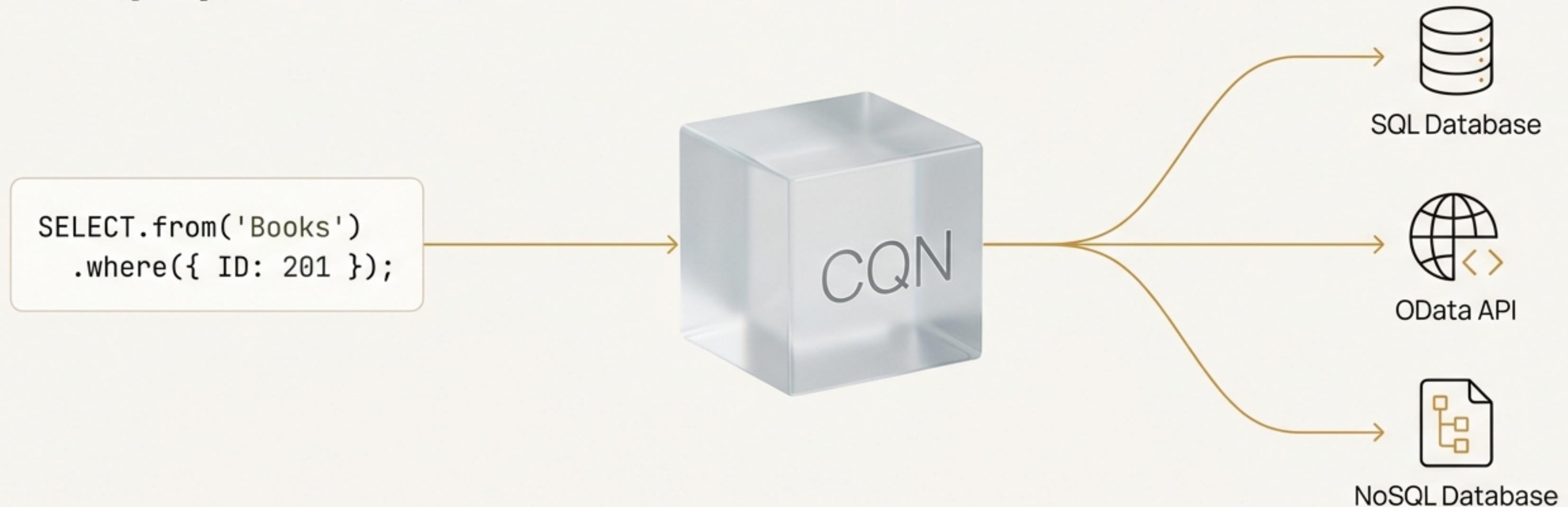


Dominando Queries com CAP Node.js

Uma Jornada Estratégica pelo `cds.ql`

Não é SQL. É uma forma universal de consultar serviços.

O `cds.ql` não gera SQL diretamente. Ele constrói objetos em **CQN (Core Query Notation)**, uma linguagem de consulta abstrata. Isso garante que suas queries possam ser executadas em qualquer tipo de serviço, seja um banco de dados SQL, NoSQL ou uma API OData remota. A sintaxe pode parecer familiar, mas a filosofia é muito mais flexível.



"While both CQL / CQN as well as the fluent API of cds.ql resemble well-known SQL syntax neither of them are locked in to SQL. In fact, queries can be sent to any kind of services..."

Os 5 Pilares para a Maestria em `cds.ql`

Para dominar `cds.ql`, precisamos entender cinco princípios fundamentais que guiam seu design e uso.



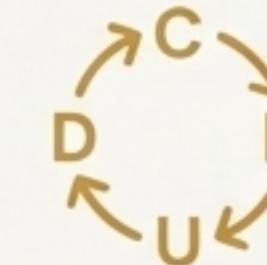
Construção Flexível: Escolha o estilo de codificação que melhor se adapta à sua necessidade.



Segurança por Padrão: Proteção nativa contra as vulnerabilidades mais comuns.



Execução Explícita: Desacople a construção da execução para máxima flexibilidade.



Maestria em CRUD: Padrões consistentes para as operações essenciais de dados.



O Poder dos Objetos: Trate queries como valores para criar lógicas complexas e performáticas.

API Fluente vs. Tagged Template Literals

Clareza e Autocompletar

```
// Estilo limpo e ideal para  
construção dinâmica  
let q = SELECT.from('Books').where(  
{ID:201}).orderBy({title:1});
```

Poder Expressivo

```
// Sintaxe próxima ao CQL,  
excelente para queries complexas  
let q = SELECT `from Books where  
ID=${201} order by title`;
```

Use a API Fluente para clareza e construção programática. Prefira TTLs para queries complexas com uma sintaxe mais próxima ao CQL.



A Melhor Prática: Definições Refletidas

A forma mais robusta de construir queries é usando as definições de entidades refletidas do seu modelo de serviço. Isso elimina a repetição de namespaces, aumenta a segurança de tipos e torna o código mais limpo.

```
const { Books } = cds.entities;  
  
// Código mais seguro e sem strings mágicas  
let q1 = SELECT.from(Books).where`ID=${201}`;
```

“It is recommended best practice to use entity definitions reflected from a service’s model to construct queries. Doing so simplifies code as it avoids repeating namespaces all over the place.”



Pilar 2: Execução Explícita

Construa uma vez, execute em qualquer lugar.

Construir uma query com `SELECT` ou `INSERT` não a executa. Você tem controle total sobre *quando e onde* ela será executada.

No Banco de Dados Padrão (`cds.db`)

```
// Simplesmente usar 'await' executa no banco  
de dados primário  
let books = await SELECT.from(Books).where({ID:  
201});
```

Em um Serviço Específico

```
// Conecte-se a qualquer serviço (local ou  
remoto) e use .run()  
const cats = await  
cds.connect.to('CatalogService');  
let query = SELECT.from(Books).where({ID: 201});  
let books = await cats.run(query);
```

Essa separação permite enviar a mesma query para o banco de dados principal, um serviço OData remoto ou qualquer outro serviço conectado ao CAP.



Pilar 3: O Poder dos Objetos

Queries não são comandos, são valores.

Em `cds.ql`, uma query é um objeto. Assim como uma função, ela pode ser armazenada em uma variável, passada como argumento e retornada de outra função. Isso permite a criação de código modular e reutilizável.

```
// A query é construída e armazenada em uma variável
let PoesBooks = SELECT.from(Books).where`author.name like '%Poe%'`;

// A variável pode ser usada posteriormente em diferentes contextos
const catalogService = await cds.connect.to('CatalogService');
let books = await catalogService.get(PoesBooks);
```

Trate **queries como blocos de construção**: construa-as, **componha-as** e **reutilize-as** para criar lógicas de dados complexas.

“...queries are first-class objects, which can be assigned to variables, modified, passed as arguments, or returned from functions.”

Pilar 3: O Poder dos Objetos

Vantagem Prática: *Late Materialization*

Faça Assim (Performático)

Deixe o banco de dados trabalhar

Componha queries usando sub-selects. O banco de dados otimiza a execução, evitando a transferência de grandes volumes de dados intermediários para a aplicação.

```
let Authors = SELECT`ID`.from`Authors`.where`name like
${input}`;
let Books = SELECT.from`Books`.where`author_ID `;
let Books = SELECT.from`Books`.where`author_ID in ${Authors}`;
// Os IDs dos autores nunca são materializados no Node.js
await cds.run(Books);
```

Não Faça Assim (Lento)

Evite materialização imperativa

Buscar IDs primeiro para depois usá-los em um loop força a "materialização" dos dados na aplicação, gerando múltiplas idas ao banco e maior consumo de memória.

```
let Authors = await
let Authors = await SELECT`ID`.from`Authors`.where`name like
${input}`;

for (let a of Authors) { // Loop sobre dados já materializados
  let Books = await SELECT.from`Books`.where`author_ID =
${a.ID}`;
}
```



Pilar 4: Segurança por Padrão

Proteção nativa contra SQL Injection

✓ Seguro: Binding Parameters

`cds.ql`, especialmente com Tagged Template Literals, usa *binding parameters* automaticamente. O input do usuário é tratado como valor, nunca como código executável.

```
let input = "201"; // Input do usuário
let books = await SELECT.from`Books`.where`ID =
${input}`;

// CQN gerado: { ref:['ID'], '=', {val:201} }
// SQL gerado: SELECT ID from Books where ID = ?
```

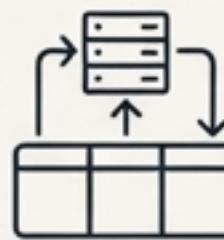
✗ Inseguro: Concatenação de Strings

A concatenação manual de strings é a única forma de criar uma vulnerabilidade. **NUNCA FAÇA ISSO.**

“Never use string concatenation when constructing queries!”

```
let input = "0; DELETE from Books; -- gotcha!";
// Input malicioso
let books = await SELECT.from`Books`.where('ID='
+ input);

// SQL gerado: SELECT ID from Books where ID=0;
DELETE from Books;
```



Pilar 5: Maestria em CRUD

Lendo Dados com 'SELECT'

A API fluente permite construir queries de leitura complexas de forma declarativa e legível, encadeando métodos para especificar projeções, filtros, ordenação e limites.

```
// Uma query que busca livros com estoque > 10,  
// projetando o ID, título e o nome do autor associado,  
// ordenado por título e limitado a 25 resultados.  
const { Books } = cds.entities;  
await SELECT.from(Books, b => {  
    b.ID,  
    b.title,  
    b.author(a => {  
        a.name  
    })  
}).where({ stock: { '>': 10 } })  
.orderBy({ title: 'asc' })  
.limit(25);
```

Projeta colunas e expande associações

Filtra os resultados

Ordena o conjunto

Limita o número de registros

Combine métodos de forma fluente para expressar qualquer necessidade de leitura de dados, desde as mais simples até as mais complexas.



Pilar 5: Maestria em CRUD

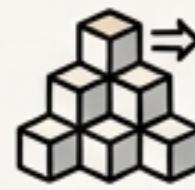
Escrevendo Dados com `INSERT` e `UPsert`

A inserção de dados, seja um único registro ou um lote, segue o padrão claro ` `.into(Entidade).entries(Dados)` . `UPsert` usa a mesma estrutura.

```
const booksToCreate = [
  { ID:201, title:'Wuthering Heights', author_ID:101, stock:12 },
  { ID:251, title:'The Raven', author_ID:150, stock:333 },
  { ID:271, title:'Catweazle', author_ID:170, stock:222 }
];
```

```
// INSERT é uma função que aceita os dados como argumento
await INSERT(booksToCreate).into(Books);
```

`UPsert(books).into(Books)` segue exatamente o mesmo padrão para operações de "update or insert".



Pilar 5: Maestria em CRUD

Modificando Dados com `UPDATE` e `DELETE`

Atualizar e deletar registros utiliza a mesma lógica de filtro do `SELECT`, garantindo consistência e previsibilidade em todas as operações.

UPDATE

Atualiza uma entidade específica (identificada pela chave) usando o método `with()` para aplicar as alterações. Expressões podem ser usadas para atualizações relativas.

```
// Diminui o estoque do livro com ID 201 em 1
// unidade
await UPDATE(Books, 201).with({ stock: { '-=':
1} });
```

DELETE

Deleta registros de uma entidade com base em uma cláusula `where`.

```
// Deleta todos os livros com estoque igual a
// zero
await DELETE.from(Books).where({ stock: 0 });
```

Sua Arma Secreta para a Maestria: `cds repl`

A maneira mais rápida de aprender, testar e prototipar queries é usando o REPL (Read-Eval-Print Loop) do CAP. Use `cds repl` para (Read-Eval-Print Loop) do CAP. Use `cds repl` para experimentar interativamente e ver o CQN exato que suas queries geram.

```
$ cds repl
> cds.ql`SELECT from Authors { ID, name, books[orderBy: title] { ID, title } }`
```

```
{  
  SELECT: {  
    from: { ref: [ 'Authors' ] },  
    columns: [  
      { ref: [ 'ID' ] },  
      { ref: [ 'name' ] },  
      { ref: [ { id: 'books', orderBy: [...] } ],  
        expand: [...] }  
    ]  
  }  
}
```

“It is a great way to learn how to construct queries and to experiment with them.”

Use o `cds repl` como sua caixa de areia para acelerar o desenvolvimento e aprofundar seu entendimento.

Resumo da Jornada: Os Princípios da Maestria



Pense em CQN, não em SQL: Abrace a abstração para obter portabilidade e poder.



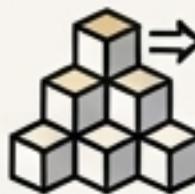
Construção Flexível: Escolha entre a clareza da **API Fluente** e o poder expressivo dos **TTLs**. Use **Definições Refletidas** para um código robusto.



Queries como Objetos: Componha e reutilize queries para um código mais limpo e performático, aproveitando a *Late Materialization*.



Segurança por Padrão: Confie nos *binding parameters* e **nunca** concatene inputs de usuário em suas queries.



Maestria em CRUD: Domine os padrões consistentes para `SELECT`, `INSERT`, `UPDATE` e `DELETE`.



Experimentação Rápida: Acelere seu aprendizado e prototipagem com o `cds repl`.

A Jornada Continua

Você agora comprehende os princípios estratégicos por trás do `cds.ql`. A verdadeira maestria vem com a prática e a exploração contínua. Use esses pilares como seu guia para escrever queries mais eficientes, seguras e elegantes.



Aprofunde seus Conhecimentos

A documentação oficial é a fonte definitiva para todos os métodos, expressões e funcionalidades avançadas.

capire.sap.com/docs/node.js/cds-ql