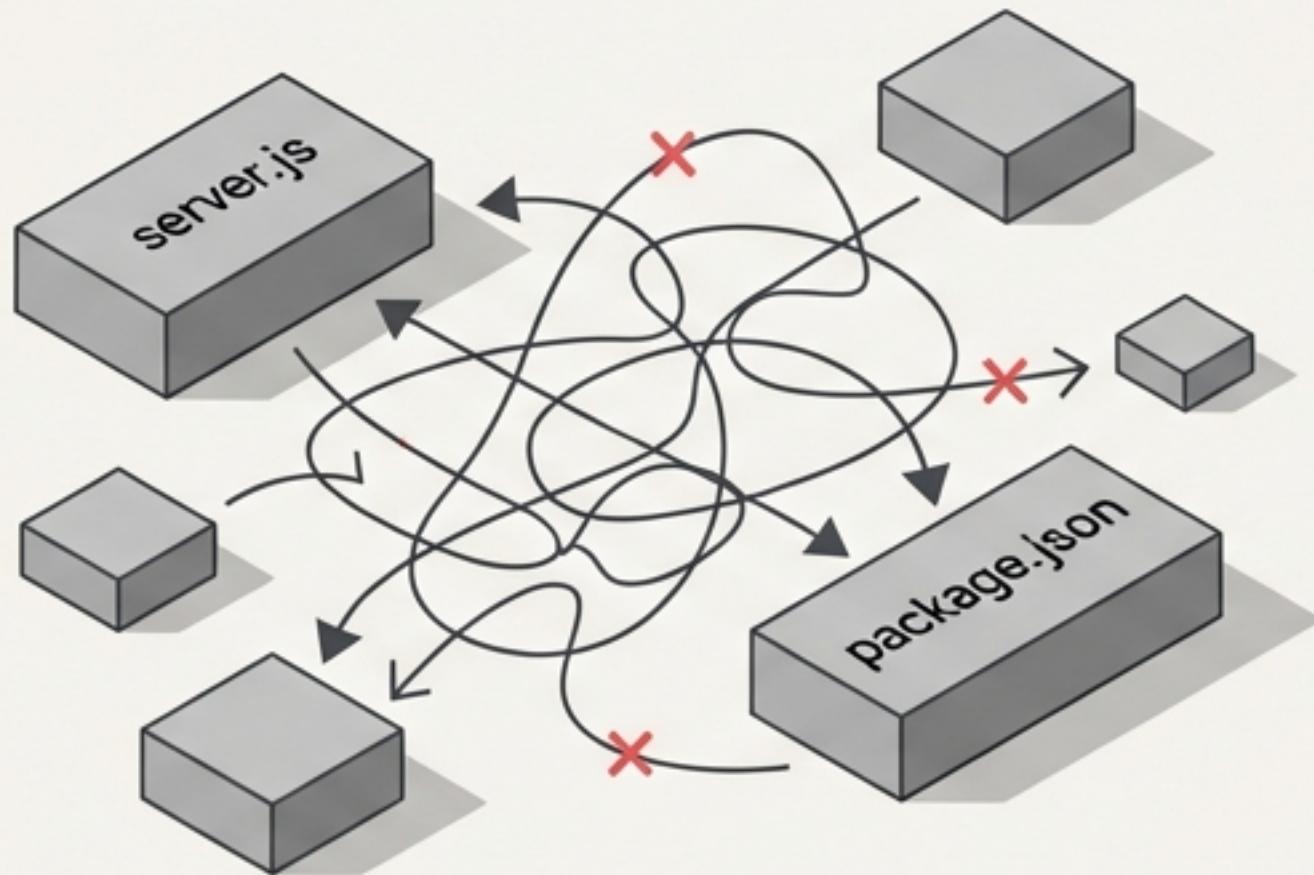




Construindo Extensões Modulares para CAP com CDS Plugin Packages

Um guia para criar pacotes reutilizáveis e com
auto-configuração para o Node.js.

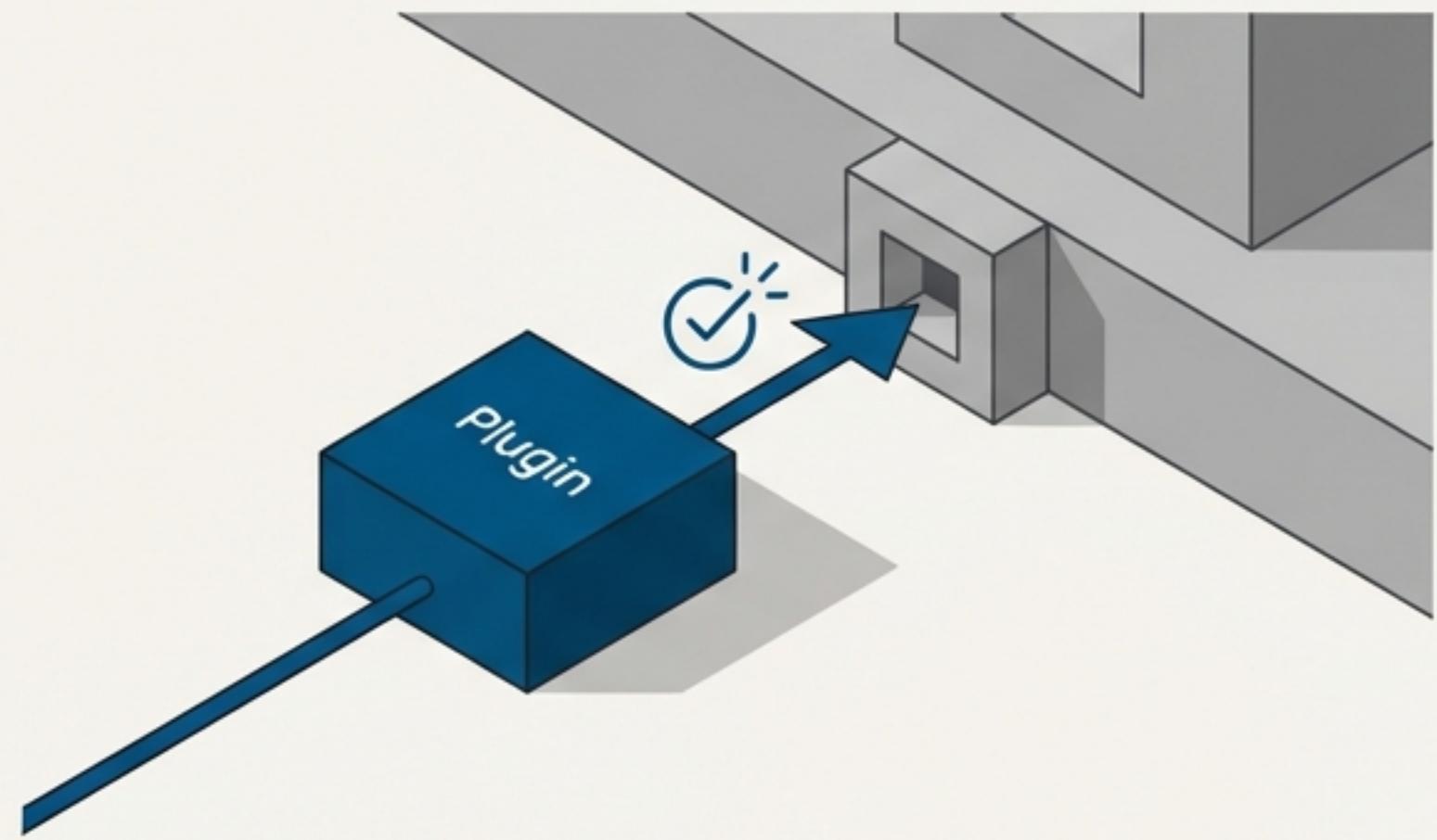
O Desafio: Extensões Reutilizáveis e a Carga da Configuração Manual



Em ecossistemas CAP, frequentemente criamos lógicas ou serviços que gostaríamos de reutilizar em múltiplos projetos (ex: logging customizado, integração com serviços, UI helpers).

Tradicionalmente, isso exige que cada projeto consumidor configure manualmente o serviço no package.json e adicione código de inicialização no server.js. Isso é repetitivo e propenso a erros.

A Oportunidade

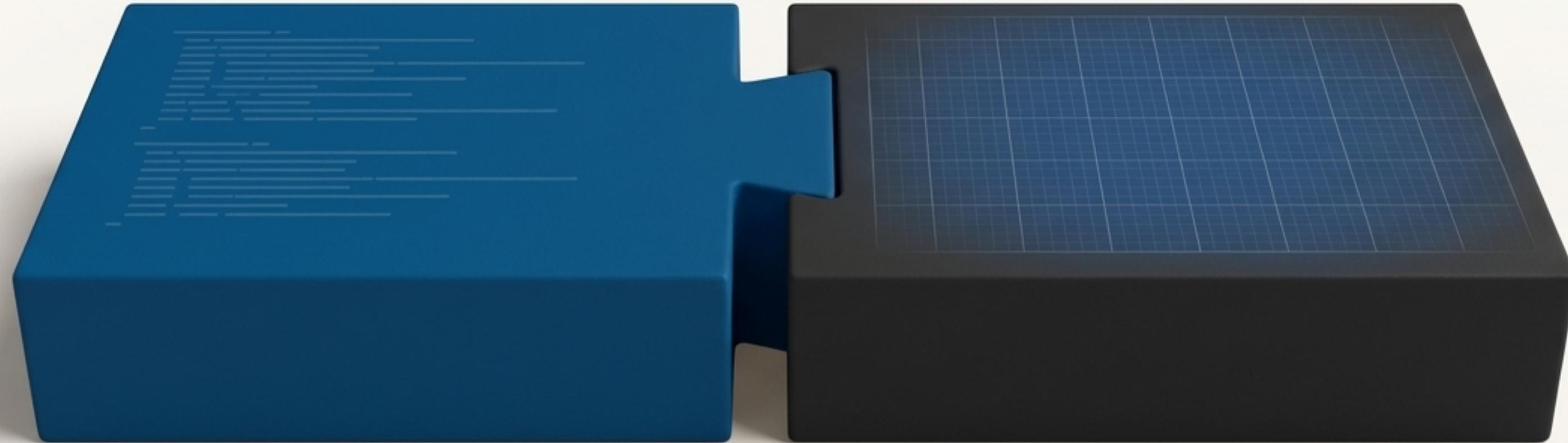


E se pudéssemos criar pacotes que se configuram sozinhos? Pacotes 'plug-and-play' que simplesmente funcionam ao serem instalados.

É exatamente isso que os **CDS Plugin Packages** permitem. Eles oferecem um mecanismo para auto-configuração e registro automático de lógica.

A Anatomia de um CDS Plugin

`'cds-plugin.js'`



Onde sua lógica customizada reside.
Reage a eventos do ciclo de vida do CAP,
exatamente como um `'server.js'`.

Onde a 'mágica' da auto-configuração
acontece. Define serviços, presets e schemas
para uma experiência 'zero-config'.

Juntos, esses dois arquivos transformam um pacote Node.js padrão em uma extensão poderosa e autônoma para o ecossistema CAP.

A Fundação: Lógica Reativa

O Primeiro Bloco: O Arquivo Arquivo `cds-plugin.js`

O Que É?

Um arquivo chamado `cds-plugin.js` na raiz do seu pacote (ao lado do `package.json`). Ele é o ponto de entrada para a lógica do seu plugin.

Como Funciona?

O CAP detecta e carrega este arquivo automaticamente. Dentro dele, você pode usar o objeto `cds` para interagir com o framework.

`cds-plugin.js`

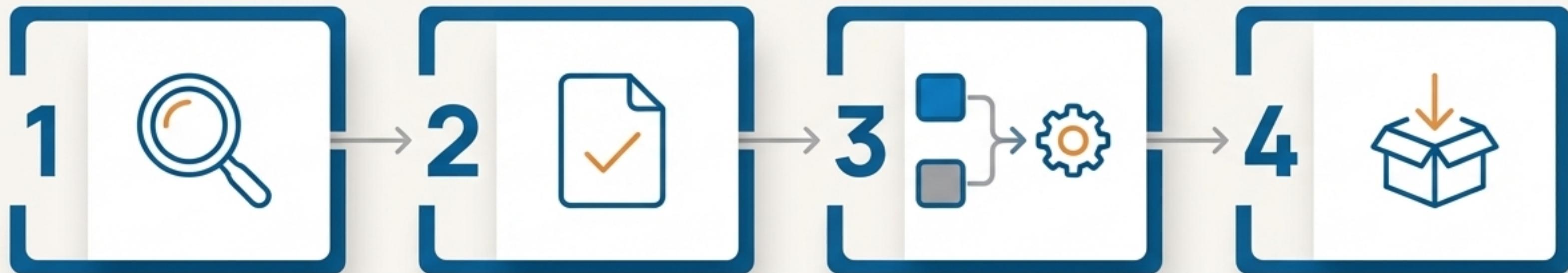
```
// cds-plugin.js
const cds = require('@sap/cds')

// Registra um handler para o evento 'served'
// Exatamente como você faria em um server.js!
cds.on('served', () => {
  console.log('Nosso plugin foi carregado e os
    serviços estão no ar!')
  // ... adicione sua lógica aqui
})
```

Por Trás das Cenas: Como o CAP Carrega seu Plugin

O mecanismo de plugin é ativado por padrão em comandos essenciais como `cds serve`, `cds watch`, `cds run`, `cds build` e `cds deploy`.

O Processo de Descoberta



Scan

O CAP verifica todas as dependencies e devDependencies do seu projeto.

Select

Ele seleciona todos os pacotes que contêm um arquivo cds-plugin.js em sua raiz.

Merge

A configuração da seção cds do package.json de cada plugin é adicionada ao ambiente (`cds.env`).

Load

Finalmente, o módulo cds-plugin.js de cada plugin é carregado (require).

Adicionando Inteligência: Auto-Configuração

O Segundo Bloco: Configuração no `package.json`

O Que É?

A capacidade de um plugin fornecer configurações padrão para o projeto que o consome, adicionando uma seção `cds` ao seu próprio `package.json`.

Por Que é Poderoso?

Isso **elimina a necessidade** de o desenvolvedor configurar manualmente os `requires` ou outros ajustes no `package.json` do projeto principal. É a chave para a experiência 'plug-and-play'.

Exemplo: O Consumidor

Para usar o plugin, o desenvolvedor apenas o instala. O `package.json` do projeto consumidor pode ser tão simples quanto isto:

`package.json` (do projeto consumidor)

```
{  
  "name": "test-app",  
  "version": "1.0.0",  
  "devDependencies": {  
    "cds-swagger-ui-express": "file:../../"  
  },  
  "cds": {}  
}
```

Exemplo Prático: Desvendando o `@cap-js/sqlite`

O pacote `@cap-js/sqlite` fornece uma configuração de banco de dados SQLite para o ambiente de desenvolvimento sem exigir nenhuma configuração do usuário. Veja como ele faz isso em seu `package.json`:

```
{  
  "cds": {  
    "requires": {  
      "db": "sql" ← 1  
    },  
    "kinds": {  
      "sql": {  
        "[development)": { ← 2  
          "kind": "sqlite"  
        }  
      },  
      "sqlite": { ← 3  
        "impl": "@cap-js/sqlite"  
      }  
    }  
  }  
}
```

- 1 Automaticamente configura o serviço `db` para usar o preset `sql`.
- 2 No ambiente de desenvolvimento...
- 3 ...o preset `sql` usa o preset `sqlite`, que por sua vez aponta para este mesmo pacote como sua implementação.

Polindo a Experiência: Schemas de Configuração (beta)

O Bloco Final: Schemas para uma Melhor DX



O Que São?

Uma forma de declarar as opções de configuração que seu plugin oferece. Essas declarações são adicionadas ao `package.json` do seu plugin.

O Principal Benefício: Melhorar a Experiência do Desenvolvedor (DX)

Ferramentas e IDEs podem usar esses schemas para fornecer:



Code Completion
(Autocompletar)



Validação de
Propriedades

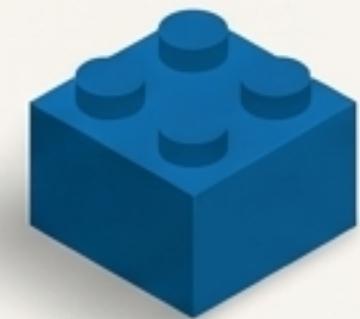


Documentação
Inline

Como Declarar

Todas as definições de schema devem ser colocadas sob o nó `schema` dentro da seção `cds` do seu `package.json`.

Definindo Novos Tipos com Schemas



Você pode estender os tipos fundamentais do CAP, como `buildTaskType` ou `databaseType`, para que seu plugin seja reconhecido nativamente pelas ferramentas.

Build Task

```
// package.json (do seu plugin)
"cds": {
  "schema": {
    "buildTaskType": {
      "name": "new-buildTaskType",
      "description": "Uma descrição do novo tipo de build task."
    }
  }
}
```

Database

```
// package.json (do seu plugin)
"cds": {
  "schema": {
    "databaseType": {
      "name": "new-databaseType",
      "description": "Uma descrição do novo tipo de banco de dados."
    }
  }
}
```

Adicionando Configurações de Alto Nível



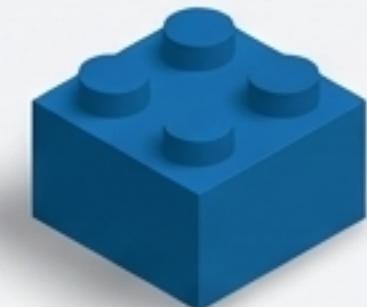
Para configurações mais complexas e específicas do seu plugin, você pode introduzir um novo objeto de configuração diretamente sob a chave `cds`.

Exemplo Prático: `cds-swagger-ui-express`

Este plugin adiciona uma chave `swagger` para permitir que os usuários configurem a geração da UI do Swagger.

```
// package.json (do plugin cds-swagger-ui-express)
"cds": {
  "schema": {
    "swagger": { // Define uma nova propriedade 'swagger' no nível raiz do 'cds'
      "description": "Configurações para o Swagger UI",
      "oneOf": [
        // ...schema detalhado para as opções do swagger
      ]
    }
  }
}
```

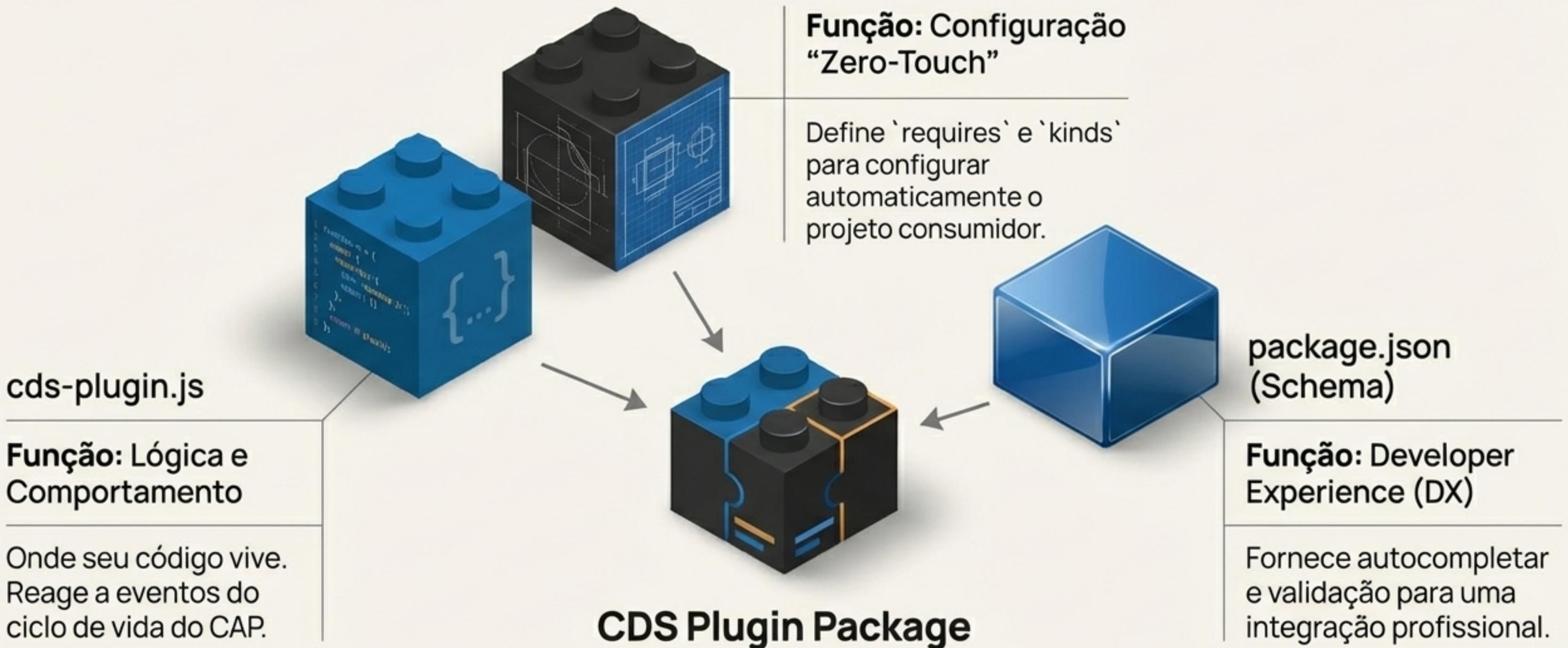
Pontos de Contribuição do Schema: O Que é Suportado?



Ponto de Contribuição	Descrição
'buildTaskType'	Adiciona um novo tipo de tarefa de build.
'databaseType'	Adiciona um novo tipo de banco de dados.
'cds'	Adiciona uma ou mais configurações de alto nível.

Síntese

Resumo: Montando os Blocos do seu Plugin



Seu Checklist para Criar um Plugin de Sucesso

Pronto para construir sua própria extensão modular para o ecossistema CAP? Siga estes passos:



- Crie seu pacote Node.js:** Comece com um `package.json` padrão.
- Adicione a lógica no `cds-plugin.js`:** Implemente o comportamento do seu plugin.
- Defina a auto-configuração:** Adicione a seção `cds` ao seu `package.json` para uma experiência 'plug-and-play'.
- (Opcional) Adicione um `schema`:** Dê um toque profissional e melhore a DX com autocompletar e validação.
- Publique e Compartilhe:** Disponibilize seu plugin para a comunidade via NPM ou registro privado.

