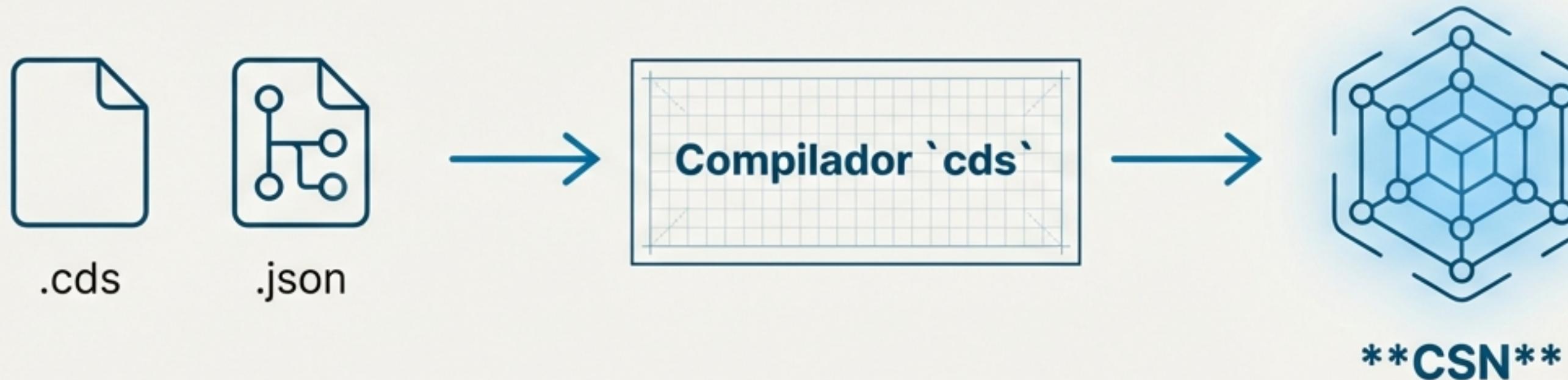


Dominando o Compilador de Modelos CAP

Da Definição ao Artefato Final: Uma Jornada pelo Ciclo de Vida do Modelo



A Função Essencial do CSN

Todo modelo CAP, seja definido em arquivos ` `.cds` ou outras fontes, precisa ser traduzido para um formato unificado e processável pelo framework. Este formato é o **CSN (Core Schema Notation)**.

O CSN é a representação abstrata e canônica do seu modelo. É o **pivô central** a partir do qual todos os artefatos finais — como esquemas de banco de dados e metadados de serviço — são gerados.

O Motor Principal: cds.compile()

Esta é a função central para compilar modelos para CSN. Ela se adapta com base no tipo do primeiro argumento, operando de forma síncrona ou assíncrona.

Compilando de Arquivos (Assíncrono)

Quando o primeiro argumento é um array de nomes de arquivos, um padrão glob ('*') ou uma string file:, os arquivos são lidos e compilados assincronamente.

Internamente, utiliza cds.resolve para encontrar os arquivos.

```
// Compila todos os modelos encontrados nos diretórios raiz
let csn = await cds.compile('*');

// Compila modelos de diretórios específicos
let csn = await cds.compile(['db', 'srv', 'app']);
```

Compilando de Strings em Memória (Síncrono)

Passe uma única string CDL para uma compilação síncrona simples, ou um objeto com múltiplas fontes nomeadas para permitir a resolução de cláusulas using from.

```
// Cláusulas 'using from' não são resolvidas aqui
let csn = cds.compile(`entity Foo { key ID: UUID; }`);
```

```
// Cláusulas 'using from' são resolvidas
let csn = cds.compile({
  'db/schema.cds': `using {cuid} from '@sap/cds/common';
  entity Foo : cuid { foo:String }`,

  'srv/services.cds': `using {Foo} from '../db/schema';
  entity Bar as projection on Foo;`});
```

Ajustando o Motor: Opções do cds.compile()

Controle a granularidade e o conteúdo do CSN resultante passando um objeto de opções.

flavor	'parsed', 'inferred'	Define o 'sabor' do CSN. O padrão é 'inferred', um modelo efetivo com todos os includes, extensões e projeções aplicados. Especifique 'parsed' para obter apenas os modelos individuais analisados.
min	boolean	Se true, aplica cds.minify() após a compilação para remover definições não utilizadas.
docs	boolean	Se true, captura todos os comentários de documentação (/** ... */) e os inclui no CSN.
locations	boolean	Se true, preserva as propriedades \$location no CSN serializado, indicando a origem de cada definição.
messages	[]	Passe um array vazio para coletar todas as mensagens (erros, avisos) do compilador.

```
let options = {  
    min: true,  
    docs: true,  
    flavor: 'inferred'  
};  
let csn = await cds.compile('*', options);
```

Utilitários de Suporte: `cds.load()` e `cds.resolve()



`cds.resolve(paths)` - Encontrando seus Arquivos

Propósito: Resolve os caminhos de origem fornecidos, retornando um array com os nomes de arquivo absolutos dos modelos `.cds` ou `.csn` a serem compilados. É a primeira etapa do processo de compilação a partir de arquivos.

Como Funciona

- Se `paths` for `'*'`: busca nos `cds.env.roots` e `cds.requires.<srv>.model`.
- Verifica a existência de `.cds`, .csn`, /index.cds`, /index.csn`.
- Retorna `undefined` se nenhum arquivo for encontrado.

```
> cds.resolve('*') // Retorna os arquivos resolvidos existentes
> cds.resolve(['db', 'srv']) // Idem para diretórios específicos
```



`cds.load(files)` - O Atalho Conveniente

Propósito: Essencialmente um atalho para `cds.compile([...])`. Além de compilar, ele emite o evento `cds 'loaded'`, integrando-se ao ciclo de vida do serviço.

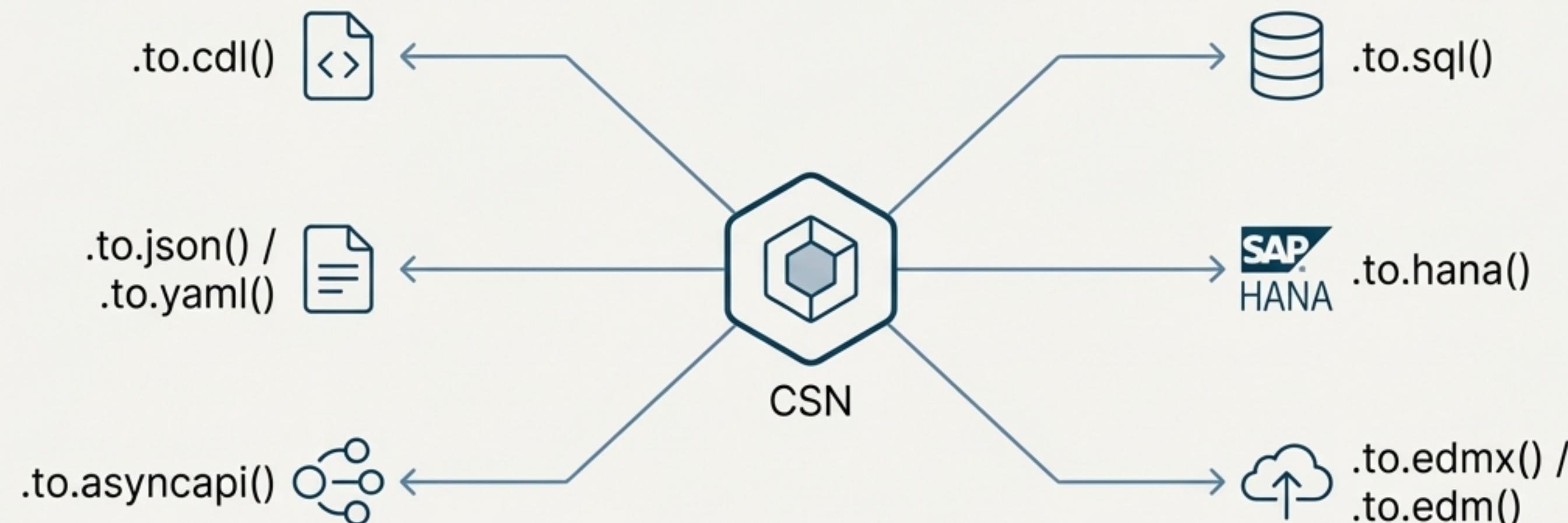
Dica Profissional

Recomenda-se omitir as extensões dos arquivos (`.cds`). Isso permite que o `cds.load` utilize automaticamente arquivos `.csn` pré-compilados, se existirem, otimizando o tempo de inicialização.

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

O Destino: Transformando CSN em Artefatos Finais

Uma vez que você tem um modelo CSN, a suíte `cds.compile.to...` é o seu centro de transformação. Esses métodos recebem um CSN como entrada e o compilam para um formato de saída específico.



```
// O CSN pode ser passado para um método 'compile' inicial  
// para criar uma API fluente.  
let sql = cds.compile(csn).to.sql({ dialect: 'sqlite' });
```

Camada de Persistência: Gerando Artefatos para Banco de Dados

cds.compile.to.sql(csn, options)

Gera instruções SQL DDL (Data Definition Language) a partir do modelo CSN. Por padrão, retorna um array de strings, cada uma sendo uma instrução DDL.

Opções Principais

- **dialect**: Escolhe o dialeto SQL a ser gerado. Valores: 'plain', 'sqlite', 'postgres', 'h2'.
- **names**: Permite gerar DDL com nomes entre aspas. Valores: 'plain', 'quoted'.
- **as**: Se 'str', retorna uma única string com todas as instruções DDL concatenadas.

```
// Gera um array de DDLs para SQLite
let ddls = cds.compile(csn).to.sql({ dialect: 'sqlite' });

// Gera um único script de DDL
let script = cds.compile(csn).to.sql({ as: 'str' });
```

cds.compile.to.hana(csn, options)

Gera artefatos específicos do SAP HANA, como .hdbtable e .hdbview. Retorna uma função geradora (generator) que produz pares [content, { file }].

Uso Avançado

O parâmetro beforeImage pode ser passado para calcular dados adicionais para arquivos .hdbmigrationtable, relevante para ferramentas de build.

O método .hdbtable() está obsoleto. Utilize .to.hana().

```
const all_artifacts = cds.compile.to.hana(csn);
for (const [content, { file }] of all_artifacts) {
  console.log(file, content);
}
```

Camada de Serviço: Gerando Metadados para APIs

cds.compile.to.edm(csn, options)
.edmx(csn, options)

Compila o CSN para um modelo **OData V4 EDM (Entity Data Model)** ou **EDMX** (formato XML).

A Opção Crucial: `service`

Se o seu modelo CSN contém **mais de uma** definição de serviço, a opção `service` é **obrigatória**.

- Para um único serviço: { service: 'CatalogService' } - Retorna o objeto EDM para o serviço especificado.
- Para todos os serviços: { service: 'all' } - Retorna um gerador (**generator**) que itera sobre cada serviço, produzindo [edm, { file, suffix }].

```
// Para um serviço específico
let edm = cds.compile.to.edm(csn, { service: 'Catalog' });

// Para todos os serviços em um modelo
let all = cds.compile.to.edm(csn, { service: 'all' });
for (let [edm, { file, suffix }] of all) {
  console.log(file, suffix, edm);
}
```

cds.compile.to.asyncapi(csn)

Converte o CSN em um documento no formato AsyncAPI, útil para arquiteturas orientadas a eventos.

```
const doc = cds.compile.to.asyncapi(csn);
```

O Kit de Ferramentas do Especialista: `cds.parse()`

Às vezes, você não precisa compilar um modelo inteiro. `cds.parse()` oferece um conjunto de funções para analisar fragmentos de sintaxe CAP – como uma única entidade, uma query CQL ou uma expressão.

`cds.compile`

Para modelos completos. Processa, infere e valida o modelo como um todo.

`cds.parse`

Para fragmentos. Analisa sintaticamente uma string e retorna sua representação estruturada (CSN ou CQN) sem aplicar inferências complexas.

Métodos Disponíveis

- `cds.parse.cdl()`: Analisa uma string de sintaxe CDL. É um atalho para `cds.compile(..., { flavor: 'parsed' })`.
- `cds.parse.cql()`: Analisa uma string de sintaxe CQL e retorna a representação CQN (Core Query Notation).
- `cds.parse.expr()`: Analisa uma expressão CQL e retorna sua representação em árvore.
- `cds.parse.xpr()`: Atalho para `cds.parse.expr`.
- `cds.parse.ref()`: Analisa uma referência de caminho.

Todos os métodos suportam tanto chamadas de função padrão quanto o uso como *tagged template strings* para uma sintaxe mais limpa.

`cds.parse()` em Ação: Entradas e Saídas

cds.parse.cdl()

```
// Entrada (CDL)
let csn = cds.parse.cdl(`entity Foo {}`);
// ou como atalho
let csn2 = cds.parse(`entity Foo {}`);
```

cds.parse.cql()

```
// Entrada (CQL)
let cqn = cds.parse.cql(`SELECT * from Foo`);
// ou como tagged template
let cqn2 = cds.parse.cql`SELECT * from Foo`;
```

cds.parse.expr()

```
// Entrada (Expressão)
let cxn = cds.parse.expr(`foo.bar > 9`);
```

Saída

Um objeto CSN parcial representando a entidade `Foo`.

Saída

Um objeto CQN representando a query `SELECT`.

Saída

```
// Saída (Estrutura da Expressão)
{
  "xpr": [
    { "ref": ["foo", "bar"] },
    ">",
    { "val": 9 }
  ]
}
```

Otimizando o Resultado Final: `cds.minify()`



O “Porquê”

Por que minificar um modelo CSN? Para remover todas as definições que não são utilizadas, ou seja, que não são alcançáveis a partir das raízes do seu modelo (serviços e entidades persistidas).



Benefício Principal

Isso é especialmente útil ao usar bibliotecas como `@sap/cds/common`. Entidades como Countries, Currencies e Languages são marcadas com `@cds.persistence.skip: 'if-unused'`. `cds.minify()` as removerá do seu CSN final se você não as utilizar, resultando em um esquema de banco de dados mais limpo e enxuto.

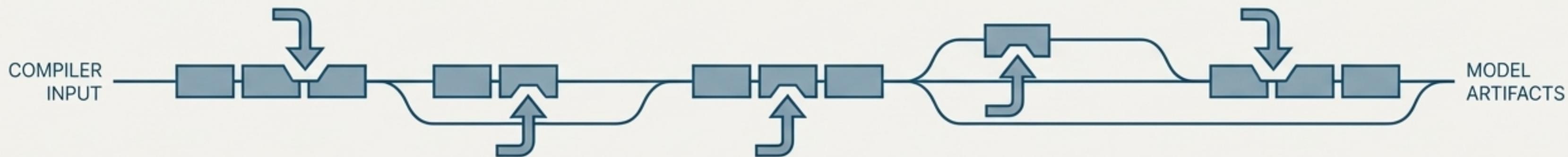
Como Usar

```
// Encadeado após carregar o modelo
let csn = await cds.load('*').then(cds.minify);
```

Exemplo de Saída do CLI (`cds minify "*" --dry`)

<input checked="" type="checkbox"/> Keep	<input type="checkbox"/> Skip
<ul style="list-style-type: none">• AdminService• AdminService.Books• sap.capire.bookshop.Books• User• sap.capire.bookshop.Authors...	<ul style="list-style-type: none">- Language- Country- sap.common- euid- temporal...

O Próximo Nível: Ganchos e Interceptadores com Eventos de Ciclo de Vida



Conceito

O compilador CAP emite eventos em momentos chave do seu processo. Você pode registrar *handlers* para esses eventos usando `cds.on()` para inspecionar ou modificar o modelo **em trânsito**.

Sintaxe de Registro

```
const cds = require('@sap/cds');

cds.on('compile.for.runtime', model => {
    // Sua lógica de modificação do modelo aqui
});
```



Por que isso é poderoso?

Isso permite aplicar transformações personalizadas, adicionar anotações dinamicamente ou injetar elementos customizados no modelo antes que ele seja consumido pelo runtime ou convertido para um artefato final.



Aviso Importante

Os *handlers* de eventos são executados de forma síncrona, na ordem em que são registrados. Garanta que seus *handlers* sejam idempotentes, pois múltiplos eventos podem ser emitidos para o mesmo modelo durante o processo de bootstrap.

Mapeando os Eventos Chave do Compilador

Cada evento oferece um ponto de intervenção em uma fase específica do ciclo de vida do modelo.

Evento `compile.for.runtime`



Quando:

Emitido uma única vez, antes que o modelo seja compilado para uso no runtime (Node.js ou Java).



Caso de Uso Ideal:

Adicionar elementos customizados que são necessários apenas em tempo de execução, como entidades ou aspectos que não devem ser persistidos no banco de dados.

Evento `compile.to.dbx`



Quando:

Emitido uma única vez, antes que os artefatos específicos do banco de dados (ex: scripts SQL DDL) sejam gerados.



Caso de Uso Ideal:

Adicionar elementos ou anotações customizadas que são relevantes apenas para a camada de persistência.

Evento `compile.to.edmx`



Quando:

Emitido imediatamente antes que o modelo seja compilado para EDMX.

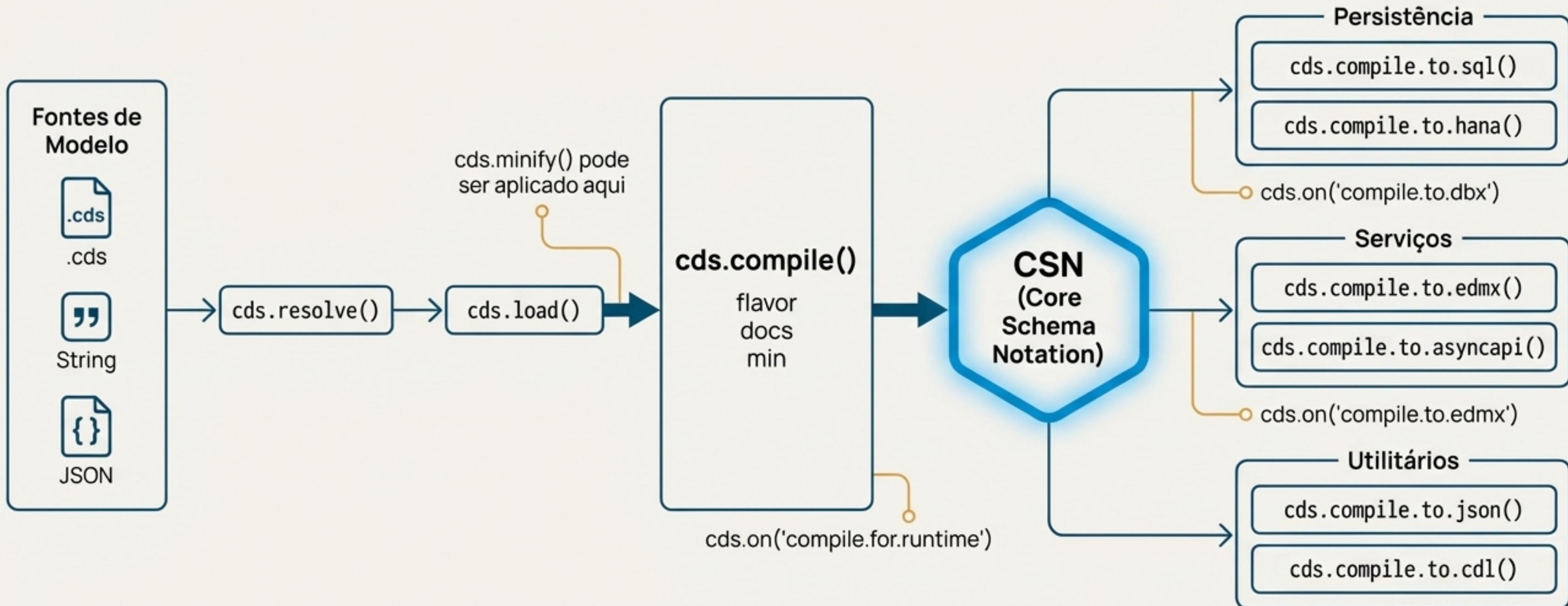


Caso de Uso Ideal:

Adicionar anotações de UI ou metadados específicos de Fiori ao modelo de serviço antes que ele seja exposto via OData.

```
cds.on('compile.for.runtime', model => { /* ... */ });
cds.on('compile.to.dbx', model => { /* ... */ });
cds.on('compile.to.edmx', model => { /* ... */ });
```

A Imagem Completa: O Mapa do Ciclo de Vida do Modelo CAP



Compreender este fluxo, do código-fonte ao artefato final, é a chave para dominar o SAP Cloud Application Programming Model.

Fontes de
Modelo



cds.compile()

CSN

Explore a documentação oficial para mais
detalhes e exemplos avançados.

cap.cloud.sap
github.com/sap/cap

Persistência

cds.contapt -> add()
cds.blamndjwri() --> Persistência
cds.compile() --> Serviços
cds.compiler() --> Utilitários

Serviços

cds.recompile() -> serviços
cds.vmt.pasmtzeo() ->
cds.compiledbicsn() -serviço

Utilitários

cds.compile() --> Mange
cds.service}*() --> Serviços
cds.compiler() --> Utilitários