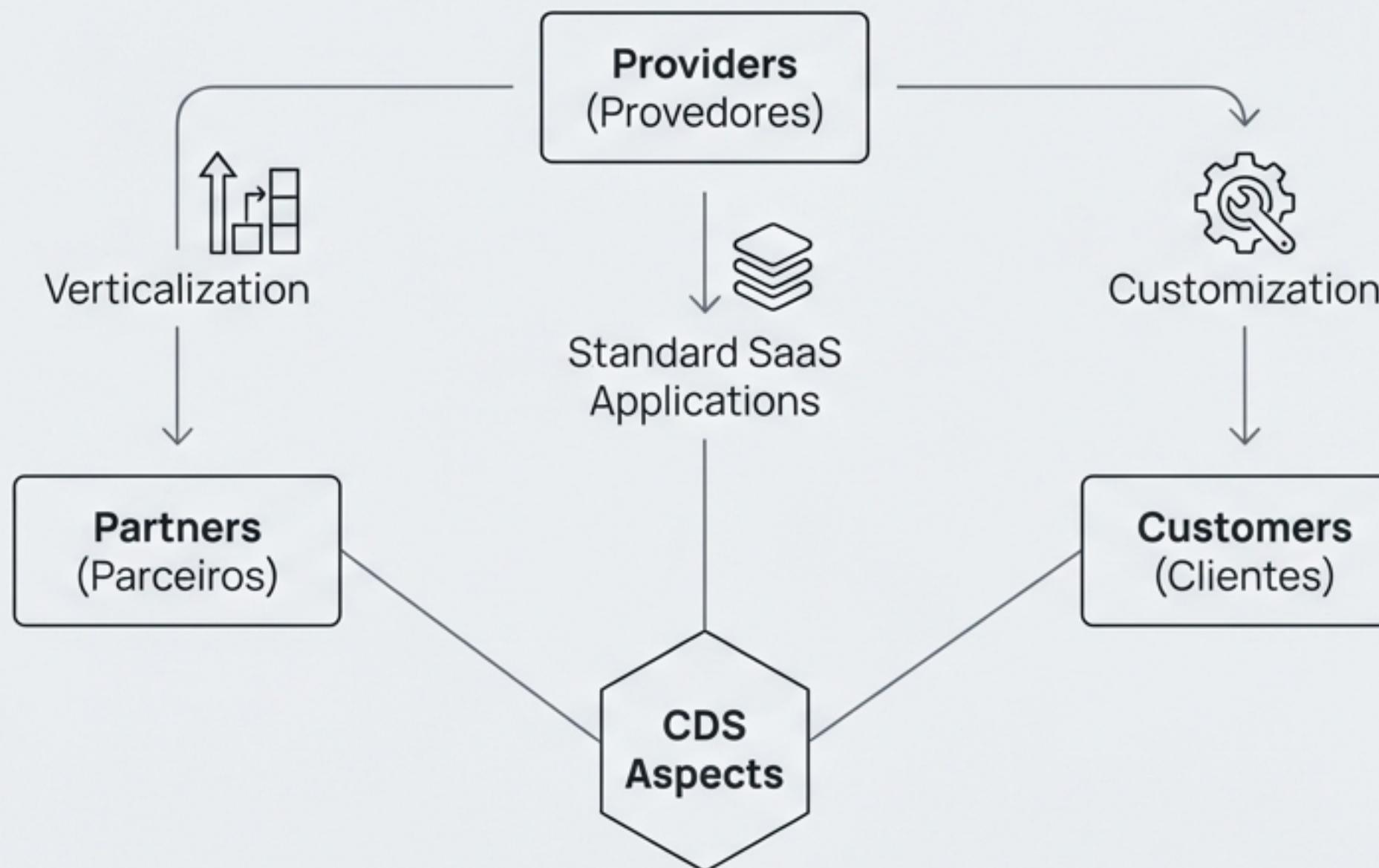


# Dominando a Extensibilidade no CAP

Um Guia de Referência para Arquitetos de Soluções

O SAP Cloud Application Programming Model (CAP) foi projetado para ser flexível. Esta apresentação é um guia prático e aprofundado sobre as três principais ferramentas do seu kit de extensibilidade. Vamos explorar como e quando usar cada padrão para construir soluções robustas, personalizáveis e escaláveis.

# O Ecossistema de Extensibilidade do CAP



A extensibilidade no CAP é um ecossistema colaborativo. Diferentes participantes utilizam um conjunto comum de primitivos, como os CDS Aspects, para alcançar objetivos distintos. Apresentamos os três padrões fundamentais que habilitam este ecossistema:

-  **1. Extensões SaaS**  
(Ajuste Fino)
-  **2. Feature Toggles**  
(Ativação Modular)
-  **3. Reuso e Composição**  
(Construção de Sistemas)

# Padrão 1: Extensões SaaS para "Ajuste Fino"

## O Quê

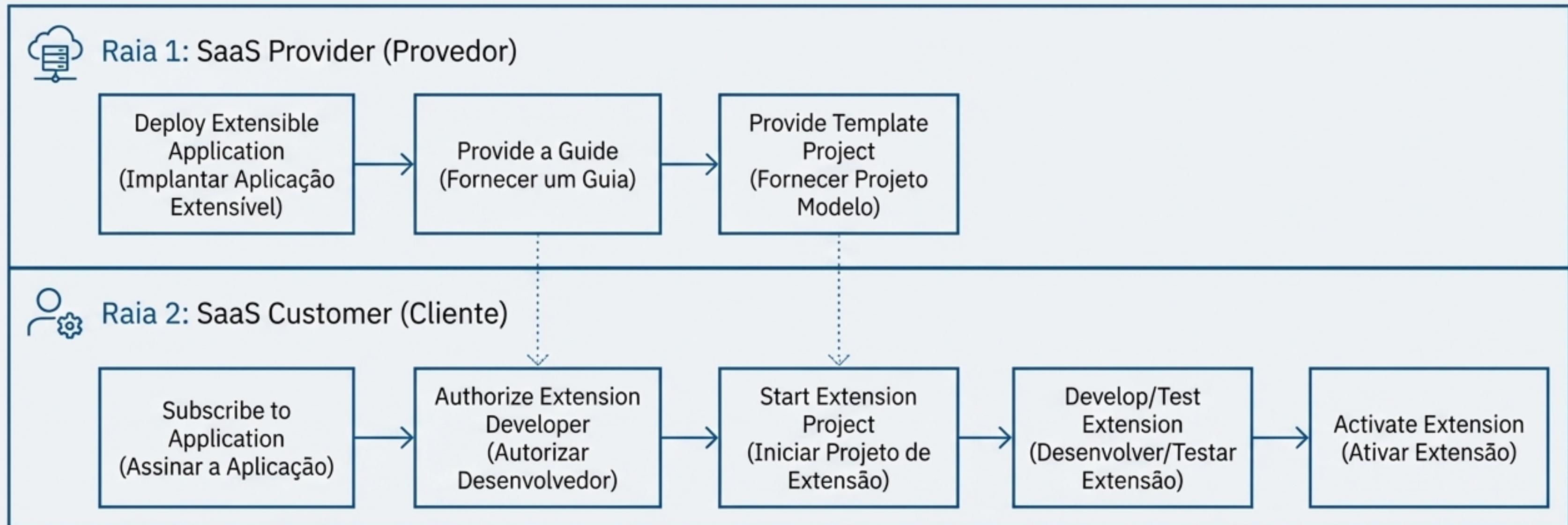
Extensões SaaS permitem que assinantes (clientes) de uma aplicação multitenant a personalizem em nível de tenant, adicionando campos, entidades e lógicas de negócio específicas sem alterar a base de código da aplicação principal.

## Por Quê

Atender às necessidades únicas de cada cliente, que frequentemente precisa adaptar a solução padrão para seus processos de negócio. O CAP oferece suporte intrínseco a essas extensões “out of the box”.



# O Fluxo de Trabalho de Extensão: Provedor e Cliente



O processo é uma parceria. O Provedor prepara a aplicação para ser extensível e fornece as ferramentas. O Cliente utiliza essas ferramentas para desenvolver, testar e ativar suas customizações de forma segura em seu próprio tenant.

# Preparando a Aplicação: O Checklist do Provedor SaaS

## 1. Habilitar Extensibilidade

O comando `cds add extensibility` prepara o projeto. Essencialmente, automatiza a configuração para o sidecar MTX.

```
cds add extensibility
```

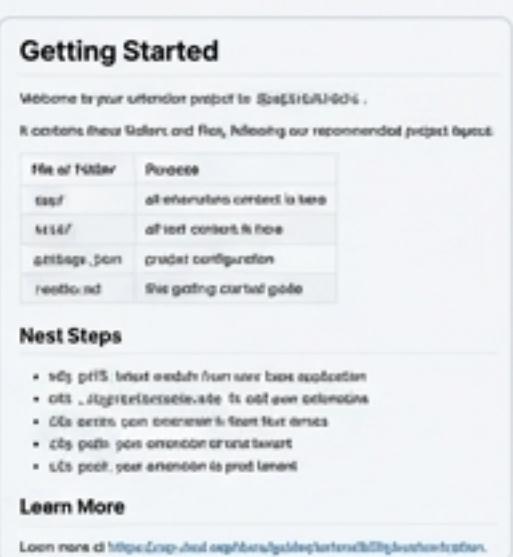
## 2. Restringir Pontos de Extensão

Definir limites claros. Exemplo de `package.json` para controlar quais entidades e serviços podem ser estendidos e em que medida (ex: `element-prefix`, `new-fields`, `new-entities`).

```
"cds": {  
  "requires": {  
    "extension-allowlist": {  
      "bind": "io:@eemory",  
      "config": {  
        "entities": {  
          "Orders": { "new-fields": true },  
          "Items": { "new-fields": true, "element-prefix": "ext_" }  
        },  
        "services": { "CatalogService": { "new-entities": true } }  
      }  
    }  
  }  
}
```

## 4. Criar Guias de Extensão

Documentar o processo para o cliente, incluindo setup de tenants, papéis necessários, e o que pode ser estendido. Um bom `README.md` é fundamental.



## 5. Implantar a Aplicação

Após a preparação, a aplicação é implantada seguindo o guia padrão de deployment.



# Criando a Extensão: A Jornada do Desenvolvedor do Cliente

## Fluxo de Passos

### 1. Setup

`cds subscribe my-tenant  
cds pull --from https://<app-url>`

### 2. Desenvolvimento

`npm install`  
`vi srv/data-model.cds`  
`vi app/annotations.cds`

### 3. Teste Local

`cds watch` para um ciclo de desenvolvimento rápido com dados de teste locais.

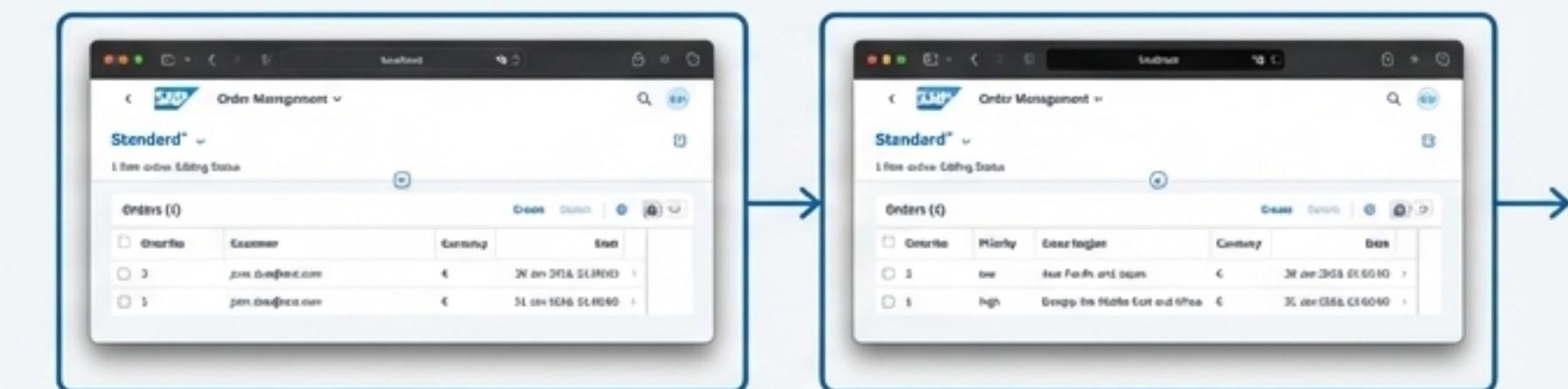
### 4. Ativação

`cds push --to <app-url>` para enviar a extensão ao tenant de teste e, finalmente, ao de produção.

`cds push --to https://<test-tenant-url>`  
`cds push --to https://<prod-tenant-url>`

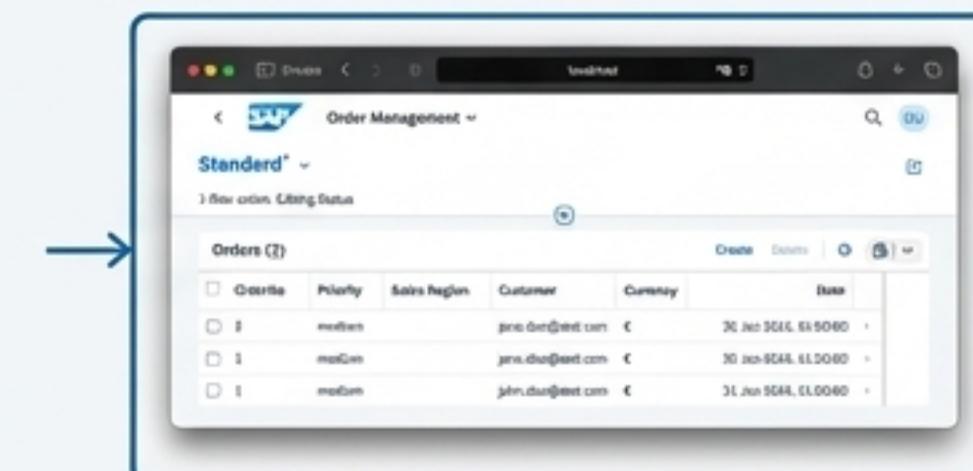
### A função do `cds login`

Para simplificar o fluxo de trabalho, armazenando tokens de autenticação de forma segura e evitando a necessidade de repetir credenciais.



Original

Teste Local (`cds watch`)



Ativado no Tenant (`cds push`)

# A Anatomia de um Modelo de Extensão CDS

## 1. Estendendo o Modelo de Dados

Adicionar novos campos:

```
extend Orders with {  
    x_priority: String;  
}  
...
```

Criar novas entidades:

```
entity x_SalesRegion :  
    sap.common.CodeList {  
        ...  
    }
```

Anotações de validação:

```
@assert.range, @mandatory,  
@assert.unique
```

## 2. Estendendo o Modelo de Serviço

A exposição de entidades é automática, mas a exposição explícita é possível:

```
extend service OrdersService with {  
    entity x_Customers as  
        projection on  
        ...  
}
```

**Dica:** É possível adicionar textos localizáveis através de arquivos `i18n.properties` na sua extensão.

## 3. Estendendo Anotações de UI

Anotar novas entidades (@UI.LineItem, etc.) ou estender arrays existentes:

```
annotate OrdersService.Orders  
with @UI: {  
    LineItem: [  
        ...  
        [... up to {Value: OrderNo},  
            {Value: x_priority}, ...]  
    ]  
},  
})
```



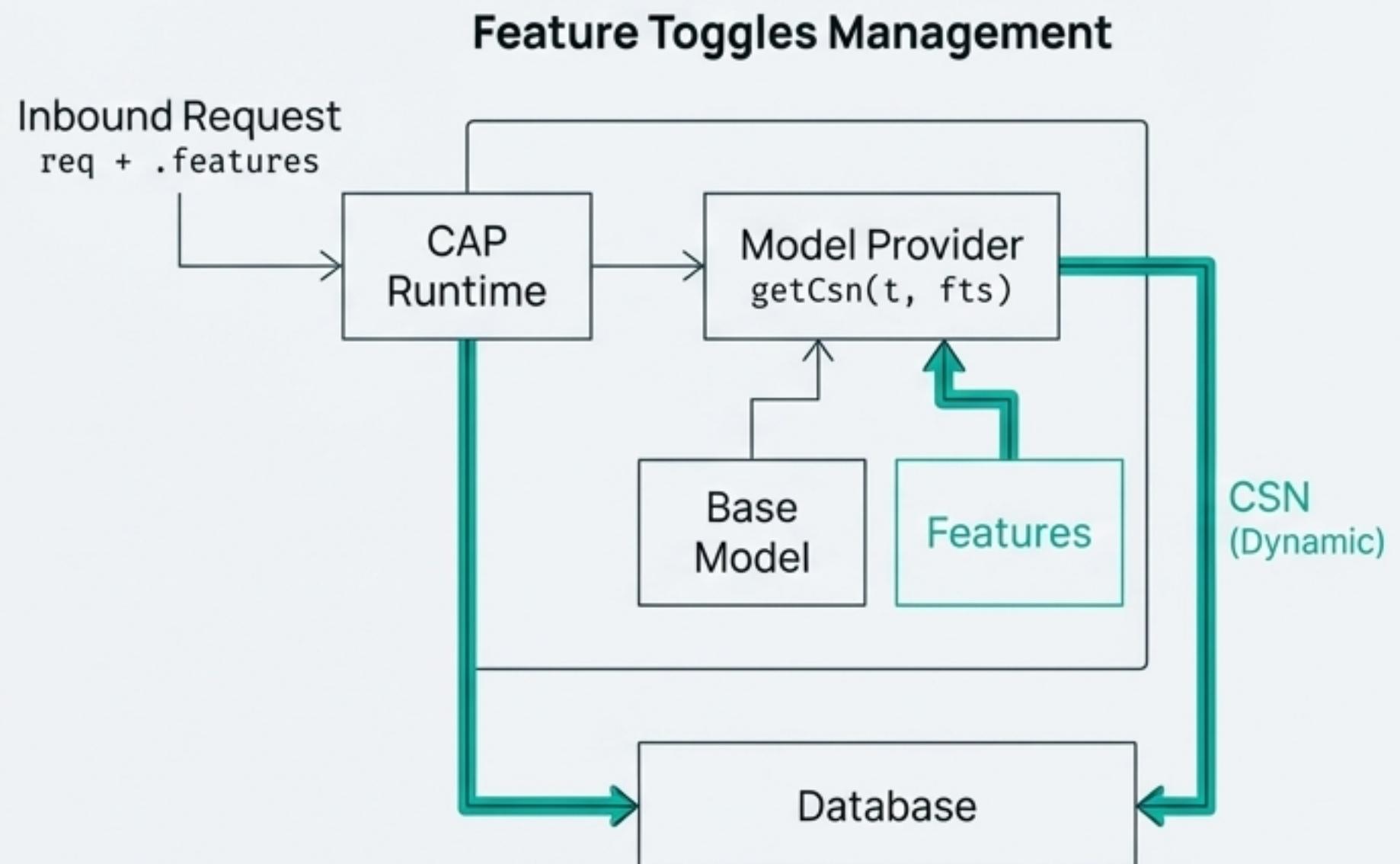
# Padrão 2: Feature Toggles para “Ativação Modular”

## O Quê

Feature Toggles são extensões pré-construídas pelo provedor da aplicação SaaS. Elas são implantadas junto com a aplicação base, mas ativadas dinamicamente em runtime por tenant, usuário ou requisição.

## Por Quê

Oferecer diferentes “sabores” ou edições de um produto SaaS (ex: básico vs. premium), especializações para indústrias, ou realizar lançamentos graduais de novas funcionalidades (canary releases).



# Implementando e Estruturando Features

## Guia de Implementação

### 1. Configuração do Projeto

Configuração do projeto é um projeto, para remasar a configuração  
npm add @sap/cds-mtxs, trn:

```
npm add @sap/cds-mtxs
```

1

```
"cds": {  
  "requires": {  
    "toggles": true  
  }  
}
```

2

### 2. Estrutura de Diretórios

A estrutura de calpa vem à estrutura diretórios commaron ao modelo.

```
project-root/  
└─ fts/  
  └─ isbn/  
    └─ schema.cds  
  └─ reviews/  
    └─ schema.cds
```

### 2. Estrutura de Diretórios

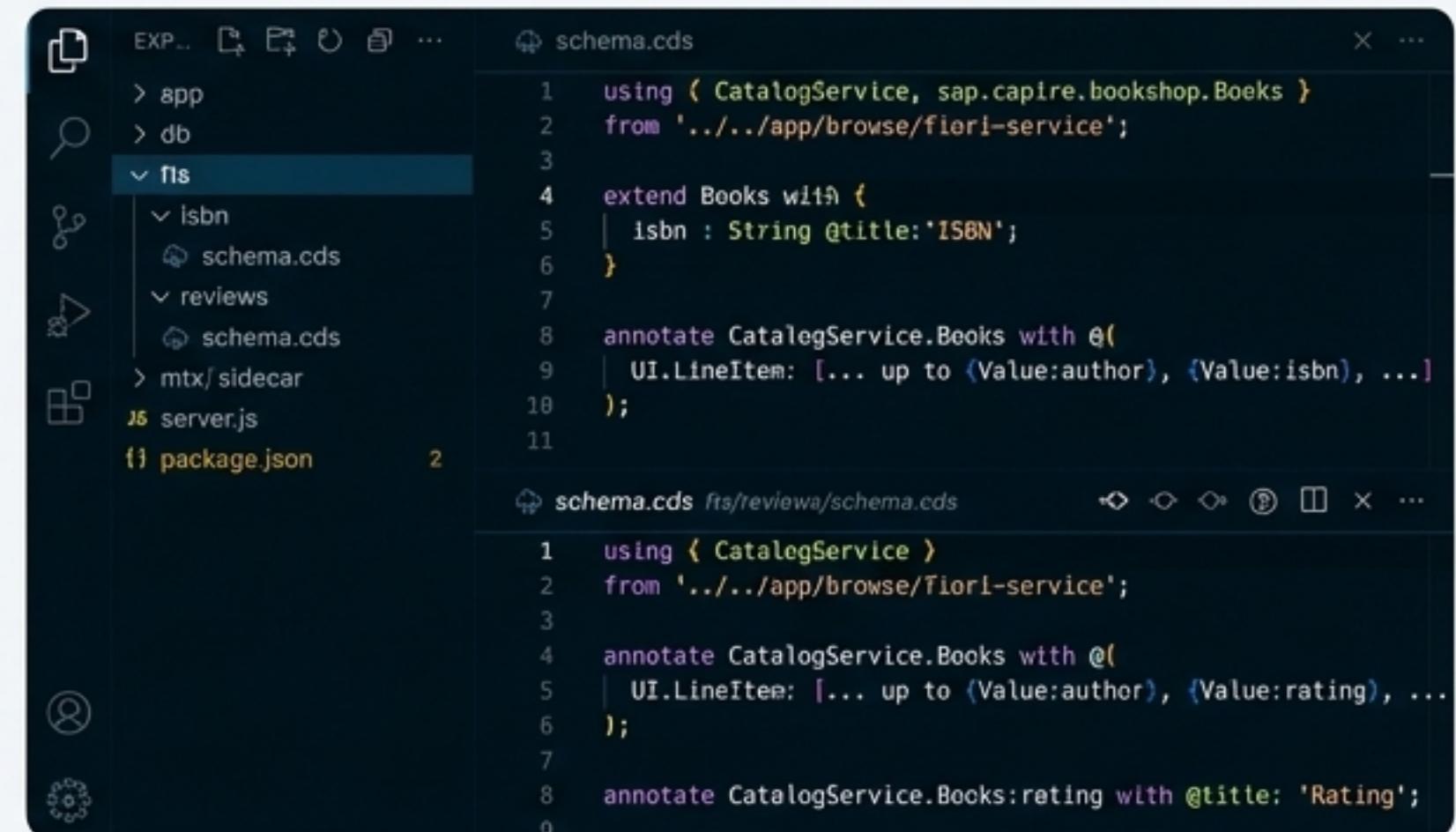
Unnomero os capotas de aostitio sum calea em estrutura de diretórios:

```
project-root/  
└─ fts/  
  └─ isbn/  
    └─ schema.cds  
  └─ reviews/  
    └─ schema.cds
```

### 3. Código da Feature

O código da código a sua taça usual e um carcinho da feature.

```
fts/isbn/schema.cds  
  
using { CatalogService, sap.capi.re.bookshop.Bo  
from '../../../../../app/browse/fiori-service';  
  
extend Books with {  
  isbn : String @title:'ISBN';  
}  
  
annotate CatalogService.Books with @{  
  UI.LineItem: [... up to {Value:author}, {Value:  
};
```



```
schema.cds  
1 using { CatalogService, sap.capi.re.bookshop.Bo  
2 from '../../../../../app/browse/fiori-service';  
3  
4 extend Books with {  
5   isbn : String @title:'ISBN';  
6 }  
7  
8 annotate CatalogService.Books with @{  
9   UI.LineItem: [... up to {Value:author}, {Value:  
10 };  
11 };  
  
schema.cds fts/reviews/schema.cds  
1 using { CatalogService }  
2 from '../../../../../app/browse/Fiori-service';  
3  
4 annotate CatalogService.Books with @{  
5   UI.LineItem: [... up to {Value:author}, {Value:  
6 };  
7  
8 annotate CatalogService.Books.rating with @title: 'Rating';  
9
```



### Atenção

Com Callout Box con o IBM Plex Sans:

- Sem subpastas dentro de um feature.
- Sem dependências using entre features.
- Todas as entidades estendidas devem pertencer ao modelo base.

# Ativando Features em Desenvolvimento e Produção

## Em Desenvolvimento (Com Mocked Auth)

Demonstrar a configuração no `package.json` para simular a ativação de features por tenant e por usuário.

```
{  
  "auth": {  
    "users": {  
      "carol": { "features": [] },  
      "erin": { "features": ["isbn", "reviews"] }  
    },  
    "tenants": {  
      "t1": { "features": ["isbn"] },  
      "t2": { "features": ["reviews"] }  
    }  
  }  
}
```

## Em Produção

Esclarecer que o gerenciamento de tenants/usuários está fora do escopo do CAP. A ativação é feita por um 'Feature Vector Provider'.

```
// server.js  
module.exports = (req, res, next) => {  
  if (req.headers.features) {  
    req.features = req.headers.features.split(',')  
  }  
  next()  
}
```

Title	Author	Rating	ISBN	Genre	Pric	Currency
Wuthering Heights	Emily Brontë	11.11	185127751228	Drama	11.11	GBP £
Jane Eyre	Charlotte Brontë	12.34	122362817629	Romance	12.34	GBP £
The Raven	Edgar Allan Poe	13.13	122153856529	Mystery	13.13	USD \$
Eleenora	Edgar Allan Poe	14.00	182757725528	Mystery	14.00	USD \$
Catweazie	Richard Carpenter	150	120257725539	Fantasy	150	JPY ¥

UI para o usuário 'erin', com todos os features ativados.



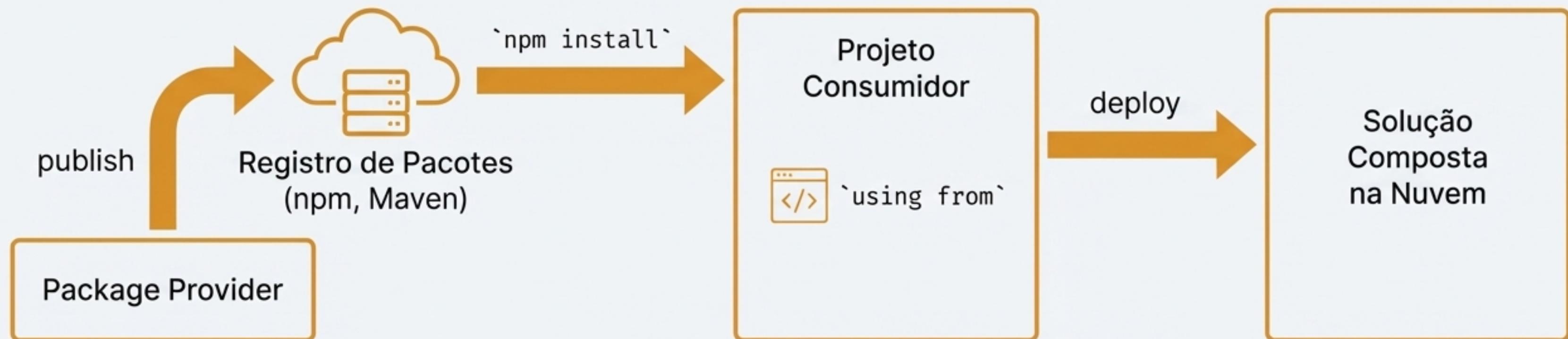
# Padrão 3: Reuso e Composição para ‘Construção de Sistemas’

## O Quê

Reuso e Composição é a prática de construir soluções aprimoradas ou verticalizadas importando conteúdo (modelos, código, dados, UIs) de pacotes reutilizáveis, baseando-se em gerenciadores de pacotes como `npm` e `Maven`.

## Por Quê

Acelerar o desenvolvimento, promover a consistência e permitir a criação de ecossistemas de software onde diferentes partes contribuem com componentes especializados.



# Cenários de Uso e o Princípio da “Ativação por Alcance”

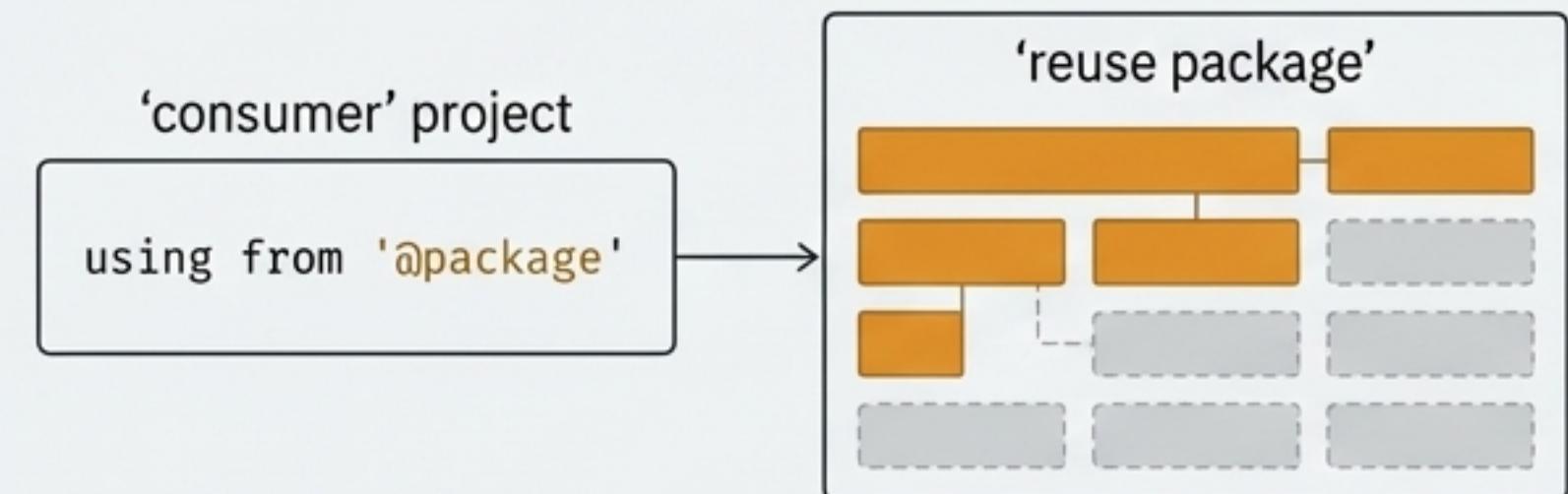
## Cinco Cenários de Uso Principais

1.  **Soluções Verticalizadas/Compostas:** Combinar múltiplos pacotes para criar uma nova solução.
2.  **Pacotes de Extensão Pré-construídos:** Oferecer aprimoramentos como um pacote reutilizável.
3.  **Pacotes de Integração Pré-construídos:** Encapsular integrações com sistemas de backend (ex: S/4HANA).
4.  **Pacotes de Dados de Negócio:** Fornecer dados iniciais (ex: Países, Moedas).
5.  **Customização de Soluções SaaS:** Clientes usando as mesmas técnicas para adaptar suas aplicações.

## Princípio Chave: Active by Reachability (Ativo por Alcance)

**"Tudo o que você referencia a partir de seus próprios modelos é servido. Tudo fora de seus modelos é ignorado."**

Mesmo que um pacote importado contenha dez serviços, apenas aqueles que você explicitamente importa com `using` em seu projeto serão ativados e implantados. Isso dá ao consumidor controle total sobre o escopo da aplicação final.



# Integrando Microsserviços Remotos vs. Embutindo Conteúdo

## A Configuração

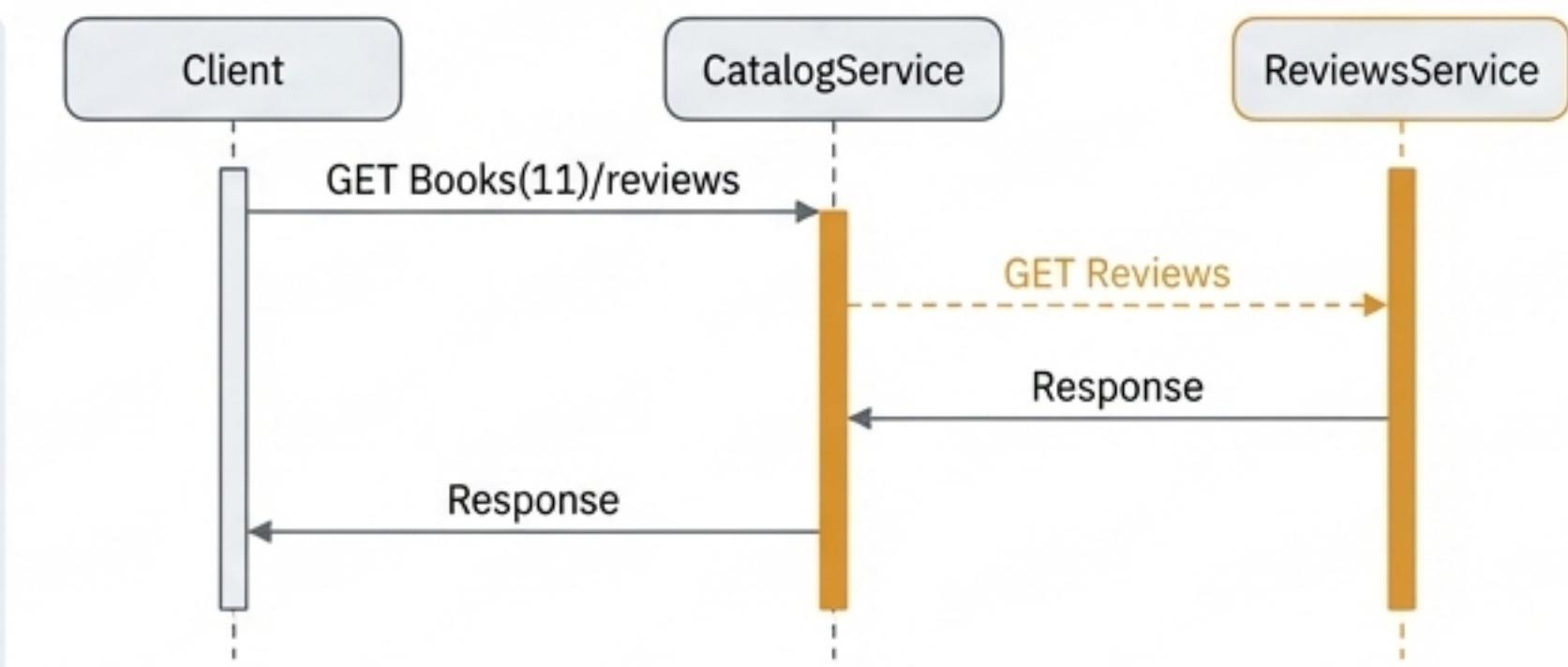
Em vez de apenas usar o conteúdo, o serviço é declarado como remoto no `package.json`.

```
"cds": {  
  "requires": {  
    "ReviewsService": {  
      "kind": "odata",  
      "model": "@capi.reviews",  
      "credentials": {  
        "destination": "reviews-api"  
      }  
    }  
  }  
}
```

## A Implementação

### mashup.js

```
const cds = require('@sap/cds')  
module.exports = async (srv) => {  
  const { ReviewsService } = cds.services  
  srv.on('READ', 'Books', async (req, next) => {  
    const items = await next()  
    if (!items) return items  
    // delegate to remote service  
    const reviews = await ReviewsService.run(  
      SELECT.from('Reviews').where({  
        book_ID: items.map((i) => i.ID),  
      })  
    )  
    // merge results  
    for (const item of items) {  
      item.reviews = reviews.filter((r) => r.book_ID === item.ID)  
    }  
    return items  
  })  
}
```



## Teste Simplificado com `cds watch`

Explicar que `cds watch` automaticamente cria mocks de serviços remotos durante o desenvolvimento. Ao rodar múltiplos serviços localmente, `cds watch` os descobre e cria bindings automaticamente através do arquivo `~/.cds-services.json`, permitindo testes de integração local sem configuração manual.

# O Toolkit Completo: Escolhendo a Ferramenta Certa

	Extensões SaaS 	Feature Toggles 	Reuso e Composição 
Metáfora	Ajuste Fino	Ativação Modular	Construção de Sistemas
Principal Usuário	Cliente / Assinante SaaS	Provedor SaaS	Parceiro / Provedor / Cliente Avançado
Cenário Ideal	Personalização específica do tenant (adicionar campos, entidades).	Oferecer funcionalidades opcionais ou edições de produto (premium/básico).	Criar soluções verticalizadas, compostas ou pacotes de extensão.
Mecanismo Central	CDS Aspects + cds pull` / `cds push`.	ModelProviderService + ativação dinâmica em runtime.	<code>npm install / mvn resolve</code> + diretivas `using from`.

# Construindo Além do Padrão

A extensibilidade no CAP não é um recurso adicional; é parte do seu DNA arquitetônico. Os **padrões** de Ajuste Fino, Ativação Modular e Construção de Sistemas **fornecem um kit de ferramentas completo** para criar aplicações que evoluem com as necessidades do negócio. Domine essas ferramentas para projetar soluções que não são apenas funcionais, mas verdadeiramente adaptáveis.

