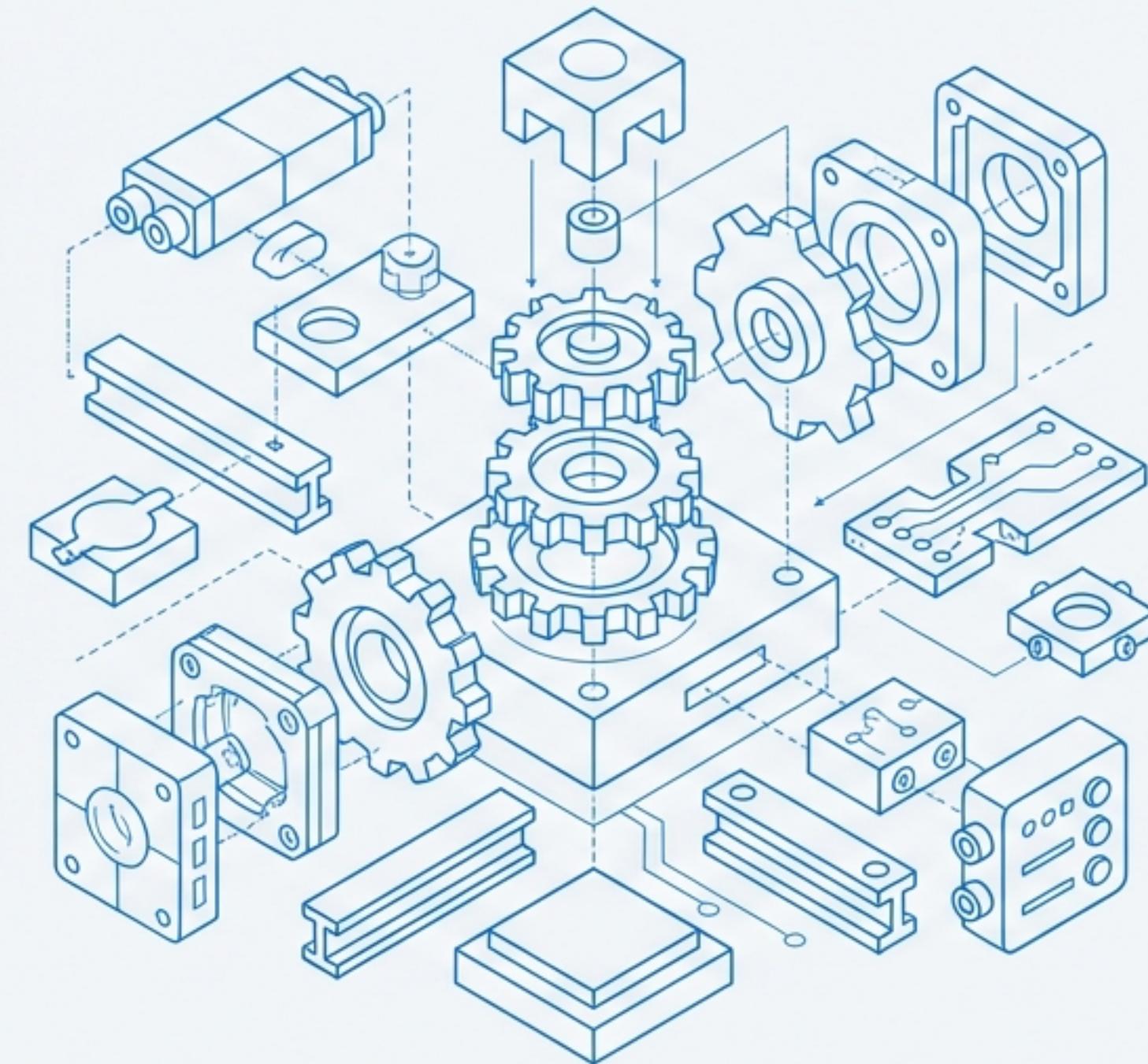


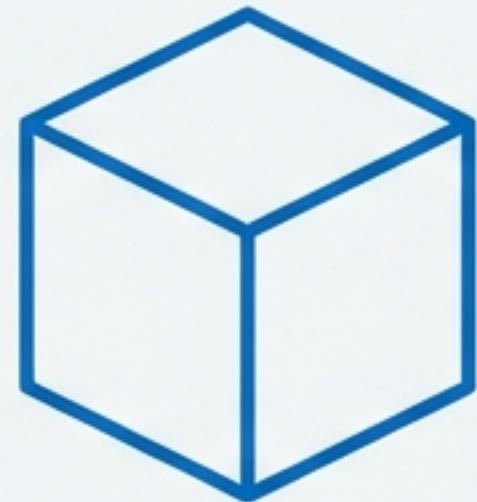
# Construindo Plugins Reutilizáveis em CAP Java

Um Guia Prático para Arquiteturas Modulares e Escaláveis



# O Desafio: A Complexidade Crescente em Projetos CAP Java

À medida que projetos crescem, ou quando se constroem serviços de plataforma para integração, surge a necessidade de estender o CAP Java com código que seja tanto customizado quanto reutilizável.



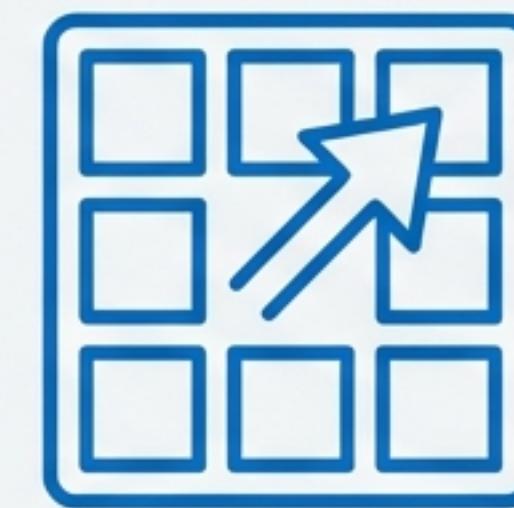
## Código Monolítico:

Dificuldade de manutenção e evolução.



## Duplicação de Lógica:

Inconsistências e retrabalho entre diferentes aplicações.



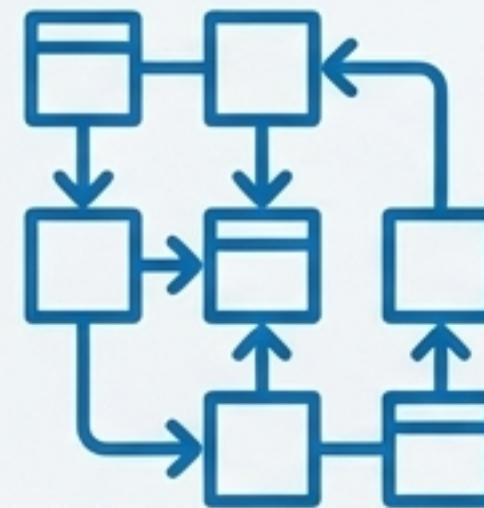
## Escalabilidade Comprometida:

Arquiteturas rígidas que impedem o crescimento modular.

*Como podemos construir aplicações mais robustas e manuteníveis sem reinventar a roda a cada novo serviço?*

# A Solução: O Kit de Ferramentas de Plugins para CAP Java

O CAP Java oferece um conjunto de mecanismos poderosos para criar componentes reutilizáveis. Vamos explorar as três ferramentas essenciais em seu kit.



## Ferramenta #1: Modelos CDS

Para compartilhar a fundação: seus modelos de dados, dados de importação e arquivos de internacionalização.



## Ferramenta #2: Event Handlers

Para empacotar a inteligência: sua lógica de negócio customizada e validações.



## Ferramenta #3: Protocol Adapters

Para criar novas portas de entrada: seus canais de comunicação inbound customizados.

# Regras Essenciais Antes de Começar

## Compatibilidade de Bytecode é Crucial

O bytecode do seu plugin deve ser compatível com as aplicações que irão consumi-lo.

### RECOMENDAÇÃO: USE JAVA 17

\*Esta é a versão mínima para aplicações CAP Java e garante a máxima compatibilidade.\*

## Evite Conflitos de Nomenclatura

A escolha do GroupId e dos pacotes Java define a responsabilidade e evita confusão.

### ⚠ AVISO: NÃO UTILIZE O GROUPID `com.sap.cds`

\*Este namespace e suas subestruturas (ex: `com.sap.cds.foo.plugin`) são reservados para a equipe do CAP Java.\*

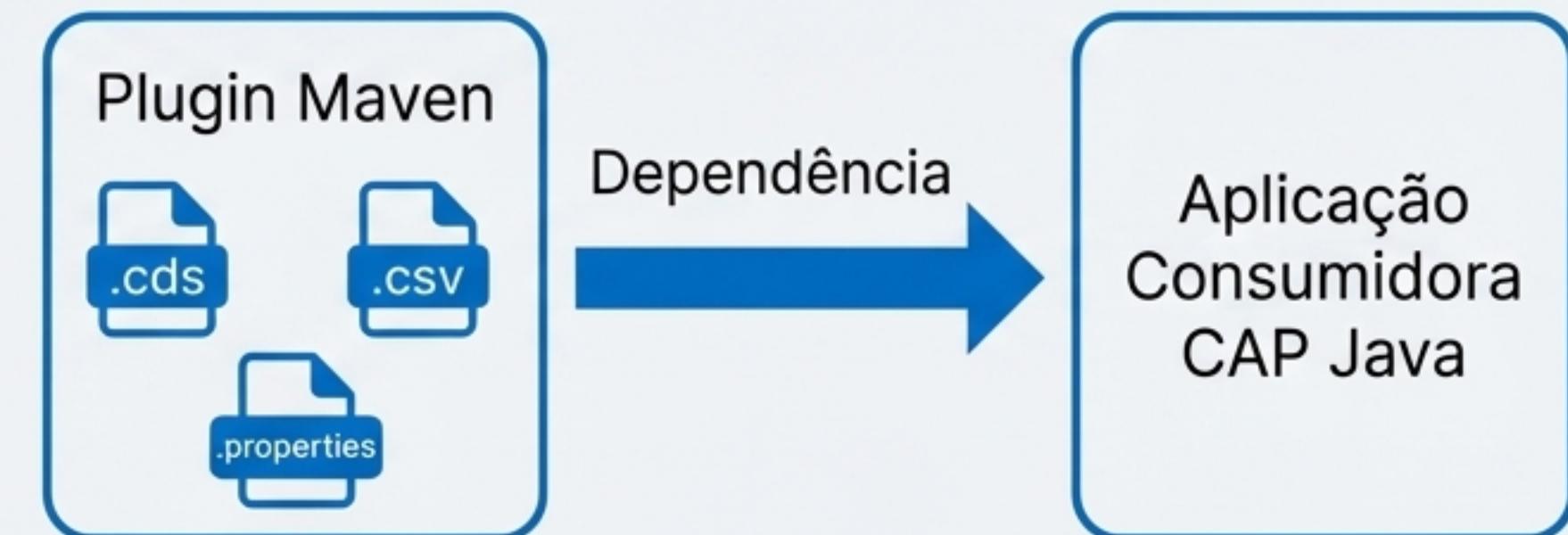
# Ferramenta #1: Compartilhando Modelos CDS via Artefatos Maven



A forma mais direta de reuso é compartilhar a definição de seus dados e recursos relacionados.

## O que você pode compartilhar em um único artefato Maven?

- Modelos de Dados (**.cds**)
- Dados de Importação (**.csv**)
- Arquivos de Internacionalização (**.properties**)
- Código Java (ex: Event Handlers associados)



# Passo a Passo: Criando o Artefato de Modelo

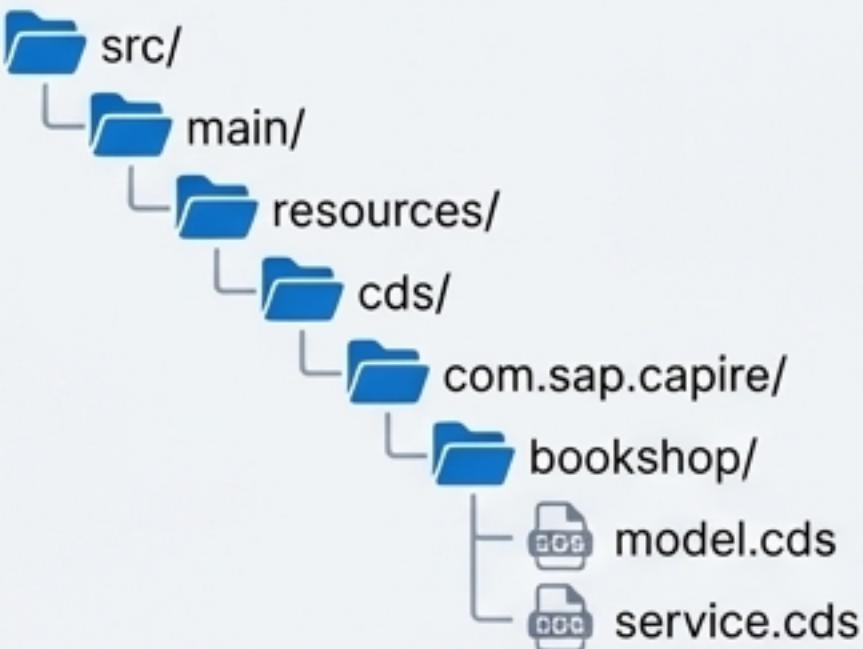
## Passo 1: Crie um Projeto Maven Simples

Utilize o arquétipo Maven para gerar a estrutura básica.

```
mvn archetype:generate -DgroupId=com.sap.capire -DartifactId=bookshop  
-DarchetypeGroupId=org.apache.maven.archetypes -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

## Passo 2: Adicione seus Modelos CDS

Coloque seus arquivos **.cds** em um diretório único dentro de **src/main/resources/cds**, seguindo a convenção **[groupId]/[artifactId]**.



### ATENÇÃO: Use um Projeto Maven Java Simples

*Seu projeto de plugin/modelo **não** deve ser criado como um projeto CAP Java ou Spring Boot.*

# Passo a Passo: Consumindo o Modelo em seu Projeto

## Passo 1: Adicione a Dependência no `srv/pom.xml`

Declare o artefato do seu plugin na seção <dependencies>.

```
<dependency>
    <groupId>com.sap.capire</groupId>
    <artifactId>bookshop</artifactId>
    <version>1.0.0</version>
</dependency>
```

## Passo 2: Configure o `cds-maven-plugin`

Adicione o goal `resolve` para que o plugin extraia os modelos para o diretório `target/cds/`, tornando-os visíveis para o compilador.

```
<plugin>
    <groupId>com.sap.cds</groupId>
    <artifactId>cds-maven-plugin</artifactId>
    ...
    <executions>
        <execution>
            <id>cds.resolve</id>
            <goals>
                <goal>resolve</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

## Passo 3: Utilize o Modelo em seu Código CDS

Importe os serviços e entidades com a diretiva `using`.

```
using { CatalogService } from 'com.sap.capire/bookshop';
```

## Ferramenta #2: Empacotando Lógica de Negócio com Event Handlers



Em CAP Java, Event Handlers não são acoplados diretamente ao runtime. Isso permite que sejam empacotados em bibliotecas externas (plugins) para fornecer funcionalidades customizadas e comuns.

### Casos de Uso Típicos

- Validações Genéricas: Implementar regras de validação que se aplicam a múltiplas entidades com base em tipos ou anotações comuns.
- Lógica de Negócio Reutilizável: Centralizar cálculos complexos ou enriquecimento de dados.
- Funcionalidades Transversais: Adicionar logging, auditoria ou controle de acesso de forma modular.

**Pré-requisito:** O projeto do plugin precisa das dependências do `cds-services-bom` para ter acesso às APIs do CAP Java.

# A Decisão Chave: Como o Plugin Será Carregado?

Ao fornecer um handler em uma biblioteca externa, você precisa decidir como ele será descoberto e registrado pela aplicação. A escolha depende das suas dependências.

## CAP Java `ServiceLoader`

### Quando Usar?

Quando seu handler é autossuficiente e **não depende** de outros componentes Spring (injeção de dependência). Ideal para lógica simples, validações ou cálculos que operam apenas no contexto do evento.

### Mecanismo

Utiliza o `ServiceLoader` padrão do Java para descobrir implementações da interface `CdsRuntimeConfiguration`.

### Vantagem

Independente de framework.

## Spring `AutoConfiguration`

RECOMENDADO

### Quando Usar?

Quando seu handler **depende** de outros componentes Spring (ex: usa `@Autowired`) ou precisa acessar serviços do CAP como `CqnService` ou `PersistenceService`.

### Mecanismo

Utiliza o mecanismo de `AutoConfiguration` do Spring Boot para registrar o handler como um Bean.

### Vantagem

Integração total com o ecossistema Spring. **Esta é a prática recomendada na maioria dos cenários.**

# Implementando com `ServiceLoader`

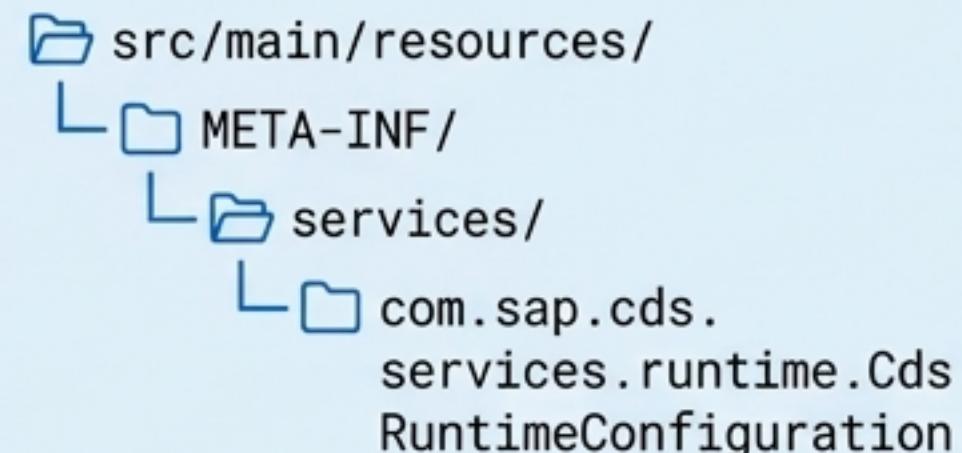
## Passo 1: Implemente a Interface `CdsRuntimeConfiguration`

Crie uma classe que implementa esta interface. No método `eventHandlers`, registre uma nova instância do seu handler.

```
public class SampleHandlerRuntime  
Configuration implements  
CdsRuntimeConfiguration {  
  
    @Override  
    public void eventHandlers(CdsRu  
ntimeConfigurer configurer) {  
        configurer.eventHandler(new  
        SampleHandler());  
    }  
}
```

## Passo 2: Crie o Arquivo de Serviço

No seu projeto de plugin, crie um arquivo de texto com o nome completo da interface no diretório `META-INF/services`.



```
src/main/resources/  
└ META-INF/  
    └ services/  
        └ com.sap.cds.  
            services.runtime.Cds  
            RuntimeConfiguration
```

## Passo 3: Registre sua Implementação

Dentro do arquivo, adicione o nome completo da sua classe de configuração.

```
com.sap.example.cds.SampleH  
andlerRuntimeConfiguration
```

# Boas Práticas: Implementando com Spring `AutoConfiguration`

## O Antipattern: Usar `@Component` Diretamente

Primary tex (Roboto Flex)

A anotação `@Component` em um plugin é frágil. O component scan do Spring Boot por padrão varre apenas os pacotes da aplicação principal. Isso forçaria o consumidor a ajustar seu `@ComponentScan` ou a seguir uma estrutura de pacotes rígida, o que quebra o propósito do reuso.

## A Solução Recomendada: `AutoConfiguration`

O mecanismo de `AutoConfiguration` do Spring Boot foi projetado exatamente para este cenário: permitir que bibliotecas externas registrem seus próprios beans (componentes) de forma automática e segura no `ApplicationContext`.

O plugin 'simplesmente funciona' ao ser adicionado como dependência, sem a necessidade de configuração manual por parte do consumidor.

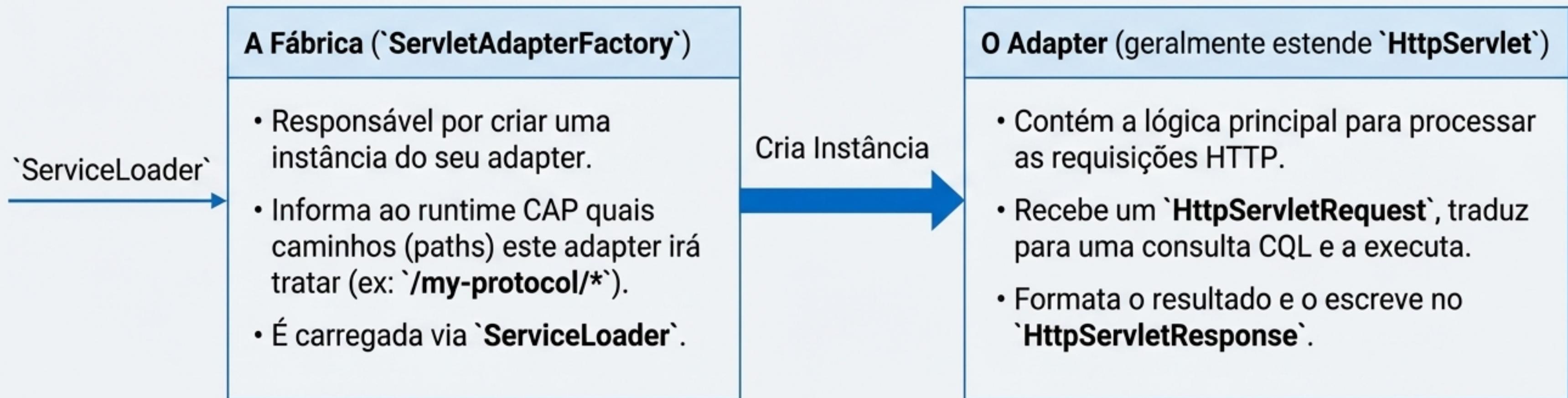
## Para Saber Mais

- Consulte a documentação de referência do Spring Boot sobre 'Creating your own auto-configuration'.
- Um exemplo completo de ponta a ponta pode ser encontrado no post do blog referenciado na documentação oficial.

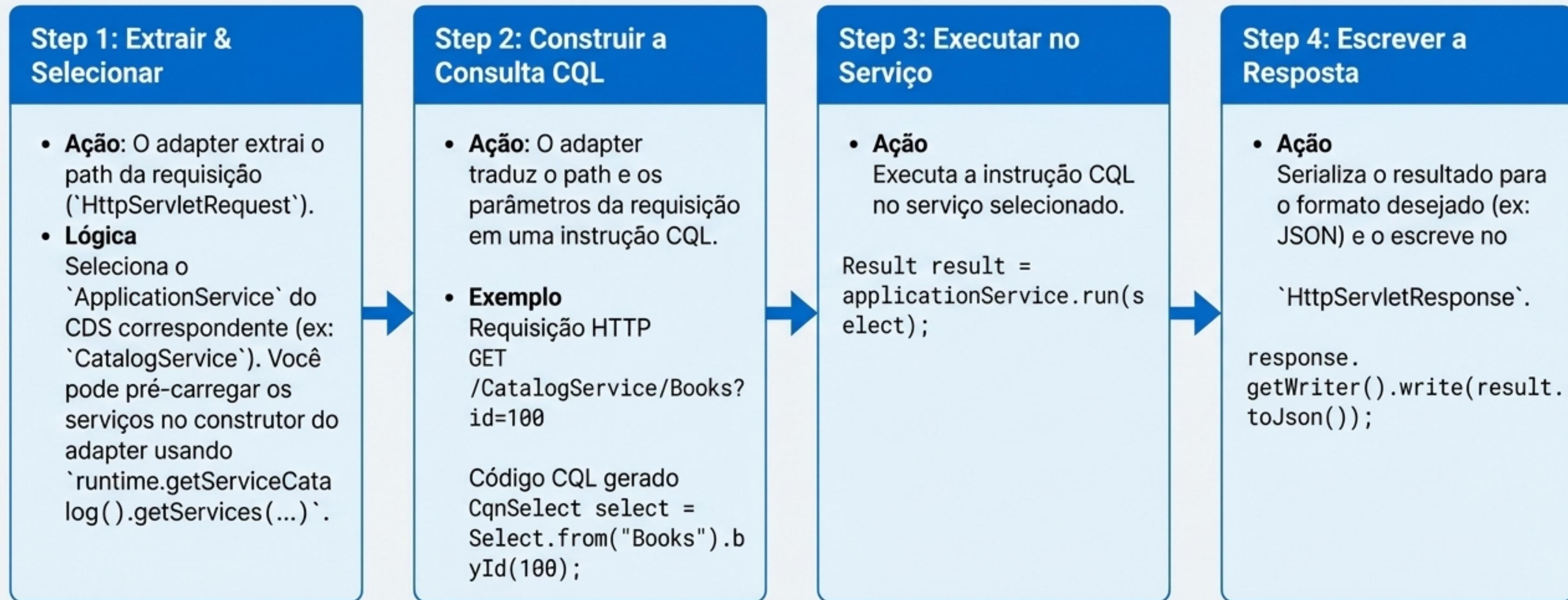
# Ferramenta #3: Criando Canais de Comunicação com Protocol Adapters

Um Protocol Adapter é o mecanismo para implementar comunicação **inbound** (de um serviço externo ou UI) para um serviço CAP. Essencialmente, você está criando um novo “dialeto” que sua aplicação CAP pode entender.

## Estrutura de um Protocol Adapter



# O Fluxo de Execução de um Protocol Adapter



# O Plano Mestre: Combinando as Ferramentas com Estratégia

## Recapitulando seu Kit de Ferramentas



**Modelos CDS:** A fundação compartilhada dos seus dados.



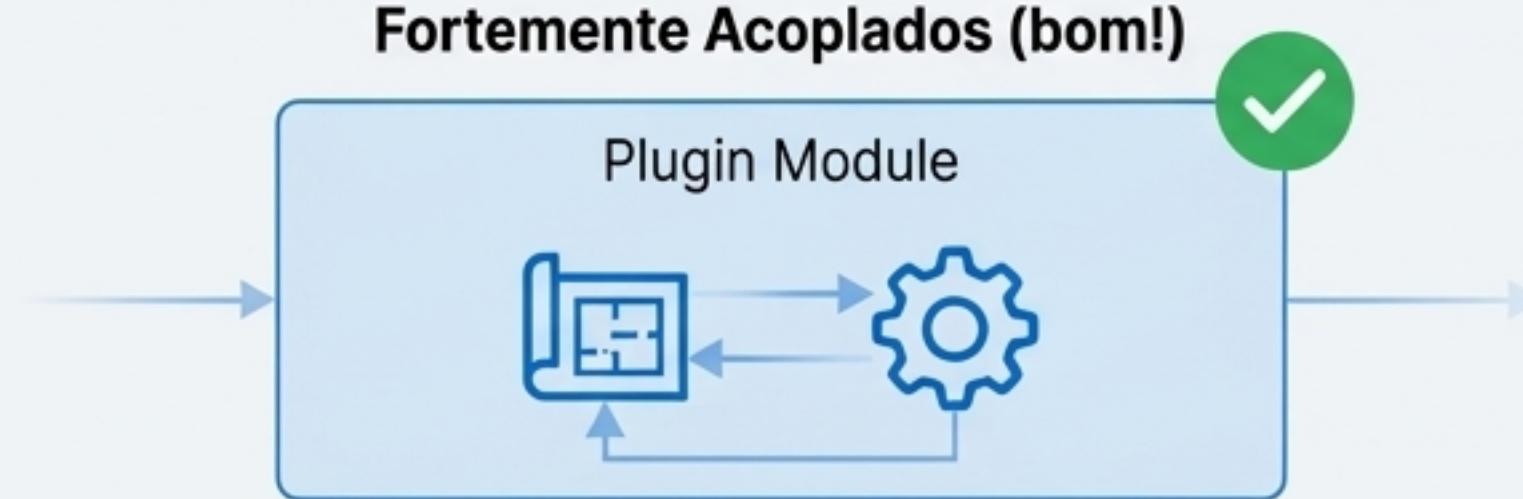
**Event Handlers:** A lógica de negócio reutilizável que atua sobre os modelos.



**Protocol Adapters:** As interfaces de comunicação independentes.

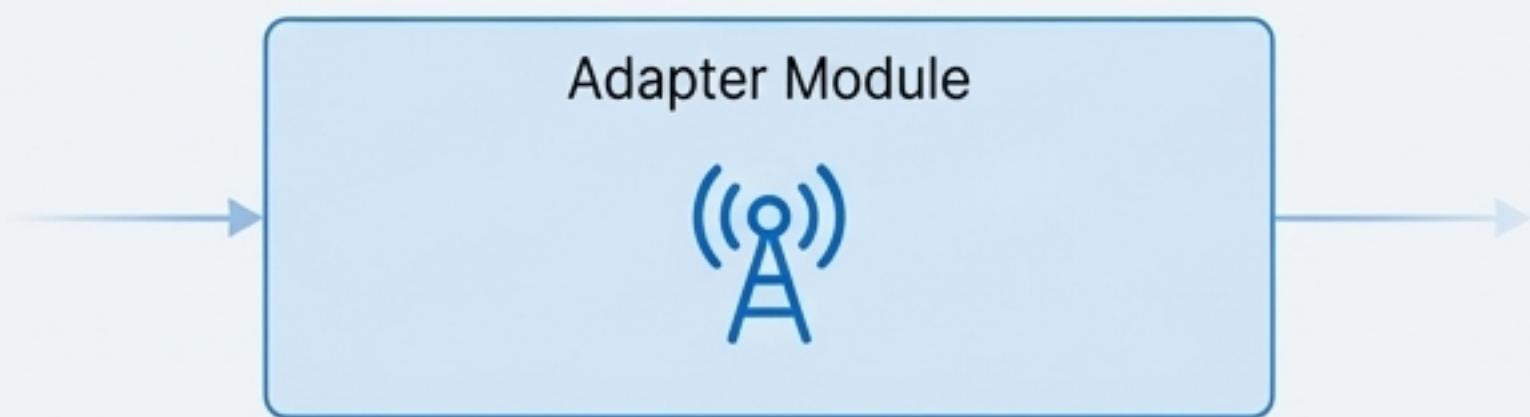
## Estratégia de Combinação Recomendada

### Fortemente Acoplados ( bom! )



É muito comum e eficaz combinar **Modelos CDS** e **Event Handlers** no mesmo módulo de plugin. A lógica de negócio (`EventHandler`) geralmente depende dos artefatos de dados (`CDS`) que ela manipula.

### Totalmente Independentes



**Protocol Adapters**, por outro lado, são genéricos e independentes do modelo. Eles devem ser empacotados e distribuídos separadamente para maximizar o reuso em diferentes domínios de negócios.

Ao dominar essas ferramentas, você transforma a complexidade em componentes modulares, construindo aplicações CAP Java que são verdadeiramente robustas, escaláveis e prontas para o futuro.