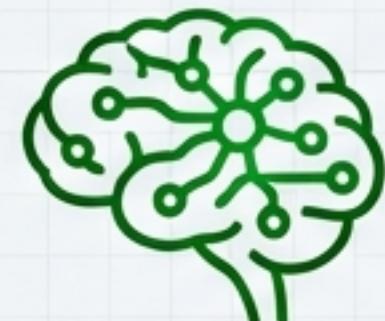
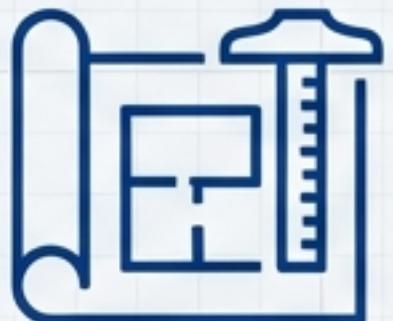


O Modelo Vivo

Introspecção e Adaptação Dinâmica
de Modelos CDS em CAP Java

Seu Modelo CDS: De Blueprint Estático a Entidade Viva



O Blueprint Estático - Visão em Tempo de Design

Em tempo de design, seu modelo CDS define a estrutura, as entidades e os serviços. É o contrato fundamental da sua aplicação.

O Blueprint Vivo - Realidade em Tempo de Execução

Em tempo de execução, o CAP Java transforma esse contrato em um objeto dinâmico. Um modelo que pode ser inspecionado e até mesmo modificado a cada request.

Nesta sessão, vamos explorar duas capacidades avançadas para interagir com este "modelo vivo":

1. ****Introspecção****: Lendo e entendendo o blueprint em detalhes.
2. ****Adaptação****: Alterando o blueprint dinamicamente.

O Ponto de Partida para Introspecção: A Interface `CdsModel`

A interface `com.sap.cds.reflect.CdsModel` representa o modelo CDS completo da sua aplicação. Obter uma instância dela é o primeiro passo.

1. A partir do `EventContext` (em Handlers)

```
// Em um handler de evento
@On(event = "READ", entity ="CatalogService.Books")
public void readBooksVerify(EventContext context) {
    CdsModel model = context.getModel();
    // ...
}
```

2. Via Injeção de Dependência (Spring)

```
// Injetado como um bean Spring
@Autowired
CdsModel model;
```

3. Diretamente de um CSN

```
// Lendo de um InputStream de um JSON CSN
InputStream csnJson = ...;
CdsModel model = CdsModel.read(csnJson);
```

Inspecionando os Detalhes de um Elemento

Vamos usar este modelo `my.bookshop` como exemplo para inspecionar o elemento `title` da entidade `Books`.

```
namespace my.bookshop;

entity Books {
    title : localized String(111);
    author : Association to Authors;
    //...
}
```

```
CdsEntity books = model.getEntity("my.bookshop.Books");
CdsElement title = books.getElement("title"); // Obtém o elemento pelo nome

boolean key = title.isKey(); //<-- Retorna: false ← false
boolean localized = title.isLocalized(); //<-- Retorna: true ←
CdsType type = title.getType(); //<-- Retorna: CdsSimpleType

if (type.isSimple()) {
    CdsSimpleType simple = type.as(CdsSimpleType.class);

    String typeName = simple.getQualifiedName(); //<-- "cds.String"
    CdsBaseType baseType = simple.getType(); //<-- CdsBaseType.STRING
    Class<?> javaType = simple.getJavaType(); //<-- String.class
    Integer length = simple.get("length"); //<-- 111
}
```

Navegando por Associações e Suas Propriedades

A API de Reflexão oferece métodos detalhados para analisar associações, permitindo que sua lógica entenda as relações do modelo.

```
CdsElement authorElement = books.getAssociation("author");
CdsAssociationType toAuthor = authorElement.getType();

CdsEntity target = toAuthor.getTarget(); ← // Entidade alvo: my.bookshop.Authors

boolean isAssociation = toAuthor.isAssociation(); //<-- true
boolean isComposition = toAuthor.isComposition(); //<-- false

// Analisando a Cardinalidade
Cardinality cardinality = toAuthor.getCardinality();
String sourceMax = cardinality.getSourceMax(); //<-- "*"
String targetMin = cardinality.getTargetMin(); //<-- "0"
String targetMax = cardinality.getTargetMax(); //<-- "1"

// Acessando as chaves da associação
Stream<CdsElement> keys = toAuthor.keys(); //<-- Stream contendo o elemento 'ID' de Authors
```



Lendo Anotações para Criar Lógica Orientada a Metadados

Imagine que você precisa obter o 'nome de exibição' de um campo. Se a anotação `@title` existir, use seu valor; caso contrário, use o nome do próprio elemento.

```
entity Orders {  
    OrderNo : String @title:'Order Number';  
    // ...  
}
```

```
CdsEntity order = model.getEntity("my.bookshop.Orders");  
CdsElement orderNo = order.getElement("OrderNo");  
  
// Procura pela anotação pelo nome  
Optional<CdsAnnotation<String>> annotation = orderNo.findAnnotation("title");  
  
// Usa o valor da anotação, se presente. Senão, usa o nome do elemento.  
String displayName = annotation.map(CdsAnnotation::getValue)  
                    .orElse(orderNo.getName());  
  
// Resultado: "Order Number"
```

Consultas Avançadas: Filtrando o Modelo Programaticamente

A API fornece predicados estáticos para filtrar streams de definições do modelo de forma declarativa.

Filtrar Entidades por Namespace

```
import static com.sap.cds.reflect.CdsDefinition  
.byNamespace;  
  
// Encontra todas as entidades no namespace  
'my.bookshop'  
Stream<CdsEntity> entities = model.entities()  
.filter(byNamespace("my.bookshop"));
```

Encontrar Elementos com uma Anotação Específica

```
import static com.sap.cds.reflect.CdsAnnotatable  
.byAnnotation;  
  
// Encontra todos os elementos em 'Orders' que  
possuem a anotação '@title'  
Stream<CdsElement> elements = order.elements()  
.filter(byAnnotation("title"));
```

Com estas ferramentas, seu código pode não apenas ler, mas também entender e reagir dinamicamente à estrutura do seu modelo.

Adaptação: Modificando o Blueprint em Tempo Real

Agora que sabemos como ler o blueprint, vamos ao próximo nível: alterá-lo. Feature Toggles permitem habilitar ou desabilitar partes do seu modelo CDS dinamicamente, por request.



Release Toggles: Liberar novas funcionalidades para clientes específicos.

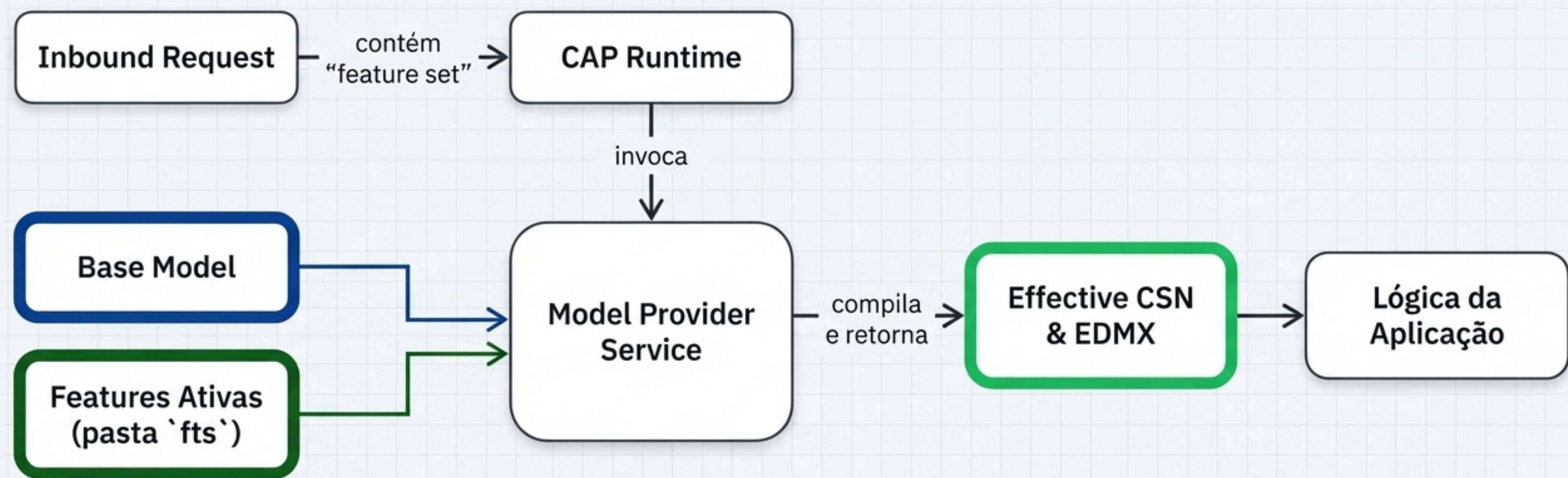


Runtime Toggles: Habilitar/desabilitar features para usuários ou tenants específicos.



UIs Dinâmicas: Alterar a UI de um Fiori Elements sem precisar de um novo deploy.

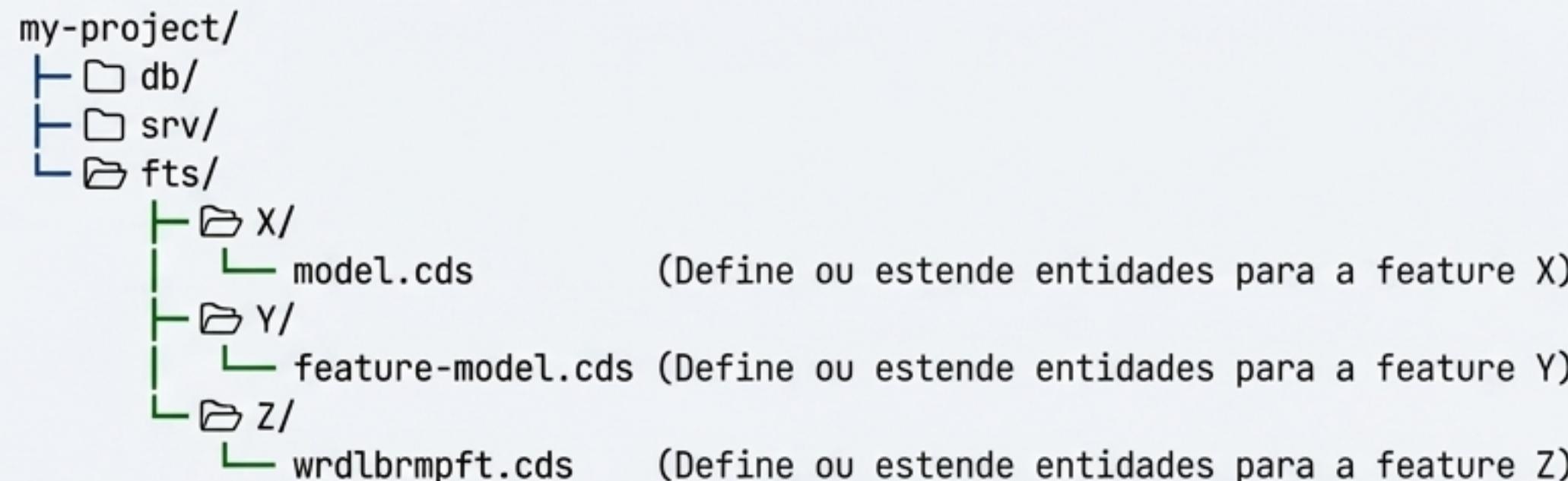
O Mecanismo por Trás da Magia: `Model Provider Service`



Este processo ocorre **por request**, garantindo que cada usuário ou tenant receba a visão do modelo que corresponde exatamente às suas features ativadas.

Estruturando suas Features: A Pasta `fts`

Definições de CDS específicas de uma *feature* são colocadas em subpastas dentro de uma pasta `fts` na raiz do seu projeto. O nome da subpasta se torna o nome da *feature*.



E o Banco de Dados?

O schema do banco de dados gerado no build time **contém todas as features**. O CAP Framework garante que o acesso aos dados em tempo de execução respeite o modelo ativo. Um `SELECT *` retornará apenas as colunas definidas na visão atual do modelo, mesmo que outras existam na tabela física.

Determinando o Conjunto de Features Ativas

O CAP Java determina as *features* ativas para cada *request* através de um `FeatureTogglesInfoProvider`. Por padrão, todas as *features* estão desativadas.

1. Configuração de Usuários Mock (application.yaml)

Ideal para desenvolvimento e testes.

```
cds:  
  security:  
    mock:  
      users:  
        - name: Bob  
          features: [ wobble ]  
        - name: Alice  
          tenant: SmartCars  
          features: [ cruise, parking ]
```

2. Implementação de um `FeatureTogglesInfoProvider` Customizado

Para cenários de produção, crie uma lógica baseada em roles, tenants, ou chamadas a serviços externos.

```
@Component  
public class DemoFTProvider implements FeatureTogglesInfoProv  
  @Override  
  public FeatureTogglesInfo get(UserInfo userInfo, ...) {  
    Map<String, Boolean> toggles = new HashMap<>();  
    if (userInfo.hasRole("expert")) {  
      toggles.put("isbn", true);  
    }  
    return FeatureTogglesInfo.create(toggles);  
  }  
}
```

Usando Feature Toggles no seu Código de Negócio

Sua lógica customizada pode verificar se uma feature está ativa para executar blocos de código condicionalmente.

O `FeatureTogglesInfo` está disponível no contexto do evento ou pode ser injetado.

```
@After  
protected void subtractDiscount(CdsReadEventContext context) {  
  
    // Verifica se a feature 'discount' está habilitada para a request atual  
    if (context.getFeatureTogglesInfo().isEnabled("discount")) {  
  
        // Lógica customizada para aplicar o desconto é executada aqui  
        // ...  
    }  
}
```

Casos de Uso Avançados



1. Alternando Elementos de UI em Fiori Elements

Não é necessário código extra. O framework garante que a resposta `'\$metadata` do serviço OData respeite o 'feature vector' da requisição. Anotações de UI em arquivos CDS de features irão aparecer ou desaparecer automaticamente na UI.



2. Definindo Features para Chamadas de Serviço Internas

Para código executado fora de um contexto de request padrão (ex: jobs, novas threads), você pode definir explicitamente o conjunto de features a serem usadas.

```
FeatureTogglesInfo isbnFeature = FeatureTogglesInfo.create(  
    Collections.singletonMap("isbn", true));  
  
Future<...> result = Executors.newSingleThreadExecutor().submit(() -> {  
    // Executa o código dentro de um novo contexto com a feature 'isbn' ativada  
    return runtime.requestContext()  
        .featureToggles(isbnFeature)  
        .run(rc -> {  
            return db.run(Select.from(Books_.class));  
        });  
});
```

O Blueprint em Ação: Uma Síntese



Introspecção (API de Reflexão)

- **Leia** o seu modelo.
- Permite que seu código **entenda e reaja** à estrutura, tipos, associações e anotações do seu modelo em tempo de execução.
- Crie lógicas genéricas e orientadas a metadados.

Adaptação (Feature Toggles)

- **Altere** o seu modelo.
- Permite que seu modelo se **transforme dinamicamente** para cada request, baseado em usuário, tenant ou qualquer outro contexto.
- Construa aplicações multi-tenant e flexíveis com UIs e lógicas contextuais.

O modelo não é apenas um contrato. É um diálogo.

Use a API de Reflexão para ouvir o que seu modelo diz.

Use Feature Toggles para responder. Transforme suas aplicações CAP Java em sistemas verdadeiramente dinâmicos e inteligentes.