

Dominando Application Services em CAP Java

Do Evento CRUD à Configuração Fina da sua API

O Application Service: A Peça Central da sua API

Application Services definem as APIs que uma aplicação CAP expõe aos seus clientes, por exemplo, através de OData. Eles funcionam como a ponte entre o modelo CDS e o mundo externo, utilizando CQN (CDS Query Notation) para interagir com a camada de persistência. É nesta camada que os eventos são disparados, permitindo a injeção da sua lógica de negócio.



O Ciclo de Vida: Do Verbo HTTP ao Evento CRUD

Os adaptadores de protocolo OData transformam requisições HTTP em consultas CQN, que por sua vez disparam eventos CRUD específicos no seu Application Service. Entender este mapeamento é fundamental.

Verbo HTTP	Evento CRUD	Constante CqnService
POST	CREATE	`CqnService.EVENT_CREATE`
GET	READ	`CqnService.EVENT_READ`
PATCH	UPDATE	`CqnService.EVENT_UPDATE`
PUT	UPDATE	`CqnService.EVENT_UPDATE`
DELETE	DELETE	`CqnService.EVENT_DELETE`

A constante de evento e sua respectiva interface de `EventContext` (ex: `CdsCreateEventContext`) devem ser usadas ao registrar e implementar os handlers de eventos.

Implementando a Lógica: O Papel dos Event Handlers

A forma primária de adicionar lógica de negócio é através de event handlers. Anotações como `@Before`, `@On` e `@After` permitem que seu código seja executado em fases específicas do processamento do evento.

`@Before`: Executa *antes* da fase principal do evento.

CqnService.EVENT_CREATE: O tipo de evento CRUD que estamos interceptando.

Books_.CDS_NAME: A entidade específica à qual este handler se aplica.

```
@Before(event = CqnService.EVENT_CREATE, entity = Books_.CDS_NAME)
public void createBooks(CdsCreateEventContext context, List<Books> books) {
    // Sua lógica de validação ou modificação aqui...
}
```

Lidando com Estruturas Profundas (Deep Documents)

Conceito Chave

Eventos CRUD são disparados apenas na entidade *raiz* da requisição. Para validações em entidades aninhadas (composições), você deve invocar a lógica de validação manualmente.

Exemplo de Código (Java)

```
@Before(event = CqnService.EVENT_CREATE, entity = Orders_.CDS_NAME)
public void validateOrders(List<Orders> orders) {
    for(Orders order : orders) {
        if (order.getItems() != null) {
            // Chamada manual para a validação da composição 'items'
            validateItems(order.getItems());
        }
    }
}

// Handler separado para quando OrderItems é criado diretamente
@Before(event = CqnService.EVENT_CREATE, entity = OrderItems_.CDS_NAME)
public void validateItems(List<OrderItems> items) {
    // Lógica de validação para os itens...
}
```

Explicação

No exemplo, `validateItems` é chamado de dentro de `validateOrders` para garantir que a lógica seja aplicada ao criar um pedido completo.

O handler `validateItems` também existe de forma independente para cobrir casos de criação direta de itens.

Dominando o Retorno: A Regra de Ouro dos Handlers `@On`

Handlers `@On` para eventos `READ`, `UPDATE` e `DELETE` **devem** definir um resultado, seja retornando-o diretamente ou usando o método `setResult()` do contexto.



READ

Deve retornar um `Iterable<Map>` ou um objeto `Result`. Use `Result` para queries com `\\$count`.



UPDATE / DELETE

Deve retornar um `Result` com o número de linhas afetadas.

⚠️ Aviso: Se nenhum resultado for definido, o framework assume 0 linhas, resultando em um `HTTP 404 Not Found` para o cliente.



INSERT / UPSERT

Pode, opcionalmente, retornar os dados que foram inseridos/atualizados.

O `ResultBuilder`: Seu Ferramental para Respostas Precisas

Use a classe `ResultBuilder` para construir objetos `Result` com a semântica correta que os adaptadores de protocolo (como o OData) esperam.

Para READ (`selectedRows`)

```
// import static  
com.sap.cds.ResultBuilder.selectedRows;  
Result res = selectedRows(  
    asList(row)).inlineCount(count).result();  
context.setResult(res);
```

Para UPDATE (`updatedRows`)

```
// import static  
com.sap.cds.ResultBuilder.updatedRows;  
int updateCount = 1;  
Result r = updatedRows(updateCount, data).result();
```

Para DELETE (`deletedRows`)

```
// import static  
com.sap.cds.ResultBuilder.deletedRows;  
int deleteCount = 7;  
Result r = deletedRows(deleteCount).result();
```

Além do CRUD: Adicionando Actions & Functions

Actions e Functions aprimoram sua API com operações customizadas. Elas possuem parâmetros de entrada e valores de retorno bem definidos no modelo CDS e são implementadas com event handlers, assim como os eventos CRUD.

Etapa 1: Definição no Modelo CDS

```
service CatalogService {  
    entity Books { ... }  
    actions {  
        action review(stars: Integer) returns Reviews;  
    };  
}
```



Etapa 2: Implementação do Handler Java

```
@On(event = "review", entity = Books_.CDS_NAME)  
public void reviewAction(ReviewEventContext context) {  
    // Lógica para criar a review...  
    context.setResult(review);  
}
```

A Magia da Geração de Código: Contextos de Evento Tipados

O `cds-maven-plugin` analisa seu modelo CDS e gera interfaces Java `EventContext` específicas para suas actions e functions. Isso proporciona acesso direto e seguro (type-safe) aos parâmetros de entrada e ao valor de retorno.

```
// Interface gerada pelo CAP Java SDK Maven Plugin:  
@EventName("review")  
public interface ReviewEventContext extends EventContext {  
    // CqnSelect para a entidade alvo  
    CqnSelect getCqn();  
  
    // Parâmetro de entrada 'stars'  
    Integer getStars();  
    void setStars(Integer stars);  
  
    // Valor de retorno  
    void setResult(Reviews review);  
    Reviews getResult();  
}
```

Destaque: Enfatizar como `getStars()` e `setResult(review)` são muito mais seguros e claros do que `context.get("stars")`.

Invocando Actions: A Abordagem Moderna e Tipada

A partir da versão 2.4.0, o Maven plugin também gera interfaces de serviço específicas. Injete e utilize essas interfaces para invocar suas actions diretamente, de forma limpa e fortemente tipada.

Bloco 1: Interface de Serviço Gerada

```
// Interface de Serviço gerada:  
@CdsName(CatalogService_.CDS_NAME)  
public interface CatalogService extends  
CqnService {  
    Reviews review(Books_ ref,  
        @CdsName(ReviewContext.STARS) Integer  
        stars);  
}
```

Bloco 2: Invocação no seu Código

```
// Invocação no seu código:  
@Autowired  
private CatalogService catService;  
  
private void someCustomMethod() {  
    // Chamada direta, limpa e type-safe  
    this.catService.review(ref, 5);  
}
```

Comparativo: Mencionar brevemente que isso substitui a abordagem mais verbosa de criar e emitir um `EventContext` manualmente.

Anatomia de uma URL de Serviço

O caminho combinado em que um serviço é exposto é composto pelo caminho base do adaptador de protocolo e pelo caminho relativo do próprio serviço. Ambos são configuráveis.

`http(s)://<application_url>/<base_path>/<service_name>`

- **<application_url>** O host da sua aplicação.
- **<base_path>** O caminho do adaptador de protocolo (ex: /odata/v4). Configurável.
- **<service_name>** O caminho relativo do serviço (ex: browse). Configurável.

Configurando Path e Protocolo: Anotações vs. `application.yaml`

Você pode configurar o caminho relativo e os protocolos de um serviço diretamente no seu modelo CDS com anotações, ou de forma centralizada no seu arquivo `application.yaml`.

Anotações no CDS

```
@path : 'browse'  
@protocols: [ 'odata-v4', 'odata-v2' ]  
service CatalogService { ... }
```

Configuração em `application.yaml`

```
cds:  
  application:  
    services:  
      CatalogService:  
        serve:  
          path: 'browse'  
        protocols:  
          - 'odata-v4'  
          - 'odata-v2'
```

Configuração Avançada: Múltiplos Endpoints por Serviço

Para cenários mais complexos, você pode servir o mesmo **serviço** em caminhos diferentes para protocolos distintos usando a anotação `@endpoints` ou a configuração equivalente em YAML.

Anotações no CDS

```
@endpoints: [
    {path : 'browse', protocol: 'odata-v4'},
    {path : 'list', protocol: 'odata-v2'}
]
service CatalogService { ... }
```

Configuração em `application.yaml`

```
cds:
  application:
    services:
      CatalogService:
        serve:
          endpoints:
            - path: 'browse'
              protocol: 'odata-v4'
            - path: 'list'
              protocol: 'odata-v2'
```

Resultado

Este serviço estará disponível em `/odata/v4/browse` e `/odata/v2/list`.

Guia de Boas Práticas e Arquitetura

1. Localização da Lógica

- Application Service: Ideal para lógica de negócio/domínio.
- Persistence Service: Para requisitos técnicos de baixo nível, como logging genérico na escrita ao banco. Lembre-se que verificações de autenticação não são executadas aqui.

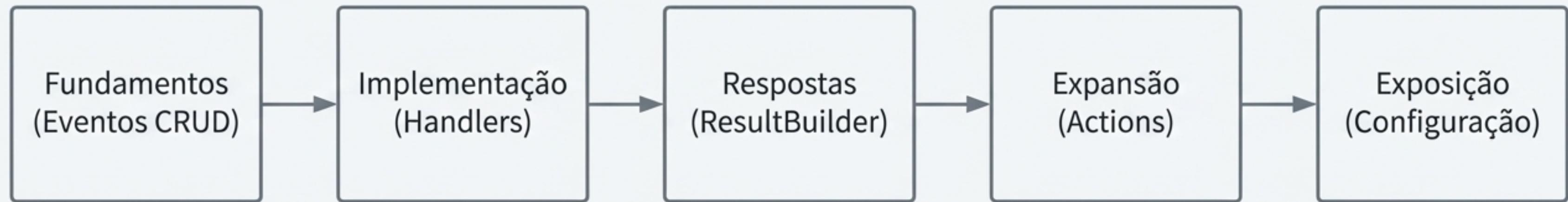
2. Interação entre Serviços

- **Consumir APIs:** Ao interagir com outro Application Service, prefira consumir sua API de serviço. Isso promove o desacoplamento e prepara para uma arquitetura de microserviços.
- **Usar Draft Service:** Para entidades com suporte a rascunho, sempre interaja com os estados de rascunho através do Draft Service.

3. Compartilhamento de Lógica

- Métodos Utilitários: Para compartilhar lógica de negócio entre diferentes handlers ou serviços, implemente-a em métodos utilitários (**utils**) utilitários (**utils**) simples e reutilizáveis.

O Blueprint para APIs Robustas e Flexíveis



Você agora possui o blueprint para construir APIs robustas, flexíveis e bem definidas com CAP Java. Da interceptação de um evento simples à configuração detalhada da exposição do seu serviço, você tem o controle total sobre o comportamento da sua aplicação.