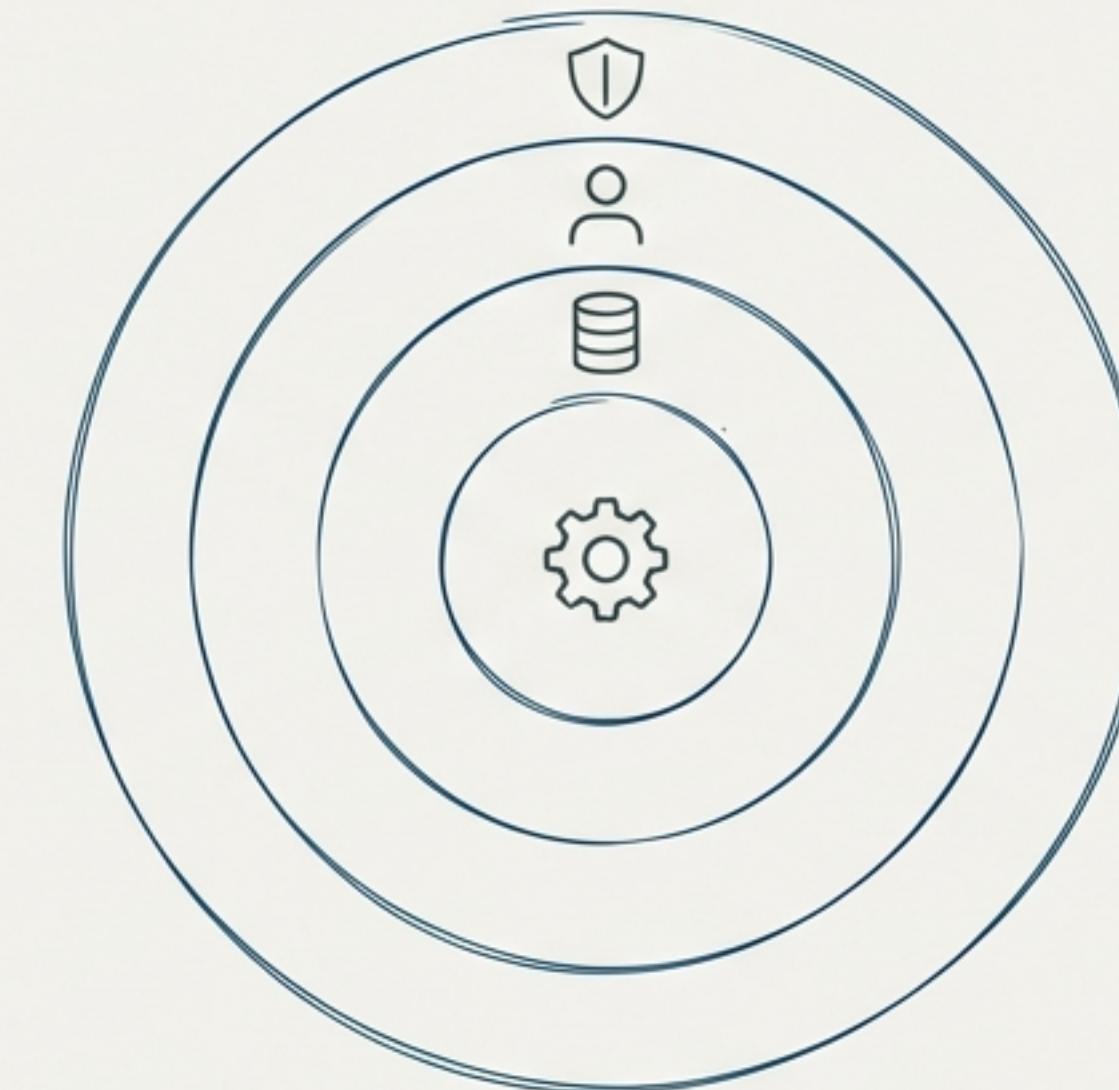


# Da Lógica à Paisagem: Dominando o Runtime Java do SAP CAP



Um Guia Visual para Desenvolvedores sobre Event  
Handlers, Contextos e Transações.

# A Jornada de um Desenvolvedor: Construindo um Modelo Mental do Runtime

Para dominar o CAP Java, não basta conhecer as APIs. É preciso entender como elas se conectam. Esta apresentação desmonta o runtime, camada por camada, começando pelo código que você escreve até o ambiente transacional que o protege.

Seguiremos esta estrutura, construindo o diagrama passo a passo para revelar a anatomia completa de uma requisição CAP.

Ambiente da Requisição ('`RequestContext`')

Contexto do Evento e Erros ('`EventContext`', '`Error Handling`')

Lógica de Negócio ('`EventHandler`')

Garantia Transacional ('`ChangeSetContext`')

# Implementando Lógica Customizada com `Event Handlers`

No CAP, tudo é um evento. `Event Handlers` são métodos Java que permitem estender ou sobrescrever o processamento desses eventos. Eles são o principal mecanismo para adicionar sua lógica de negócios customizada.

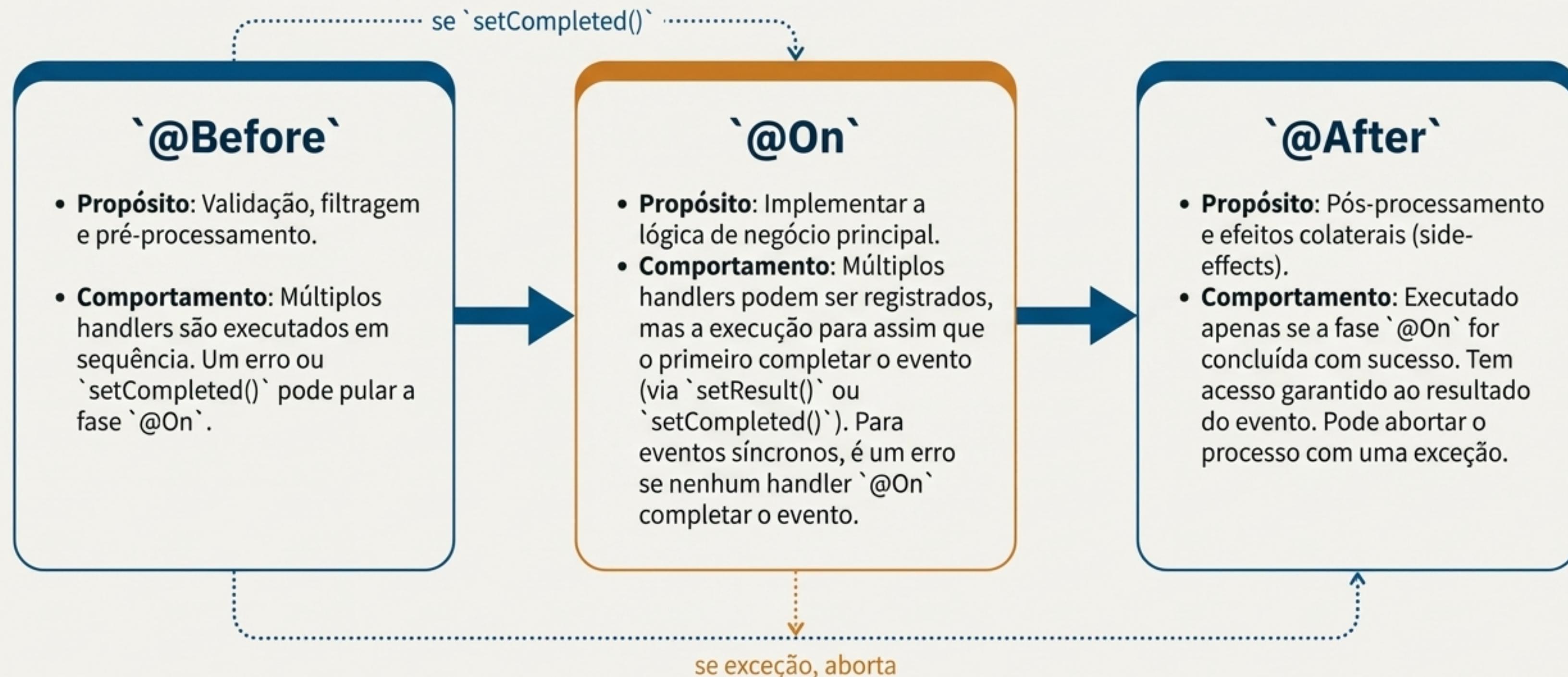
- Manipular eventos CRUD (CREATE, READ, UPDATE, DELETE).
- Implementar `actions` e `functions` definidas no modelo CDS.
- Processar eventos assíncronos de serviços de mensageria.



## PRO TIP

**Você sabia?** A maioria das funcionalidades nativas do CAP, como autorização e validações, são implementadas usando `Event Handlers`.

# O Ciclo de Vida de um Evento: As Fases `@Before`, `@On` e `@After`



# Estruturando seus Handlers: Classes e Anotações

```
@Component  
@ServiceName(AdminService_.CDS_NAME)  
public class AdminServiceHandler implements EventHandler {  
  
    @Before(event = CqnService.EVENT_CREATE, entity = Books_.CDS_NAME)  
    public void validateNewBook(Books book) {  
        // ... sua lógica de validação ...  
    }  
  
    @On(event = "myCustomAction")  
    public void onMyCustomAction(MyCustomActionContext context) {  
        // ... sua lógica principal ...  
        context.setCompleted();  
    }  
  
    @After(event = CqnService.EVENT_READ, entity = Books_.CDS_NAME)  
    @HandlerOrder(HandlerOrder.LATE)  
    public void enrichReadBooks(List<Books> books) {  
        // ... enriquecer dados após a leitura ...  
    }  
}
```

**@Component:** Torna a classe um bean gerenciado pelo Spring.

**implements EventHandler:** Interface de marcação para que o CAP identifique a classe.

**@ServiceName:** Define o serviço padrão para todos os handlers na classe. Pode ser sobrescrito em cada método.

**@Before, @On, @After:** Anotações que definem a fase e o alvo do handler, com atributos como:

- **event:** O(s) evento(s) alvo.
- **entity:** A(s) entidade(s) alvo.

**@HandlerOrder:** Controla a ordem de execução dos handlers dentro da mesma fase ('EARLY', 'LATE').

## SEÇÃO 2: INTERAÇÃO E REAÇÃO

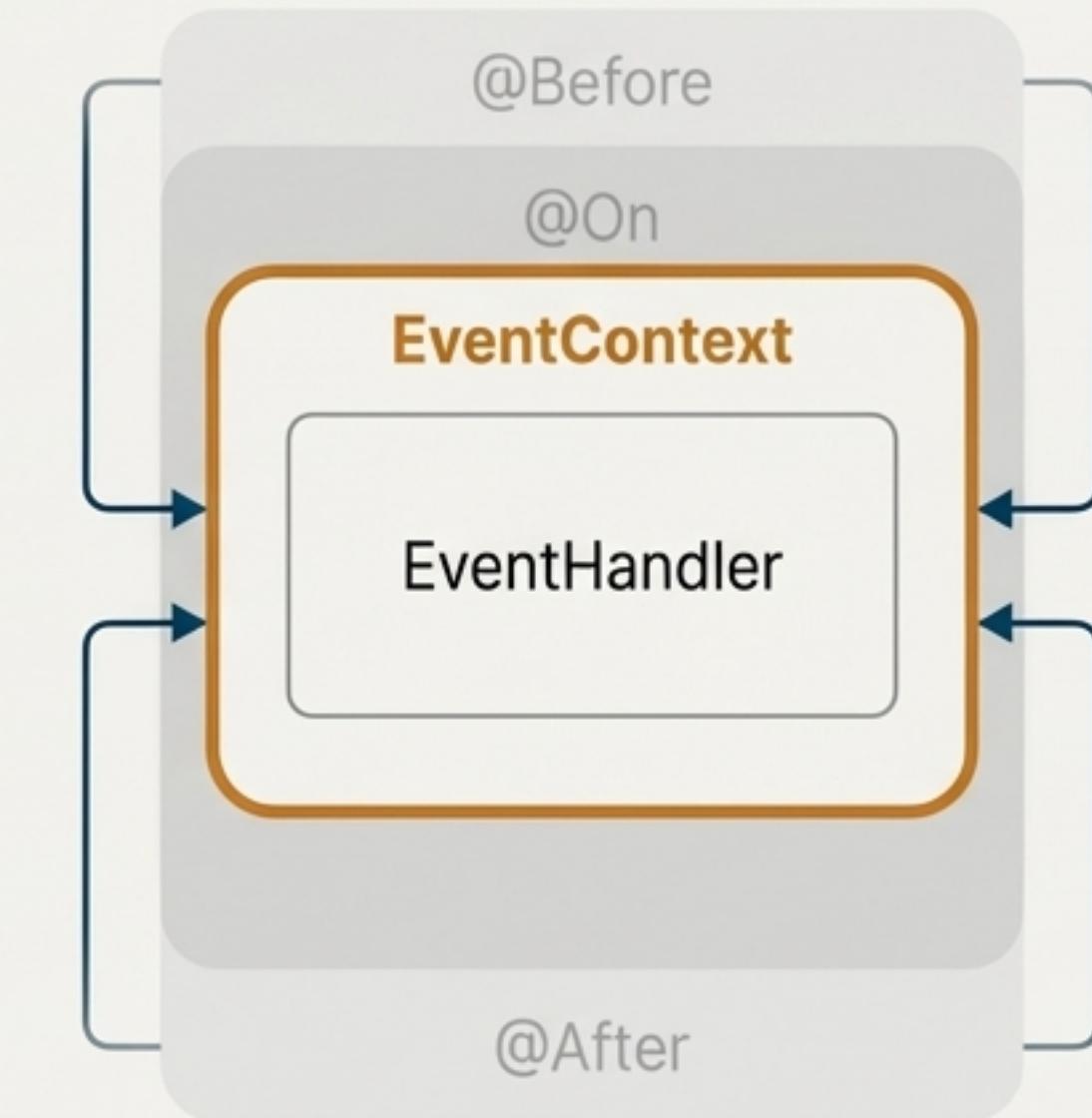
# Acessando Dados do Evento com o `EventContext`

**Problema:** O `EventContext` genérico usa `get("key")` e `put("key", value)`, o que não é type-safe e propenso a erros.

```
// Não ideal: propenso a erros de digitação e casting
Object cqn = context.get("cqn");
context.put("result", myResult);
```

**Solução:** Interfaces de contexto específicas para cada evento, que oferecem getters e setters tipados.

```
// Ideal: type-safe e com autocompletar da IDE
CdsReadEventContext readContext = context.as(CdsReadEventContext.class);
CqnSelect select = readContext.getCqn();
readContext.setResult(result);
```



### MELHOR PRÁTICA

Sempre use as interfaces de `EventContext` específicas do evento (`CdsReadEventContext`, `CdsCreateEventContext`, etc.) ou as geradas pelo Maven Plugin para suas actions/functions. O método `as()` é seu melhor amigo.

# Assinaturas de Métodos Inteligentes: Injeção Direta de Dados

O CAP Java simplifica seu código ao permitir que você declare o que precisa diretamente como argumentos do método. O framework infere o contexto e fornece os dados corretos.

## Acessando Dados da Entidade

O CAP extrai os dados do `CqnStatement` (em `@Before`/`@On`) ou do `Result` (em `@After`).

```
@Before(event = "CREATE")
public void changeBooks(List<Books> books) { /* ... */ }
```

## Acessando Referências da Entidade

Obtém uma referência para construir queries type-safe.

```
@After(event = "UPDATE")
public void changedBook(Books_ ref) {
    var select = Select.from(ref).columns(b -> b.title());
}
```

## Acessando o Contexto Tipado

Infere o evento a partir do tipo do contexto.

```
@On(entity = Books_.CDS_NAME)
public void readBooks(CdsReadEventContext context) { /* ... */ }
```

## Acessando o Serviço Atual

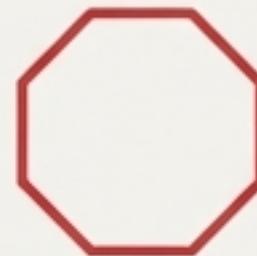
Injeta a instância do serviço que está processando o evento.

```
@After
public void myHandler(AdminService service) { /* ... */ }
```

# Lidando com Falhas: Exceções vs. a API de Mensagens

O CAP Java oferece duas maneiras distintas de indicar erros, cada uma com um impacto diferente no fluxo da requisição.

## Lançar uma `ServiceException`



- Aborta o processamento do evento imediatamente.
- Causa rollback da transação ativa.

### Uso Ideal

Erros fatais e irrecuperáveis que devem interromper toda a operação (ex: violação de regra de negócio crítica).

## Usar a API `Messages`



- Adiciona mensagens (erro, aviso, info) ao contexto da requisição, mas \*não\* interrompe o fluxo.
- Não afeta a transação diretamente.

### Uso Ideal

Coletar múltiplos erros de validação antes de, opcionalmente, abortar a requisição com `messages.throwIfError()`.

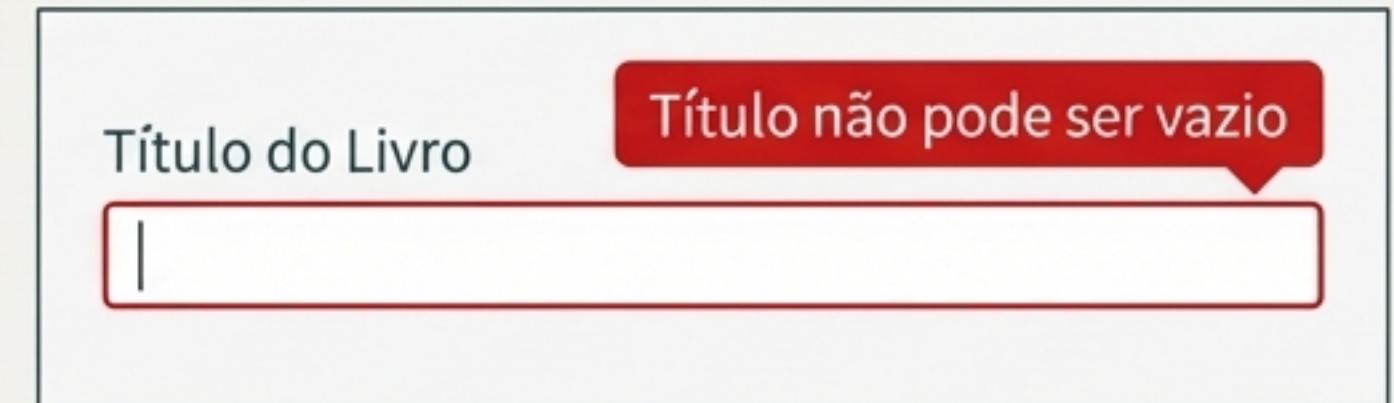
# Interrompendo o Fluxo com `ServiceException`

Use `ServiceException` para erros que devem parar o processamento. É a forma recomendada de sinalizar uma falha em um handler.

```
// Especificando um status HTTP e mensagem  
throw new ServiceException(ErrorStatuses.CONFLICT, "Estoque insuficiente para o livro '{}'",  
    bookTitle);
```

## `messageTarget`

Para melhorar a experiência do usuário em UIs como Fiori, direcione a mensagem de erro para o campo exato que a causou.



```
// Usando a API tipada para segurança e clareza  
throw new ServiceException(ErrorStatuses.BAD_REQUEST, "Título não pode ser vazio")  
    .messageTarget(Books_.class, b -> b.title());
```

O OData Adapter converte `ServiceException` em uma resposta de erro OData, com a mensagem principal e as mensagens coletadas na API `Messages` na seção de detalhes.

# Quem Está Requisitando? O `RequestContext`

## Necessidade

Uma única requisição HTTP pode disparar múltiplos eventos em diferentes serviços (ex: `ApplicationService`, `PersistenceService`). Todos precisam de acesso a informações compartilhadas como usuário, tenant, locale e headers.

## Solução

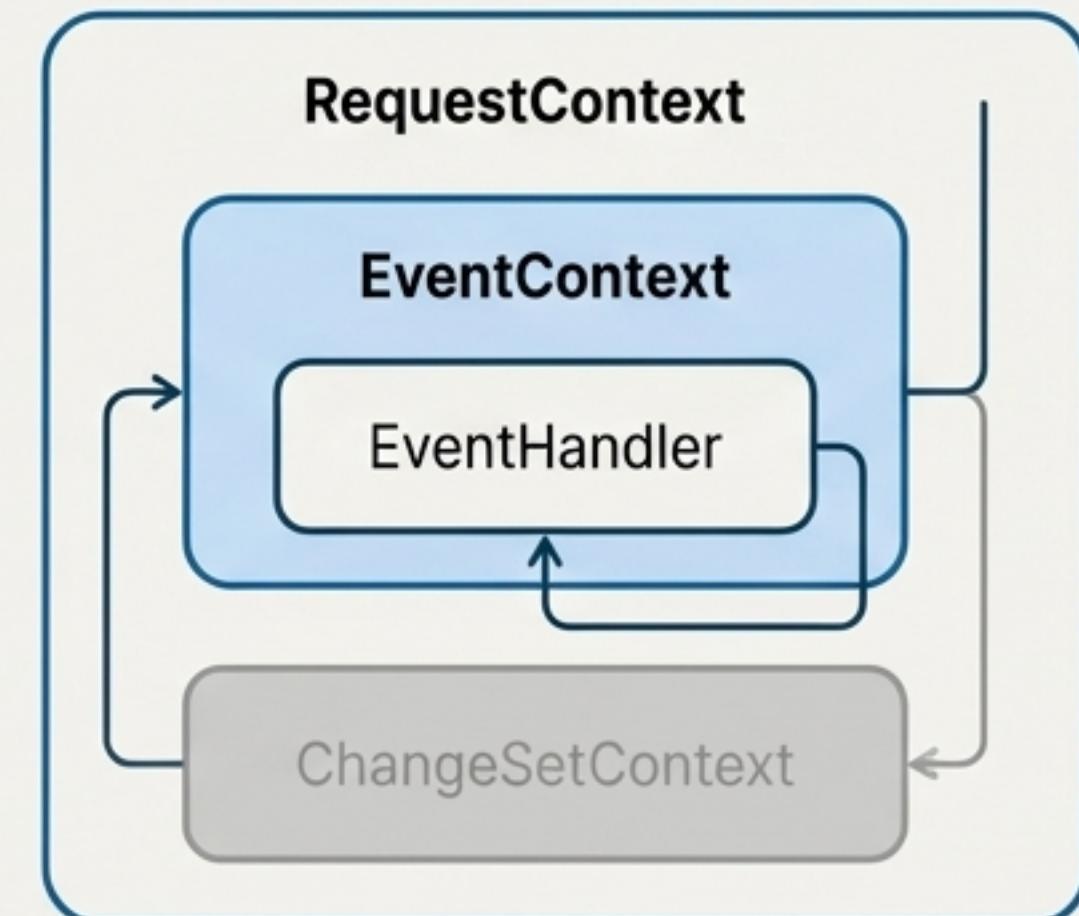
O `RequestContext` armazena essas informações e está disponível como um `thread-local` durante todo o ciclo de vida da requisição.

## Componentes Principais

-  **`UserInfo`**: Dados do usuário autenticado (ID, nome, roles, tenant).
-  **`ParameterInfo`**: Dados da requisição (headers, query parameters, locale).
-  **`AuthenticationInfo`**: Detalhes da autenticação (ex: JWT token).
-  **`FeatureTogglesInfo`**: Acesso a feature toggles ativos.

## Acesso

Pode ser acessado via `eventContext.getRequestContext()` ou injetado diretamente com `@Autowired` em beans Spring.



# Trocando de Identidade: Gerenciando Contextos com `RequestContextRunner`

Em cenários avançados, pode ser necessário executar uma parte do código com um contexto diferente. Por exemplo, como um usuário técnico para chamar um serviço interno ou para processar dados de um tenant específico em um job.

## runtime.requestContext()

### Cenários e Código

#### Executar como Usuário Técnico (mantendo o tenant)

Útil para rebaixar privilégios de um usuário nominal.

```
runtime.requestContext().systemUser().run(reqCtx -> { /* ... */ });
```

#### Executar como Usuário Técnico do Tenant Provedor

Para acessar dados compartilhados ou de configuração.

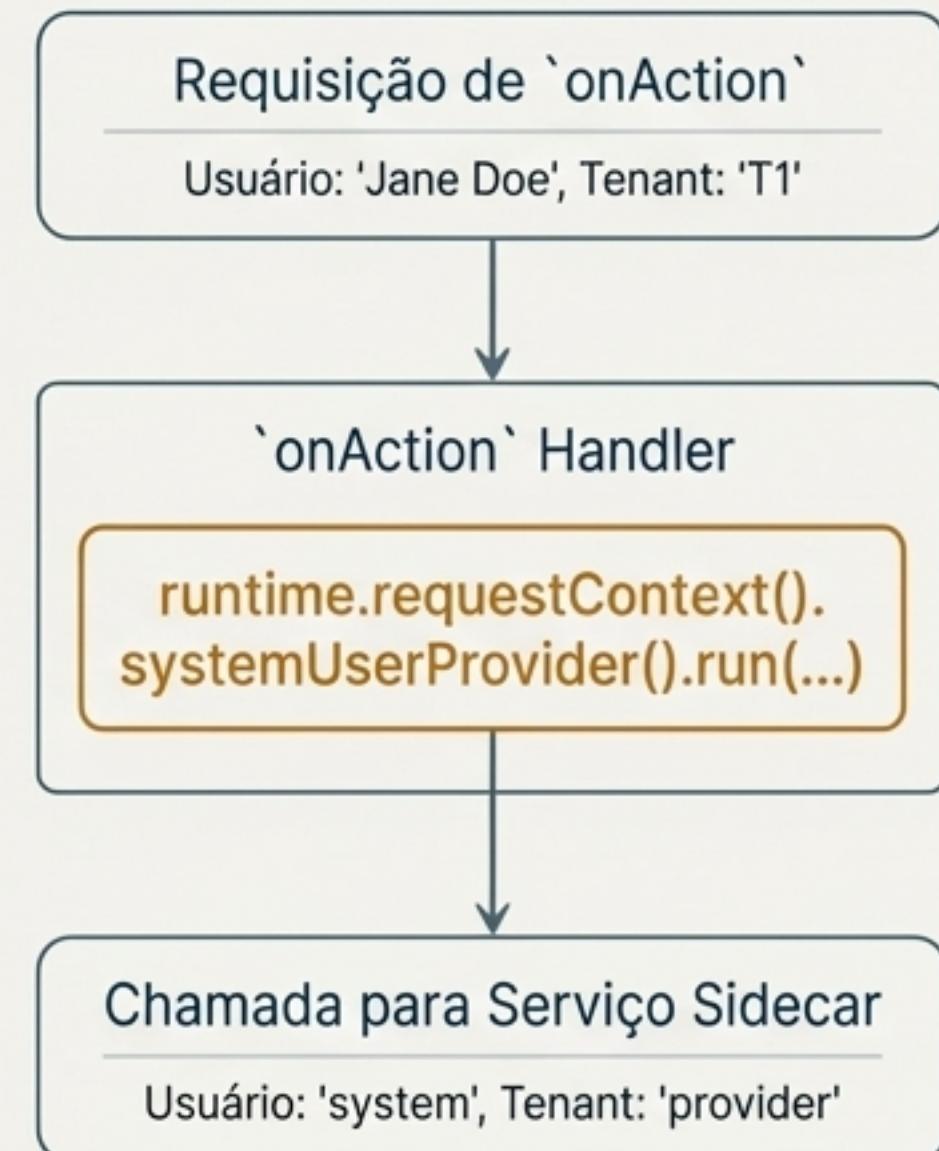
```
runtime.requestContext().systemUserProvider().run(reqCtx -> { /* ... */ });
```

#### Executar em um Tenant Específico

Essencial para background jobs em aplicações multitenant.

```
runtime.requestContext().systemUser("tenant-id").run(reqCtx -> { /* ... */ });
```

### Fluxo Visual



## SEÇÃO 4: A FUNDAÇÃO

# A Rede de Segurança: O `ChangeSetContext`

### Conceito

‘ChangeSetContext’ é a abstração leve do CAP para transações. Ele define os limites transacionais, mas delega a gestão da transação real (ex: JDBC) para um transaction manager.

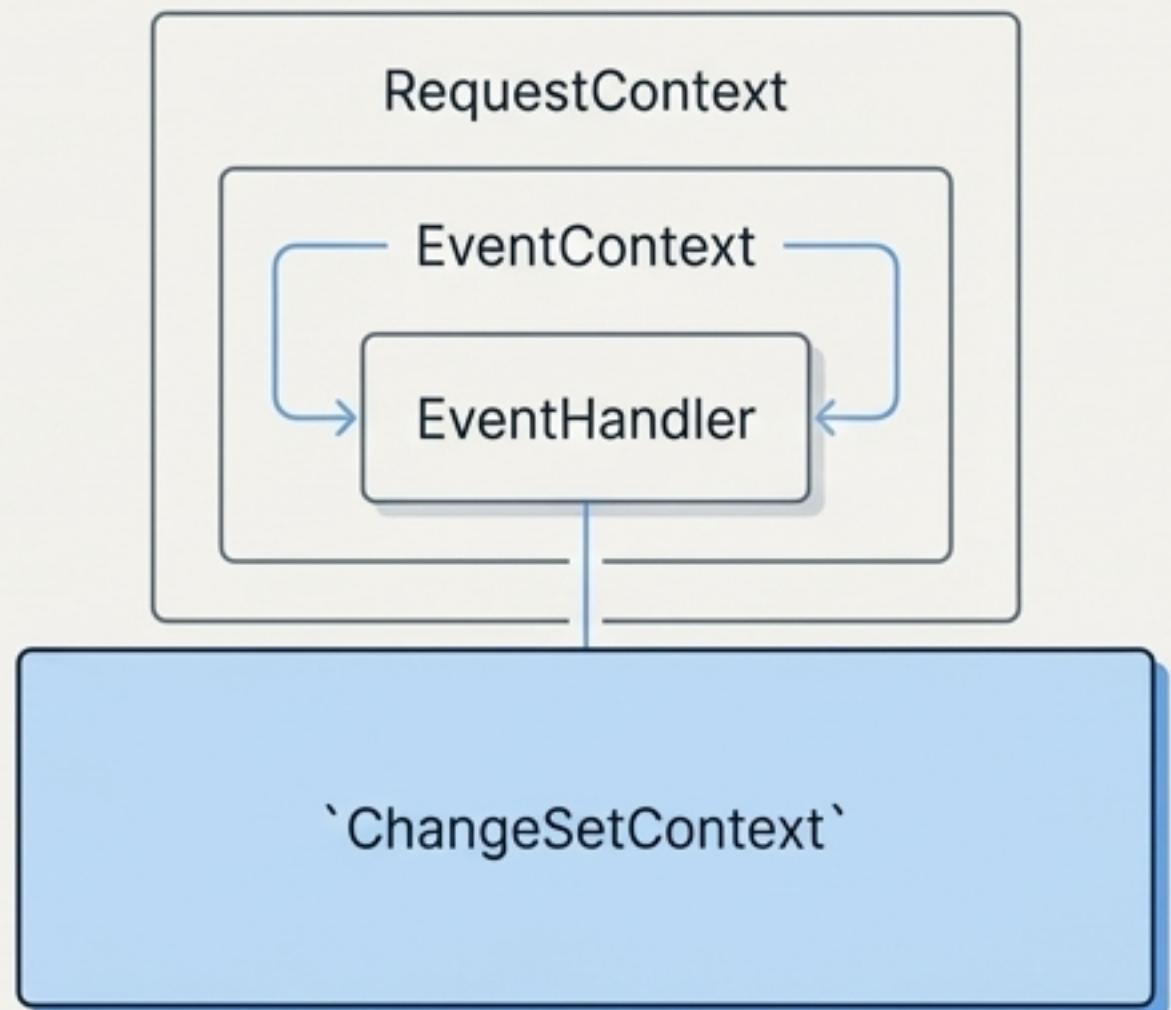
### Comportamento Padrão

- Por padrão, um ‘ChangeSetContext’ é aberto ao redor do evento mais externo de uma requisição.
- Isso garante que todas as operações dentro daquele evento (incluindo múltiplas chamadas ao ‘PersistenceService’) ocorram em uma única transação.
- A transação é iniciada de forma *lazy* na primeira interação com o banco de dados.

### API Explícita

Embora geralmente automático, você pode criar um ‘ChangeSetContext’ manualmente para um controle mais granular.

```
runtime.changeSetContext().run(context -> {
    // Este código é executado em um ChangeSet dedicado
});
```



# O Melhor dos Dois Mundos: Integrando com o `@Transactional` do Spring

Em aplicações Spring Boot, o CAP se integra perfeitamente com o gerenciamento de transações do Spring. Você pode usar a anotação `@Transactional` diretamente nos seus `Event Handlers`.



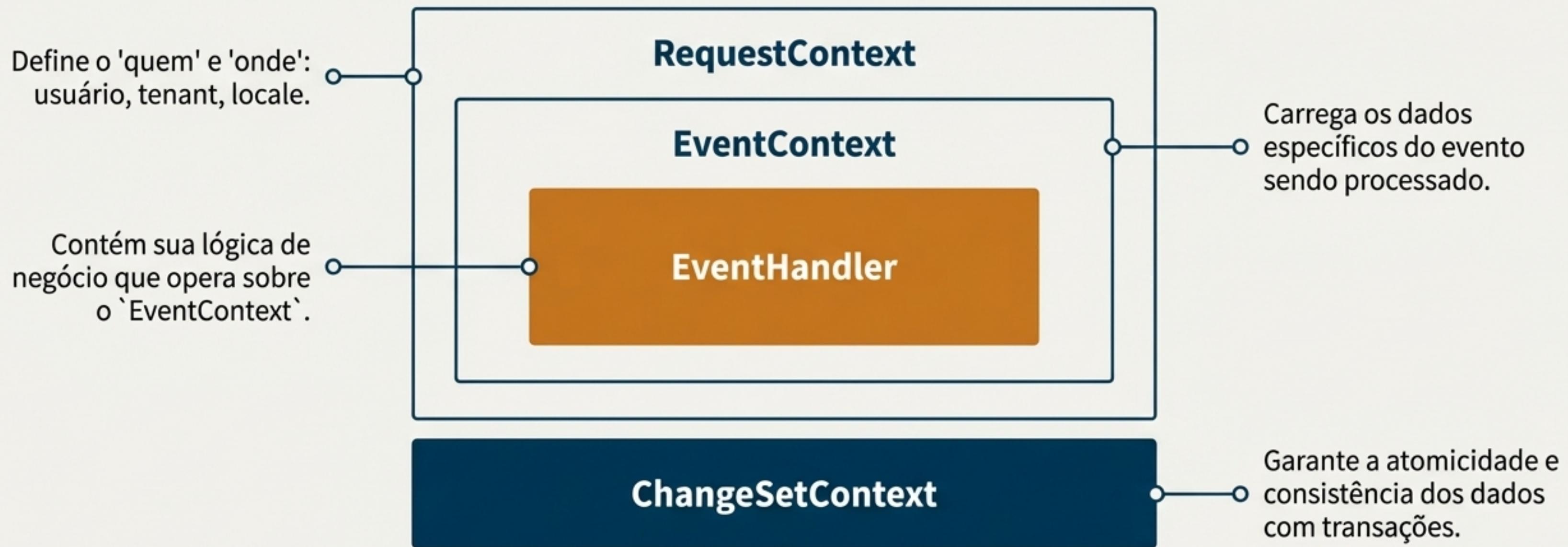
```
@Before(event = CqnService.EVENT_CREATE, entity = Books_.CDS_NAME)
@Transactional // Garante que uma transação JDBC esteja ativa aqui
public void beforeCreateBooks(List<Books> books) {
    // Acesso seguro a um JdbcTemplate ou Connection
    jdbc.queryForList("SELECT 1 FROM DUMMY");
}
```

## Benefícios

- **Controle Eager:** Inicia a transação no início do método, em vez de esperar pela primeira chamada de persistência.
- **Propagação:** Suporta atributos de propagação do Spring, como `REQUIRES\_NEW`, para suspender a transação atual e iniciar uma nova.
- **Interoperabilidade:** Facilita o uso de APIs baseadas em JDBC (como `JdbcTemplate`) dentro da mesma transação gerenciada pelo CAP.



# A Anatomia Completa de uma Requisição CAP



Sua lógica (`EventHandler`) acessa dados através de um `EventContext`. Ambos operam dentro de um `RequestContext` que define o ambiente. E tudo é sustentado por um `ChangeSetContext`, que garante a integridade transacional. Entender essas camadas é a chave para escrever aplicações CAP robustas e eficientes.

# Melhores Práticas para o Desenvolvedor CAP Java



## Use Tipos a seu Favor

Prefira sempre interfaces `EventContext` específicas e data accessors gerados pelo Maven Plugin em vez de mapas genéricos.



## Escolha a Ferramenta de Erro Certa

Use `ServiceException` para falhas que devem parar tudo e a API `Messages` para coletar erros de validação sem interromper o fluxo imediatamente.



## Seja Explícito sobre o Contexto

Utilize o `RequestContextRunner` quando precisar executar lógicas com diferentes permissões ou em nome de outros tenants. É explícito e seguro.



## Confie no Padrão Transacional

Para a maioria dos casos, o `ChangeSetContext` automático do CAP oferece a segurança transacional necessária sem configuração extra.



## Abrace o Ecossistema Spring

Use `@Transactional` quando precisar de controle fino sobre os limites da transação ou para integrar com código JDBC legado.



## Mire na UI

Use `messageTarget` ao lançar exceções para fornecer feedback preciso e contextualizado em aplicações Fiori.