

A Arquitetura Orientada a Eventos com CAP: Do Monolito aos Microsserviços, Sem Alterar o Código

Como o princípio de 'eventos ubíquos' do CAP permite criar aplicações desacopladas, resilientes e prontas para a nuvem.

O Princípio Fundamental do CAP: Tudo é um Evento

No CAP, tudo o que acontece em tempo de execução é uma resposta a eventos.

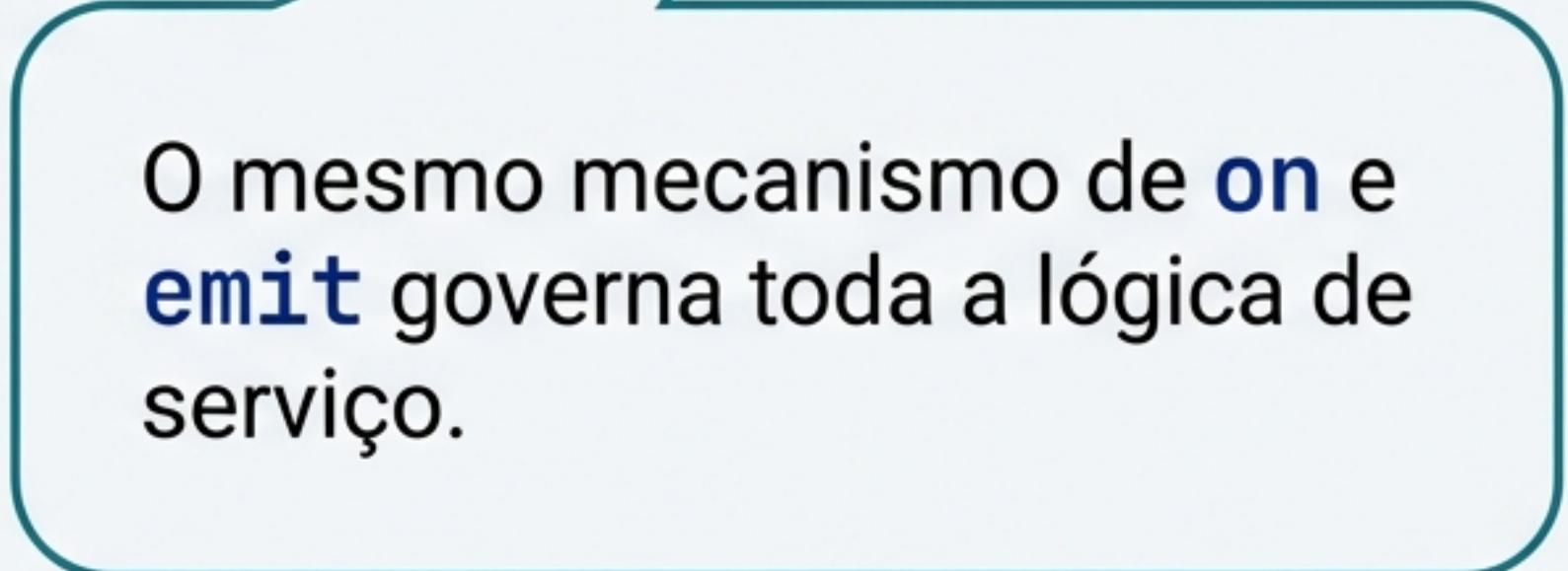
Todas as implementações de serviço ocorrem em **event handlers**.

Os serviços podem emitir eventos, recebê-los e reagir a eles de forma intrínseca.

Este modelo unificado simplifica drasticamente o desenvolvimento, tratando tanto requisições síncronas quanto mensagens assíncronas com a mesma mecânica.

```
// O núcleo do modelo de processamento do CAP
let srv = new cds.Service

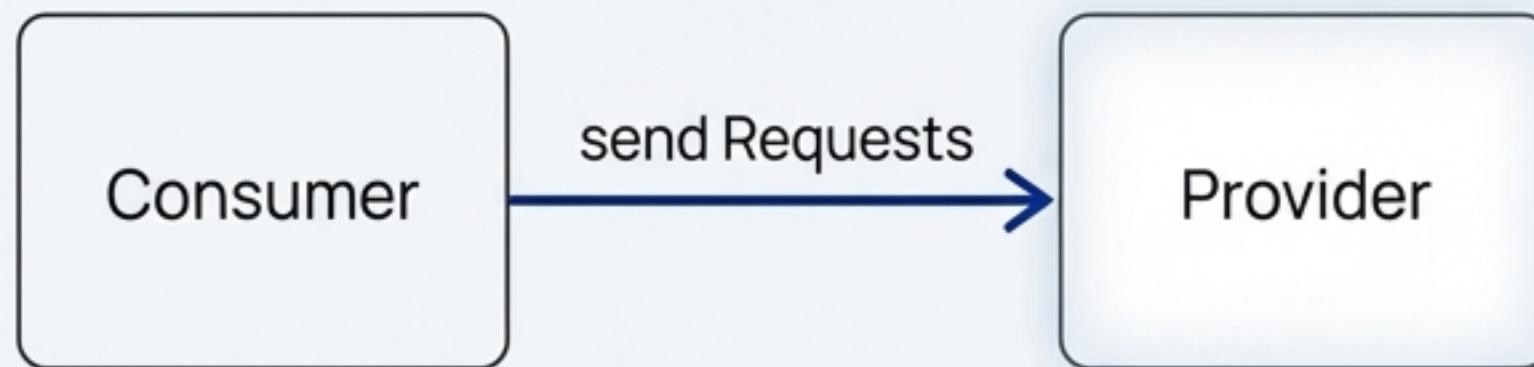
// Recebendo Eventos (Listeners)
srv.on ('some_event', msg => console.log('Listener 1 recebeu:', msg))
srv.on ('some_event', msg => console.log('Listener 2 recebeu:', msg))
// Emitindo Eventos
await srv.emit ('some_event', { foo:'11', bar:'12' })
```



O mesmo mecanismo de **on** e **emit** governa toda a lógica de serviço.

APIs Síncronas e Assíncronas: Duas Faces da Mesma Moeda

Comunicação Síncrona (Requests)



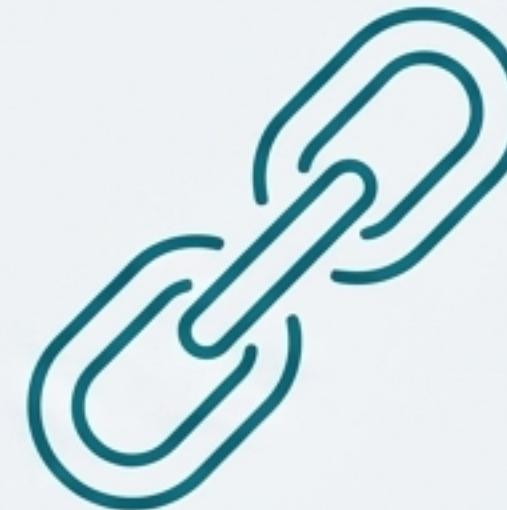
O Consumidor conhece e se conecta ativamente ao Provedor para enviar uma requisição e aguardar uma resposta. A **iniciativa é do consumidor**. Os *handlers* atuam como interceptadores (apenas o primeiro é chamado).

Comunicação Assíncrona (Events)



O Emissor não conhece os Receptores. Ele simplesmente emite um evento. Os Receptores se inscrevem para ouvir esses eventos. A **iniciativa é invertida**. Os *handlers* atuam como *listeners* (todos são chamados).

Por Que Usar Mensageria? Resiliência e Desacoplamento



Resiliência

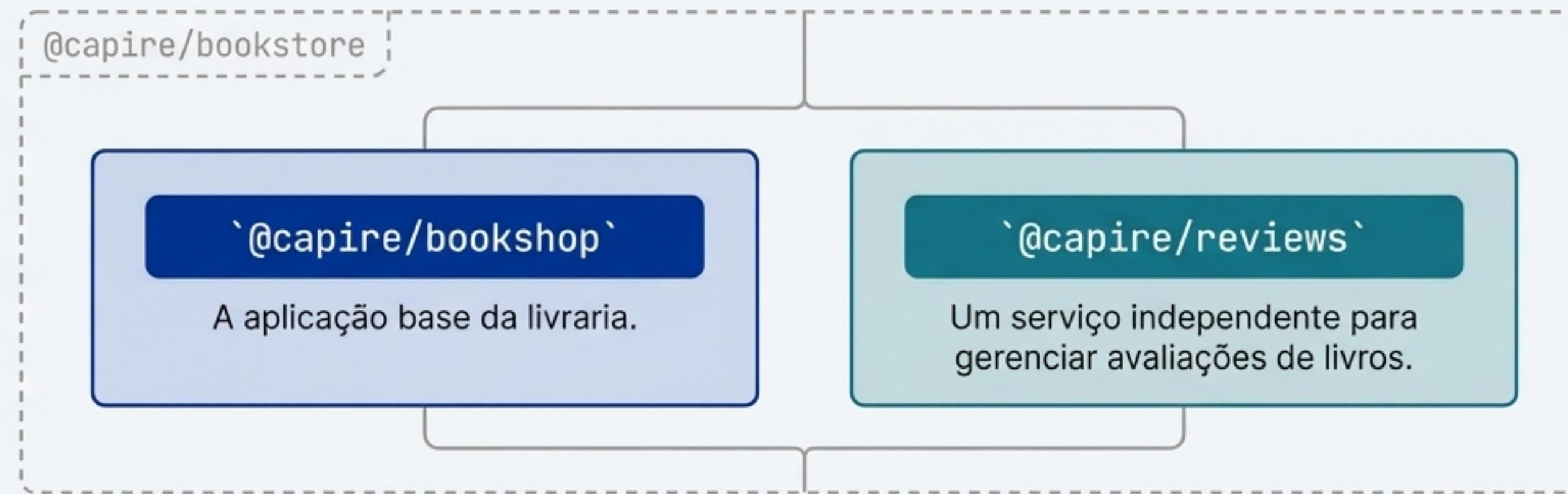
Se um serviço receptor ficar offline, as mensagens de evento são armazenadas com segurança em uma fila. A entrega é garantida assim que o serviço voltar a operar. Sem perda de dados, sem falhas em cascata.

Desacoplamento

Emissores de eventos não precisam conhecer seus receptores. Isso permite que um serviço **emita eventos** que podem ser consumidos por outros serviços no futuro, **criando pontos de extensão** e permitindo que as equipes **trabalhem de forma independente**.

Nossa Aplicação Exemplo: Book Reviews

Vamos acompanhar a evolução de uma aplicação composta que combina uma livraria com um serviço de avaliações.



O Objetivo: Quando uma nova avaliação é adicionada no serviço **@capire/reviews**, o serviço **@capire/bookshop** deve ser notificado para atualizar a média de avaliação do livro correspondente.

Declarando Eventos no Nível Conceitual com CDS

Em CAP, os eventos são parte da API do serviço, definidos diretamente no modelo CDS. Focamos no domínio, não em protocolos de baixo nível.

`event AverageRatings.Changed` declara a API assíncrona do serviço. Qualquer alteração na avaliação de um livro emitirá este evento.

```
// in @capire/reviews
service ReviewsService @(path:'/reviews/api') {
    /**
     * Summary of average ratings per subject.
     */
    @readonly entity AverageRatings as projection on Reviews {
        key subject,
        round(avg(rating),2) as rating : Rating,
        count(*) as reviews : Integer
    } group by subject;

    /**
     * Inform about changes to a subject's average rating
     */
    event AverageRatings.Changed : AverageRatings;
}
```

O Código em Ação: Emitindo e Recebendo Eventos

A implementação é direta e agnóstica à topologia. O emissor simplesmente informa sobre um fato ocorrido; o receptor se inscreve para reagir a esse fato.

Emitindo o Evento (`ReviewsService`)

```
// Emite o evento para notificar potenciais listeners  
// srv.emit() informa o framework, que cuida do resto.  
this.after(['CREATE', 'UPDATE', 'DELETE'], 'Reviews', async (_,  
req) => {  
  // ...cálculo da nova média...  
  return this.emit('AverageRatings.Changed', { subject, rating,  
})
```

Um diagrama de fluxo mostra uma seta apontando de cima para baixo dentro de um retângulo contendo o código. No final da seta, há um ponto azul.

`this.emit()` emite um evento para o próprio serviço. Ele não sabe (e não se importa) quem está ouvindo.

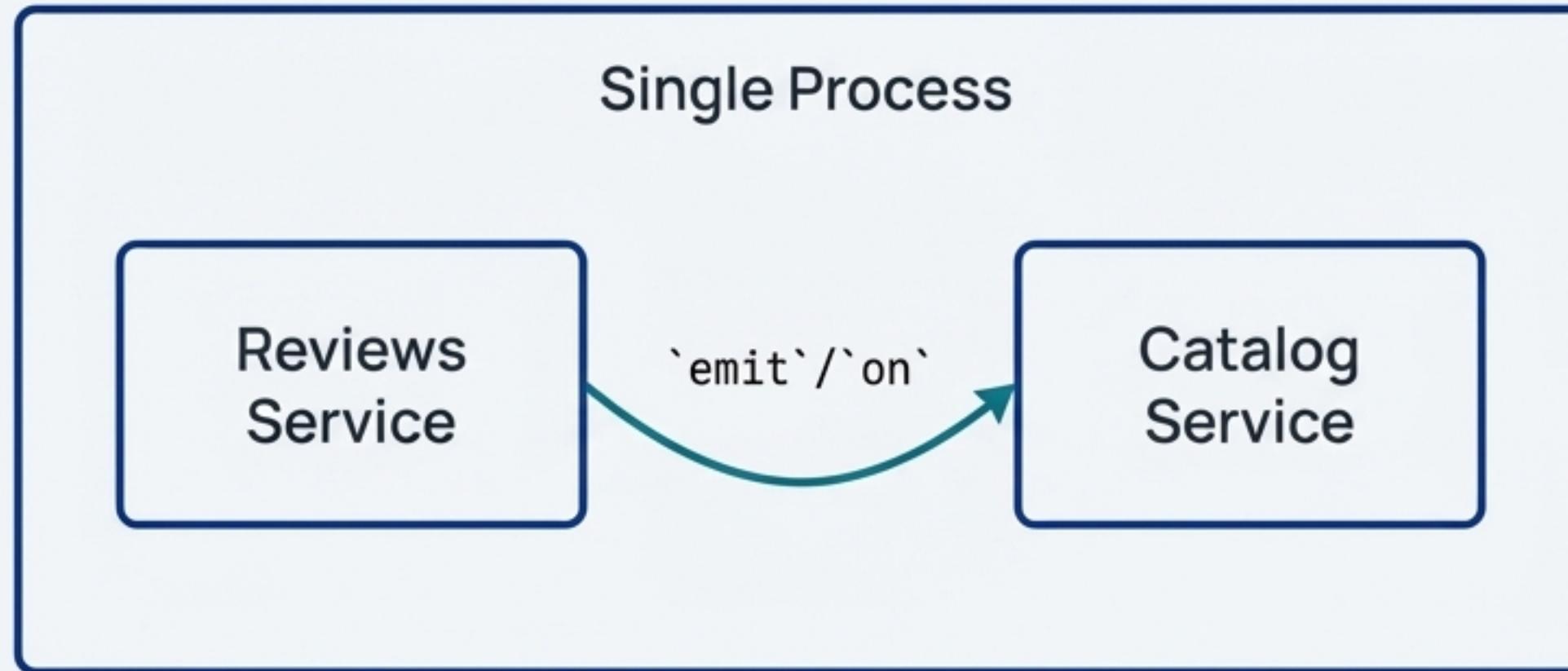
Recebendo o Evento (`Bookstore`)

```
// Conecta-se ao serviço remoto e registra um handler  
const ReviewsService = await cds.connect.to('ReviewsService')  
  
ReviewsService.on('AverageRatings.Changed', (msg) => {  
  console.debug('> received:', msg.event, msg.data)  
  // ...lógica para atualizar a entidade Books...  
})
```

Um diagrama de fluxo mostra uma seta apontando de cima para baixo dentro de um retângulo contendo o código. No final da seta, há um ponto azul.

`ReviewsService.on()` se inscreve para o evento. O payload está em `msg.data`.

Cenário 1: Eventos In-Process (A Aplicação Monolítica)



Quando o emissor e o receptor rodam no mesmo processo, nada mais é necessário. A troca de eventos é uma característica intrínseca do runtime do CAP.

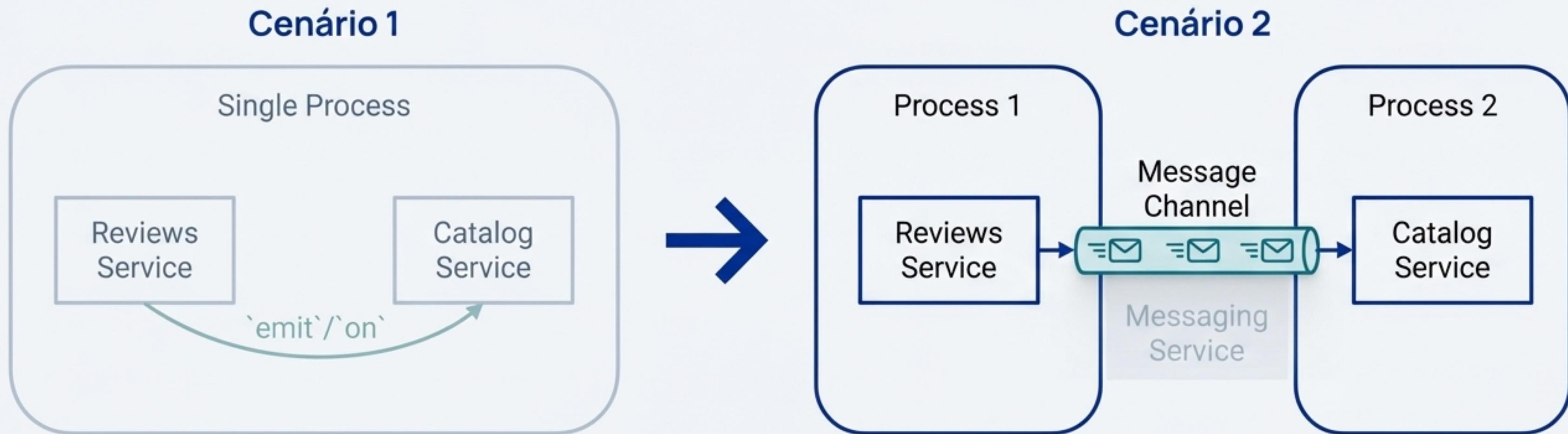
```
`cds watch bookstore`
```

```
[...]
[cds] - mocking ReviewsService { path: '/reviews', ... }
[cds] - serving CatalogService { path: '/browse', ... }
[cds] - server listening on { url: 'http://localhost:4004' }
```

Observe que o `ReviewsService` é "mocked" e servido no mesmo processo.

Cenário 2: A Mudança para Microsserviços

E se movermos o “Reviews Service” para um processo separado? A comunicação direta in-process não é mais possível. Precisamos de um canal de mensagens para transportar os eventos.



A lógica de negócio nos serviços permanece a mesma. A mudança ocorre na infraestrutura.

A Mágica do CAP: Nenhuma Alteração no Código. Apenas Configuração.

O código de implementação (`srv.emit` e `srv.on`) que escrevemos anteriormente permanece idêntico.

Para habilitar a comunicação distribuída, simplesmente configuramos um serviço de mensageria no `package.json`. Para desenvolvimento, o CAP oferece uma implementação simples baseada em arquivos.

```
"cds": {  
  "requires": {  
    "messaging": {  
      "[development)": { "kind": "file-based-messaging" }  
    }  
  }  
}
```

Com essa configuração, o CAP automaticamente utiliza um canal de mensagens em vez da comunicação in-process. Zero impacto no código do serviço.

Microserviços em Ação

1. Em um terminal, inicie o serviço de avaliações:

```
`cds watch reviews`
```

2. Em outro terminal, inicie a livraria:

```
`cds watch bookstore`
```

`reviews` (Emissor)

```
[cds] - connect to messaging > file-based-messaging  
[cds] - serving ReviewsService...  
...  
< emitting: AverageRatings.Changed { subject: '251', ... }
```

`bookstore` (Receptor)

```
[cds] - connect to messaging > file-based-messaging  
[cds] - connect to ReviewsService > odata http://localhost:4005  
...  
> received: AverageRatings.Changed { subject: '251', ... }
```

Conclusão: Sem tocar em uma linha de código, o evento emitido pelo `ReviewsService` foi transportado pelo canal `file-based-messaging` e recebido pelo `bookstore`, exatamente como antes.

O Resultado Final: Resiliência por Design

- 1. Desligue o Receptor:** Pare o servidor `bookstore` (`Ctrl+C`).
- 2. Continue Emitindo:** Adicione ou atualize mais avaliações na UI do `ReviewsService`. Os eventos são emitidos, mas não há ninguém para ouvi-los... ainda.
- 3. Religue o Receptor:** Reinicie o servidor `bookstore` com `cds watch bookstore`.

Terminal `bookstore` ao reiniciar

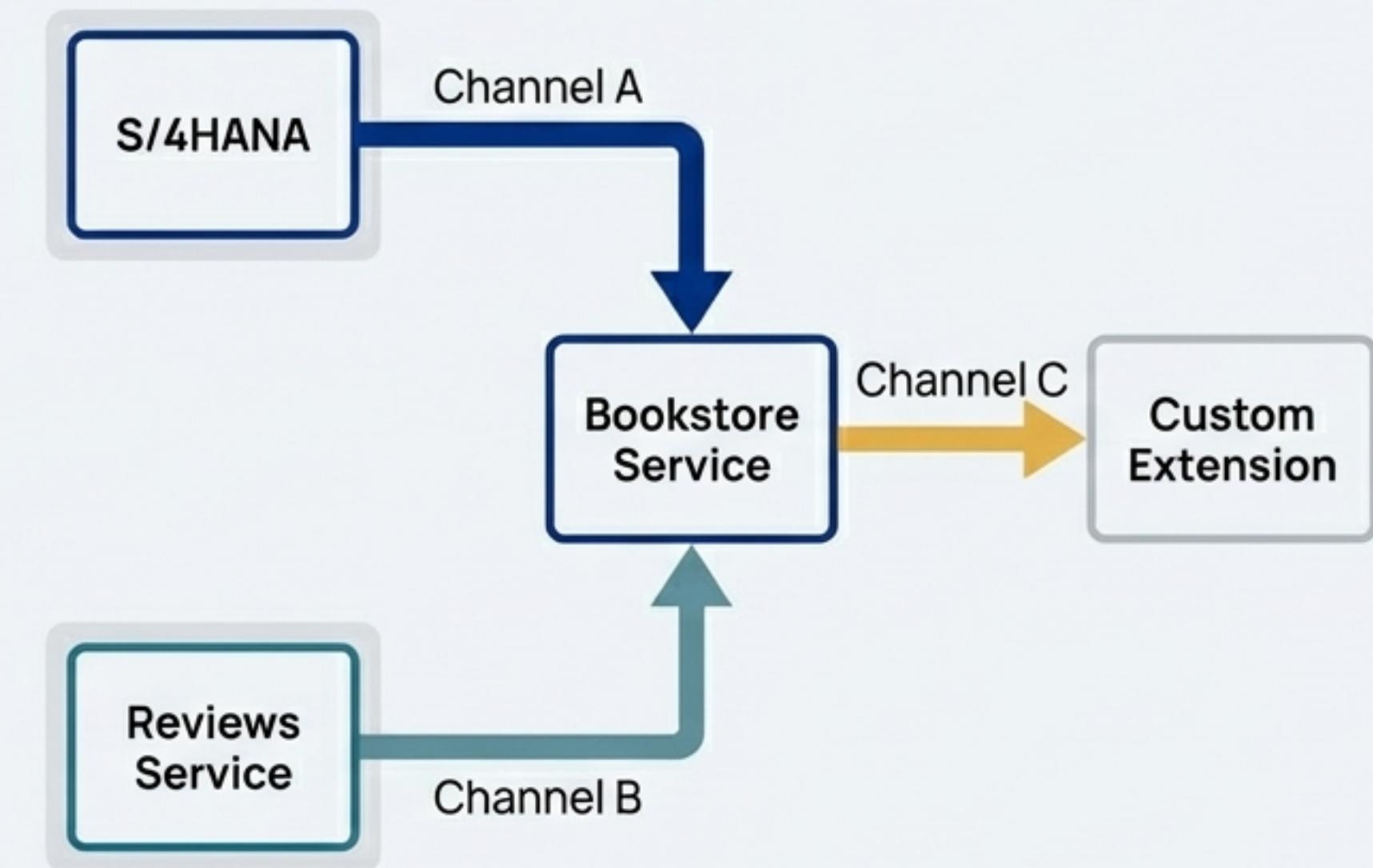
```
[cds] - server listening on { url: 'http://localhost:4004' }
[cds] - launched at ...
> received: AverageRatings.Changed { subject: '207', ... }
> received: AverageRatings.Changed { subject: '201', ... }
> received: AverageRatings.Changed { subject: '251', ... }
```

Todas as mensagens emitidas enquanto o receptor estava offline permaneceram na fila e foram entregues assim que o serviço voltou. A resiliência é um comportamento emergente do design, não um esforço de implementação.

Escalando para a Empresa: Gerenciando Múltiplos Canais de Mensagens

Em uma arquitetura real, você pode precisar de canais separados para diferentes fluxos de eventos. Por exemplo:

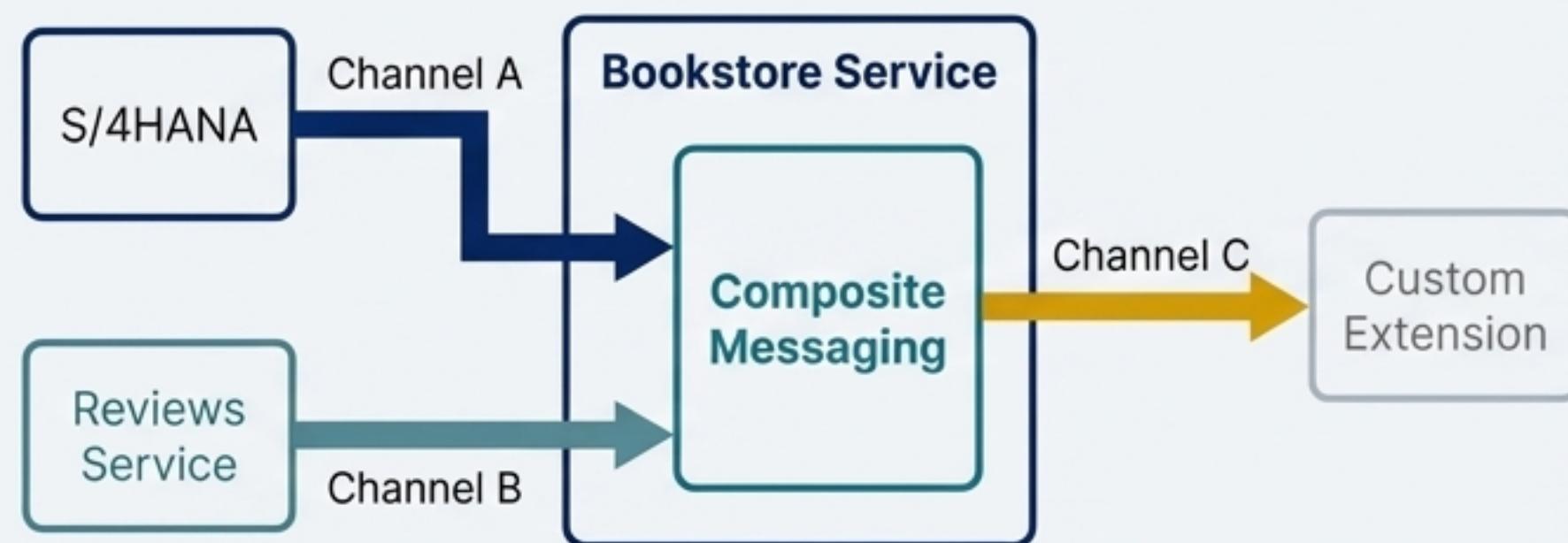
- Um canal para receber eventos do SAP S/4HANA.
- Um canal para eventos internos entre seus microsserviços.
- Um canal para eventos públicos para extensões de clientes.



A Pergunta: Como gerenciar essa topologia sem cair para uma implementação de baixo nível e perder os benefícios da abstração? A **Mangunta:** Manrope Medium.

A Solução CAP: Roteamento Flexível com `composite-messaging`

O CAP fornece a implementação `composite-messaging`, que atua como um despachante transparente. Você define rotas para diferentes eventos, direcionando-os para os canais corretos, tudo via configuração.



```
"messaging": {  
    "kind": "composite-messaging",  
    "routes": {  
        // Eventos do S/4HANA vão para o ChannelB  
        "ChannelB": ["**/sap/s4/**"],  
        // Eventos do ReviewsService vão para o ChannelA  
        "ChannelA": ["**/ReviewsService/*"],  
        // Eventos emitidos pelo bookshop vão para o ChannelC  
        "ChannelC": ["**/bookshop/**"]  
    },  
    "ChannelA": { "kind": "enterprise-messaging", ... },  
    "ChannelB": { "kind": "enterprise-messaging", ... },  
    "ChannelC": { ... }  
}
```

Benefício: Topologias de mensagens podem ser alteradas em tempo de implantação, sem tocar no código-fonte ou nos modelos.

A Vantagem do CAP: Mensageria Conceitual vs. Baixo Nível

Demonstramos uma jornada completa, do monolito aos microsserviços, sem alterar a lógica de negócio. Isso é possível porque o CAP opera em um nível conceitual.

Mensageria Conceitual (O Caminho CAP)	Mensageria de Baixo Nível
<ul style="list-style-type: none">✓ Nomes de eventos locais ao serviço (`'AverageRatings.Changed`)✓ Declarações de eventos no modelo CDS✓ APIs tipadas e geradas automaticamente✓ Flexibilidade para rodar in-process ou distribuído com zero mudança de código✓ Abstração de protocolos (ex: CloudEvents) e brokers	<ul style="list-style-type: none">✗ Nomes de eventos completos e frágeis (`'my.namespace.ReviewsService.Changed`)✗ Acoplamento com a implementação do broker de mensagens✗ Sem declaração formal no modelo✗ Perda da flexibilidade de topologia

Recomendação Final: Sempre prefira a API de nível conceitual. Ela permite que você se concentre na lógica de domínio e evolua sua arquitetura de forma flexível e robusta.