Authors;  
    chapters : Composition of many Chapters;  
}  
// ... outras entidades  
;  
  
let { Books } = m.entities;  
  
// Acessando as chaves (herdadas de 'cuid')  
console.log(Object.keys(Books.keys)); //> [ 'ID' ]  
  
// Acessando associações e composições  
let { author } = Books.associations;  
let { chapters } = Books.compositions;  
  
console.log(author.target); //> 'Authors'  
console.log(chapters.target); //> 'Books.chapters'
```

# Navegando pelas Conexões: `cds.Association`

## O Conceito

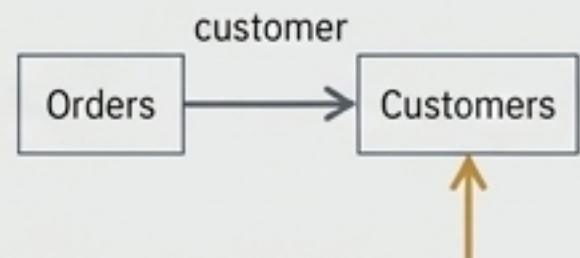
Todas as associações e composições são instâncias de `cds.Association`. A API resolve e enriquece a definição para facilitar a navegação entre entidades.

### \*\*Propriedades de Navegação\*\*

- `._target`: Uma **referência direta** ao objeto da definição do alvo da associação.
- `.isAssociation` / `.isComposition`: Tags booleanas para diferenciar os tipos.
- `.is2one` / `.is2many`: Atalhos booleanos para verificar a cardinalidade.
- `.keys`: As chaves declaradas ou inferidas da associação.
- `.foreignKeys`: As chaves estrangeiras efetivas em associações gerenciadas.

## O Código em Ação

```
let m = cds.linked`  
    entity Orders {  
        key ID: UUID;  
        customer: Association to one Customers;  
        Items: Composition of many OrderItems on Items.order = $self;  
    }  
    entity Customers { key ID: UUID; name: String; }  
    entity OrderItems {  
        key ID: UUID;  
        product: String;  
        order: Association to Orders; }  
`;  
  
let { customer, Items } = m.entities.Orders.elements;  
  
// Verificando cardinalidade  
console.log(customer.is2one); //> true  
console.log(Items.is2many); //> true  
  
// Navegando para o alvo  
let customerEntity = customer._target;  
console.log(customerEntity.name); //> 'Customers'  
console.log(Object.keys(customerEntity.keys)); //> ['ID']
```



# A Arquitetura Interna: `cds.linked.classes`

## O Conceito

A propriedade `cds.linked.classes` expõe o conjunto completo de classes usadas pela API de reflexão. Ela revela a hierarquia de herança, explicando por que `entity instanceof struct` retorna `true`. Compreender essa estrutura permite escrever um código mais robusto e à prova de futuro.

## Uso

```
const { entity, Association } = cds.linked.classes;  
  
if (myDef instanceof entity) { ... }
```

## A Hierarquia de Classes

```
class any  
  └ class context  
    └ cds.service  
  └ class type  
    └ class scalar  
      └ class string  
      └ class number  
      └ ...  
  └ cds.array  
  └ cds.struct  
    └ cds.entity  
    └ cds.event  
  └ cds.Association  
    └ cds.Composition
```

# Estendendo o Padrão: Adicionando Comportamentos com `mixin()`

## O Conceito

A API de reflexão não é uma caixa preta; ela é extensível.

O método `cds.linked.classes.mixin()` permite injetar seus próprios métodos nas classes de reflexão (`type`, `struct`, `entity`, etc.).

Isso abre a porta para a criação de ferramentas poderosas, como geradores de código, validadores personalizados ou serializadores.

## \*\*Caso de Uso\*\*

Vamos implementar um conversor `toCDL()` para transformar definições de volta em sua representação de texto CDL.

## Exemplo: `csn2cdl`

```
cds.linked.classes.mixin(  
    class type {  
        toCDL() { return `${this.kind} ${this.name} :  
            ${this.typeAsCDL()};\n` }  
        typeAsCDL() { return `${this.type.replace(/^cds\./, '')}` }  
    },  
    class struct {  
        typeAsCDL() { return `{\n${Object.values(this.elements).map(  
            e => ` ${e.toCDL()}`)  
        ).join('')}}` }  
    },  
    // ... outras extensões para Association, etc.  
);  
  
// Test drive  
let m = cds.linked`...`;  
m.foreach(d => console.log(d.toCDL()));
```

# Recursão Elegante: `cds.builtin.types`

## O Conceito

A propriedade `cds.builtin.types` dá acesso a todas as definições dos tipos pré-definidos do CAP (`UUID`, `Integer`, `String`, etc.). A beleza deste recurso é que o próprio objeto `cds.builtin.types` é um modelo ‘linkado’ — uma instância de `LinkedDefinitions`. Isso demonstra a consistência e o poder do design da API: a reflexão é usada para definir os próprios fundamentos do framework.



## A Estrutura

A propriedade é construída de forma conceitual como:

```
cds.builtin.types = cds.linked  
  using from './roots';  
  
context cds {  
  type UUID : String(36);  
  type Boolean : boolean;  
  type Integer : number;  
  type Decimal : number;  
  type Date : date;  
  type String : string;  
  // ... e todos os outros tipos primitivos  
}').definitions;
```

Isso significa que você pode inspecionar `cds.String` da mesma forma que inspeciona uma de suas próprias entidades.

# Colocando a Reflexão em Prática: Casos de Uso Avançados

Com um modelo vivo e navegável, você pode ir além do código padrão e escrever lógicas mais inteligentes e dinâmicas.



## Construção Dinâmica de CQN

Itere sobre os `elements` de uma entidade para construir colunas de projeção ou cláusulas `where` dinamicamente com base em metadados.



## Validação de Modelo Genérica

Use `m.each('entity')` para percorrer todas as entidades e aplicar regras de validação consistentes consistentes (ex: toda entidade deve ter um elemento `name`).



## Plugins e Extensões Reutilizáveis

Crie middlewares que inspecionam a entidade alvo (`req.target`) para aplicar lógicas de autorização, logging ou transformação de dados com base em suas anotações ou estrutura.



## Ferramentas de Introspecção

Construa seus próprios utilitários de linha de comando ou endpoints de API que expõem a estrutura do modelo em tempo de execução para fins de depuração ou documentação.

# Seu Modelo, Potencializado.

A API `cds.linked` transforma o CSN de um **blaupause** estático para um **modelo vivo e acionável**.

## O que você ganha:

- **Navegação Intuitiva:** Um grafo de objetos rico com atalhos convenientes.
- **Segurança de Tipos:** Verificação em tempo de execução com `instanceof` para um código mais robusto.



- **Introspecção Completa:** Acesso programático a cada detalhe do seu modelo, de chaves a cardinalidades.
- **Extensibilidade Total:** A capacidade de adicionar seus próprios comportamentos ao modelo com `mixin()`.

Use a API de reflexão para escrever aplicações CAP mais dinâmicas, genéricas e inteligentes.