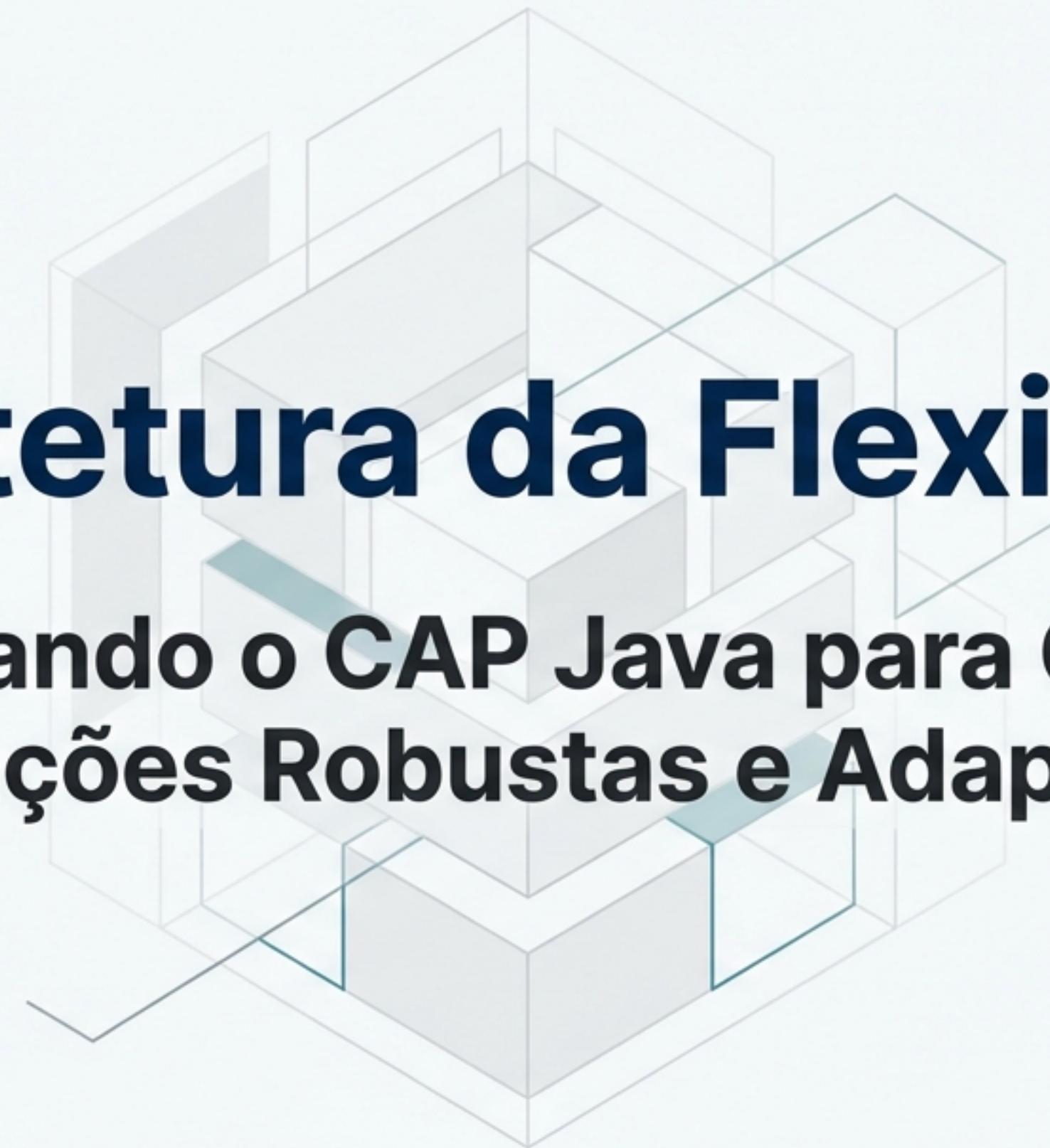


A Arquitetura da Flexibilidade

Desvendando o CAP Java para Construir Aplicações Robustas e Adaptáveis



Uma Filosofia de Design com Propósito



Opinativo para Guiar, Aberto para Customizar

Um dos princípios chave de design do CAP é ser um framework opinativo, mas ainda assim aberto. Ele oferece uma orientação clara para tecnologias de ponta, ao mesmo tempo que mantém a porta aberta para escolhas personalizadas.



Desacoplado para Durar, Agnóstico para Adaptar

Manter funcionalidades ortogonais separadas em componentes independentes torna esses componentes intercambiáveis. A aplicação se torna agnóstica em relação à plataforma e aos serviços, reduzindo o risco de adaptações custosas no código.

Os Benefícios de uma Arquitetura Desacoplada



Manutenção Simplificada

Substitua componentes concretos (como serviços de persistência ou o ambiente da plataforma) de forma independente. A preparação para diferentes contextos de implantação torna-se uma questão de configuração, não de adaptação adaptação de código.



Otimização de Recursos

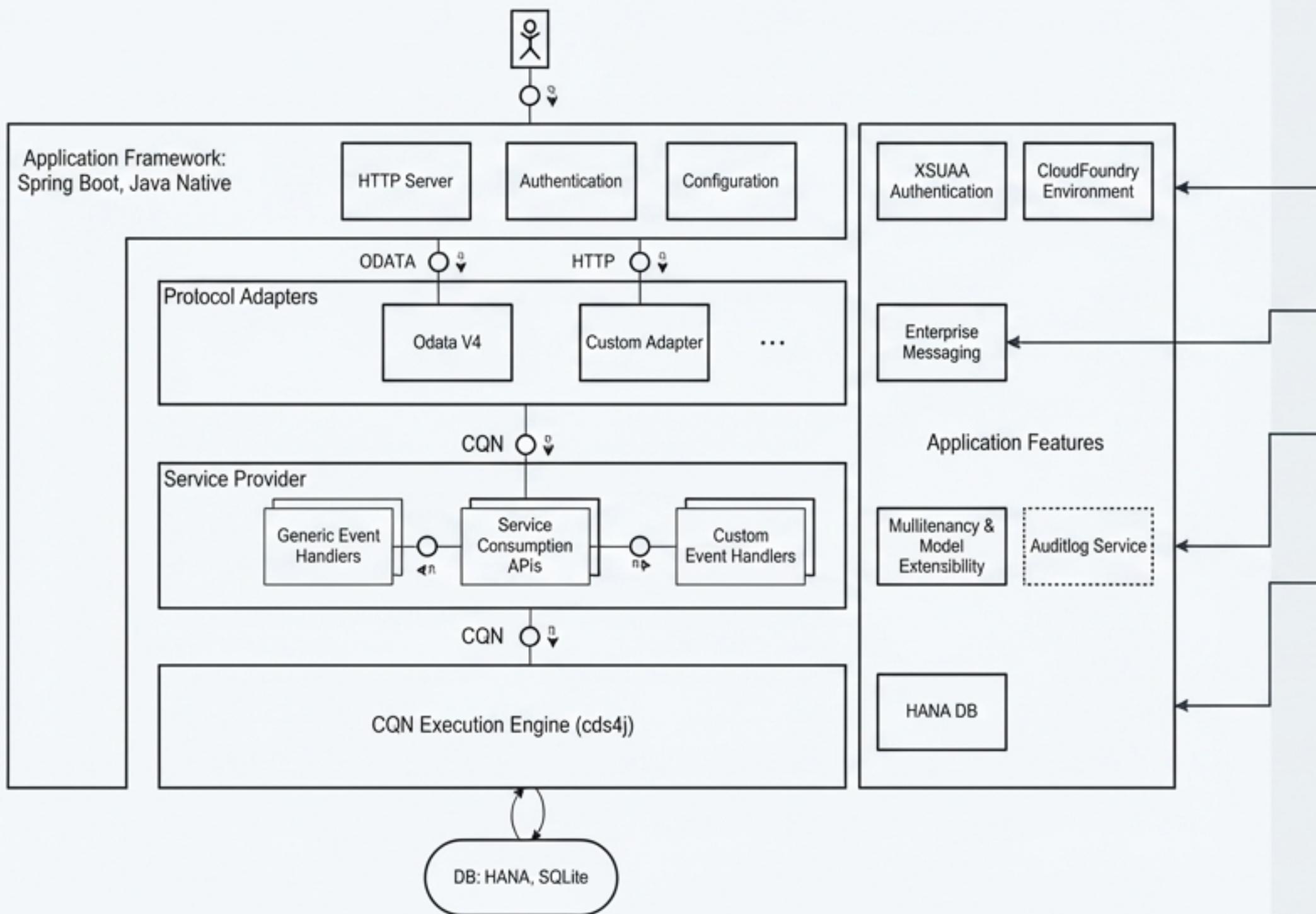
A modularização refinada permite montar um conjunto mínimo de componentes, apenas o necessário para os requisitos da aplicação. Isso reduz significativamente o consumo de recursos em tempo de execução e os custos de manutenção.



Testes Locais Massivamente Simplificados

Com componentes fracamente acoplados, o escopo do teste pode ser definido com precisão. É possível substituir dependências de serviços remotos por provedores locais, como rodar a aplicação localmente com H2 em vez de SAP HANA.

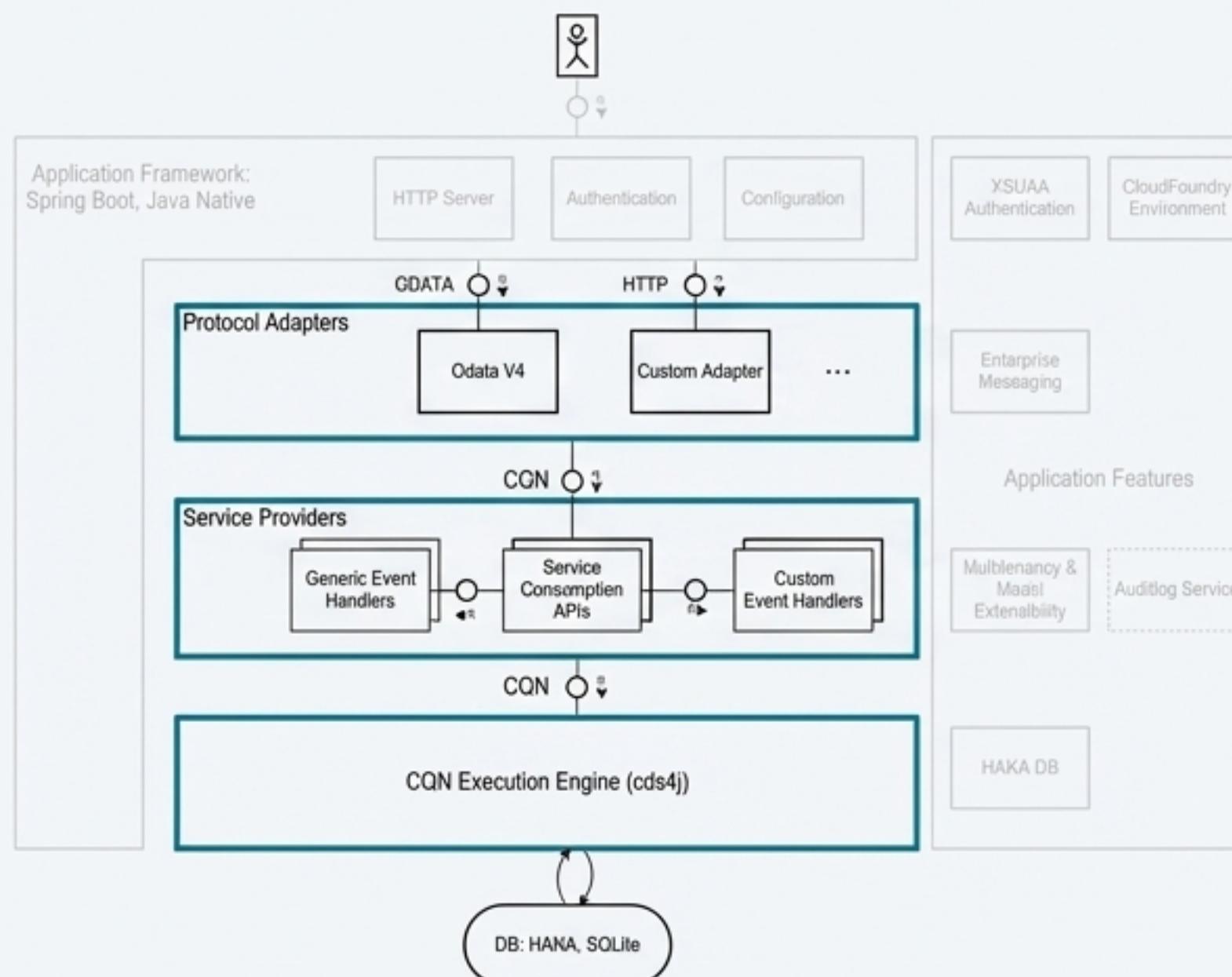
Anatomia do CAP Java: Uma Arquitetura em Camadas



A arquitetura modular reflete o requisito de flexibilidade máxima. Podemos reconhecer cinco áreas distintas na stack, com componentes agrupados de acordo com suas tarefas.

- 1. Application Framework:** A base da sua aplicação (ex: Spring Boot).
- 2. Protocol Adapters:** Mapeiam eventos web para o formato interno do CAP (CQN).
- 3. Service Providers:** Onde a lógica de negócio é implementada em event handlers.
- 4. CQN Execution Engine:** Traduz declarações CQN para a linguagem nativa do banco de dados.
- 5. Application Features:** Extensões opcionais para adicionar capacidades como multitenancy ou integração com serviços de plataforma.

O Fluxo Central: Do Protocolo à Persistência



1 Etapa 1: Protocol Adapters

Mapeiam protocolos de entrada, como OData V4, para eventos CQN (CDS Query Notation). Múltiplos adaptadores podem servir diferentes endpoints simultaneamente.

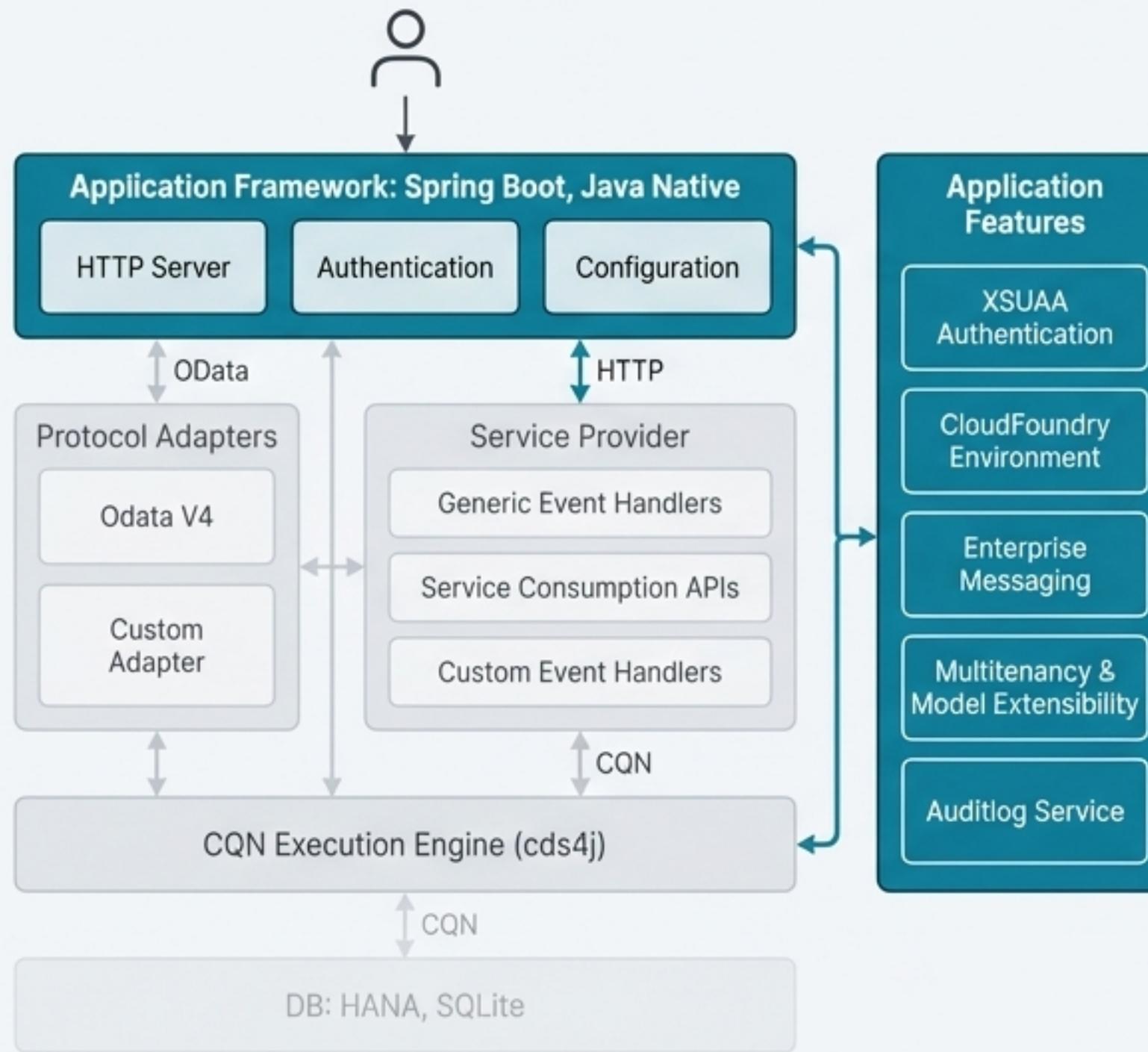
2 Etapa 2: Service Providers

Consomem os eventos CQN. Contêm os serviços genéricos autoexpostos pelo CAP, além de serviços técnicos como Persistência e Auditlog. Lógica de negócio customizada é implementada aqui via event handlers.

3 Etapa 3: CQN Execution Engine (cds4j)

O motor central que processa os eventos CQN e os traduz em declarações nativas para serem executadas em um serviço de persistência alvo, como SAP HANA ou H2.

A Base e as Extensões: Framework e Features



Escolhendo sua Fundação

Fornece a base da sua aplicação web, separando a lógica de negócio de tarefas comuns como processamento de endpoints HTTP. O CAP Java integra-se perfeitamente com Spring Boot (a primeira escolha), mas também suporta aplicações baseadas em Servlets Java ou frameworks de terceiros.

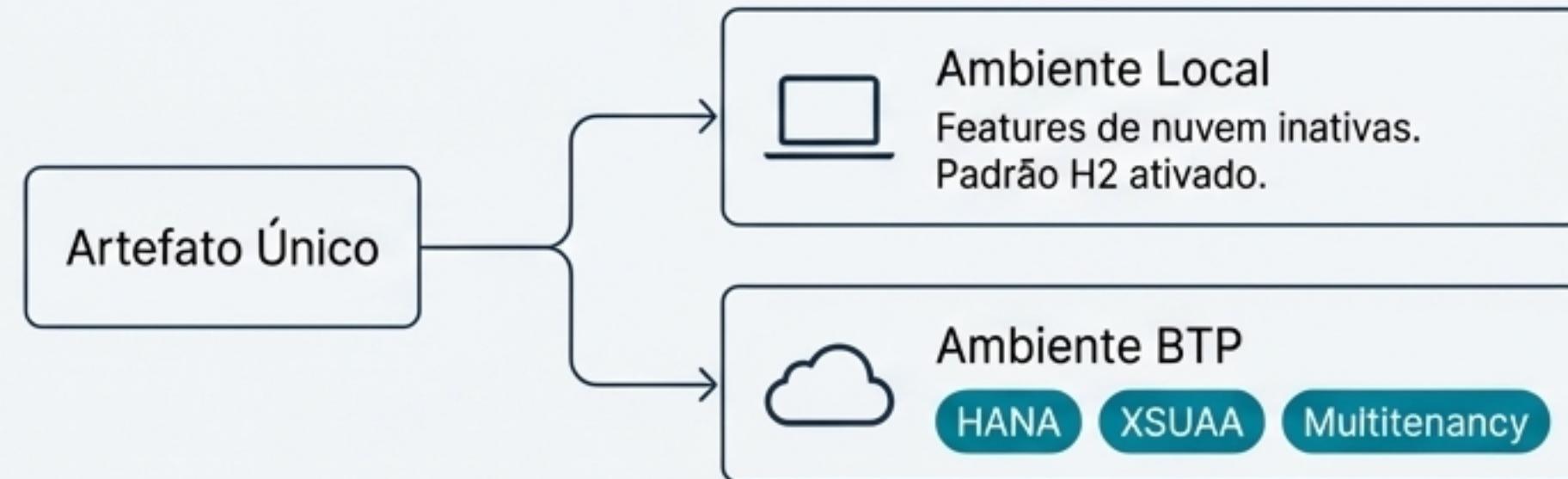
RECOMENDAÇÃO: Use `cds-framework-spring-boot` para uma rica integração out-of-the-box, injeção de dependência e auto-configuração.

O Poder dos Plugins

Módulos opcionais que adaptam o comportamento em tempo de execução. O CAP Java usa essa técnica para oferecer funcionalidades como integração com SAP Event Mesh e Audit Logging. Plugins customizados são carregados automaticamente pelo runtime.

Configure uma Vez, Adapte em Qualquer Lugar: A Inteligência das Features

Adicionar a dependência Maven de uma feature no momento da compilação habilita a aplicação a usá-la. Se a feature não encontrar o ambiente necessário em tempo de execução (ex: um binding para o SAP HANA), ela simplesmente não será ativada, e o stack recorre a uma implementação padrão para uso local (ex: H2).



`cds-feature-mt`: Torna a aplicação multitenant.

`cds-feature-xsuaa`: Adiciona autenticação baseada em XSUAA.

`cds-feature-enterprise-messaging`: Conecta ao SAP Event Mesh.

`cds-feature-cloudfoundry`: Prepara a aplicação para o ambiente SAP BTP, Cloud Foundry.

`cds-feature-k8s`: Adiciona suporte a Service Binding para SAP BTP, Kyma Runtime.

DICA: `cds-feature-cloudfoundry` e `cds-feature-k8s` podem ser combinados para criar binários que suportam ambos os ambientes.

Montando seu Projeto: A Fundação com Maven

Todos os módulos do CAP Java são artefatos Maven disponíveis no Maven Central. A configuração correta das dependências é a chave para montar a stack desejada.

Foco Principal: O Bill of Materials (BOM) - A Garantia de Consistência

```
<properties>
    <cds.services.version>2.6.0</cds.services.version>
</properties>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.sap.cds</groupId>
            <artifactId>cds-services-bom</artifactId> 1
            <version>${cds.services.version}</version>
            <type>pom</type>
            <scope>import</scope> 2
        </dependency>
    </dependencies>
</dependencyManagement>
```

1. O Bill of Materials: Um pom especial que centraliza e controla as versões de todos os artefatos CAP.

2. Importado no dependencyManagement: Garante que todas as dependências do CAP no seu projeto usarão versões consistentes e compatíveis entre si, evitando conflitos.

ATENÇÃO: Importar o `cds-services-bom` é a prática recomendada para garantir que as versões de todos os módulos CAP estejam em sincronia.

Traduzindo Arquitetura em Dependências

Exemplo de dependências para uma aplicação Spring Boot com endpoints OData V4.

```
<!-- Camada: Application Framework -->
<!-- Camada: Protocol Adapter -->
<!-- Camada: Core Runtime -->

<dependencies>
    <dependency>
        <groupId>com.sap.cds</groupId>
        <artifactId>cds-framework-spring-boot</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>com.sap.cds</groupId>
        <artifactId>cds-adapter-odata-v4</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>com.sap.cds</groupId>
        <artifactId>cds-services-api</artifactId>
        <!-- Scope 'compile' por padrão -->
    </dependency>
    <dependency>
        <groupId>com.sap.cds</groupId>
        <artifactId>cds-services-impl</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

Boas Práticas

Apenas módulos de API (ex: `cds-services-api`) devem ser adicionados sem um escopo definido. Todas as outras dependências devem ter um escopo explícito, como `runtime` ou `test`, para evitar uso indevido.

Acelere a Configuração com Starter Bundles

Para simplificar a configuração, o CAP Java oferece vários 'starters' que agrupam dependências para os casos de uso mais comuns.

- `cds-starter-spring-boot`: Agrupa todas as dependências para uma aplicação web baseada em Spring Boot.
- `cds-starter-cloudfoundry`: Inclui features para produção no SAP BTP, Cloud Foundry (XSUAA, SAP HANA, multitenancy).
- `cds-starter-k8s`: Inclui features para produção no SAP BTP, Kyma (similar ao de CF).

Configuração Manual

```
<dependencies>
  <dependency>
    <groupId>com.sap.cds</groupId>
    <artifactId>cds-feature-mt</artifactId>
  </dependency>
  <dependency>
    <groupId>com.sap.cds</groupId>
    <artifactId>cds-feature-xsuaa</artifactId>
  </dependency>
  <dependency>
    <groupId>com.sap.cds</groupId>
    <artifactId>cds-feature-hana</artifactId>
  </dependency>
</dependencies>
```

Com `cds-starter-cloudfoundry`

```
<dependencies>
  <dependency>
    <groupId>com.sap.cds</groupId>
    <artifactId>cds-starter-spring-boot</artifactId>
  </dependency>

  <dependency>
    <groupId>com.sap.cds</groupId>
    <artifactId>cds-adapter-odata-v4</artifactId>
    <scope>runtim</scope>
  </dependency>

  <dependency>
    <groupId>com.sap.cds</groupId>
    <artifactId>cds-starter-cloudfoundry</artifactId>
    <scope>runtim</scope>
  </dependency>

</dependencies>
```

Gerando a Estrutura do Projeto com Maven Archetype

```
mvn archetype:generate -DarchetypeArtifactId=cds-services-archetype \
-DarchetypeGroupId=com.sap.cds \
-DarchetypeVersion=[LATEST_VERSION]
```

O arquétipo 'cds-services-archetype' cria um projeto CAP Java completo. Personalize a geração inicial com opções na linha de comando:

- `-DincludeModel=true`: Adiciona um modelo CDS de exemplo minimalista.
- `-DodataVersion=[v2|v4]`: Especifica qual adaptador de protocolo é ativado por padrão (padrão: v4).
- `-DtargerPlatform=cloudfoundry`: Adiciona suporte à plataforma de destino Cloud Foundry.
- `-DinMemoryDatabase=[h2sqlite]`: Define o banco de dados em memória para testes locais (padrão: h2).
- `-DjdkVersion=[17|21]`: Especifica a versão alvo do JDK (padrão: 21).

Passo Seguinte: Após a geração, você pode construir e executar sua aplicação com o comando: `mvn spring-boot:run`

Refinando o Código: Geração de Acesso Tipado

O `cds-maven-plugin` pode gerar interfaces Java a partir do seu modelo CDS, permitindo acesso a dados com segurança de tipos e auto-complete na IDE.

A Escolha do Estilo

Decida o estilo das interfaces que melhor se adapta à sua equipe e projeto. A escolha é configurada na opção `methodStyle` do goal `generate`.

`BEAN` (Padrão)

Estilo inspirado na especificação JavaBeans, com prefixos `get` e `set`.

```
Authors author = Authors.create();
author.setName("Emily Brontë");

Books book = Books.create();
book.setAuthor(author);
book.setTitle("Wuthering Heights");
```

`FLUENT`

Métodos setter retornam a própria interface, permitindo encadeamento de chamadas (chaining).

```
Authors author = Authors.create().name("Emily
Brontë");

Books.create().author(author).title("Wuthering
Heights");
```

Controle Fino sobre a Geração de Código

O goal `generate` do `cds-maven-plugin` oferece múltiplas opções para customizar as interfaces geradas.

Pacote Base (`basePackage`)

Define um prefixo de pacote para todo o código gerado. Ex:

```
<basePackage>cds.gen</basePackage>.
```

Filtro de Entidades

(`includes` / `excludes`)

Permite especificar quais partes do seu modelo CDS devem ser incluídas ou excluídas da geração de código. Usa padrões como `my.bookshop.**`.

Filtros simples. Se partes incluídas referenciarem tipos de áreas excluídas, o código resultante não compilará.

Features de Geração

- `strictSetters`: Exige tipos concretos nos setters em vez de `Map<String, ?>`, melhorando a segurança de tipo na compilação.
- `betterNames`: Converte nomes de elementos CDS com caracteres especiais (como `/` ou ` `\$`) ou que conflitam com palavras-chave Java para nomes de métodos válidos (ex: `/DMO/GET_MATERIAL` vira `DmoGetMaterial`).

Gerenciando o Ambiente de Build: O `@sap/cds-dk`

O build do modelo CDS requer o `@sap/cds-dk`. A abordagem moderna e recomendada é gerenciá-lo como uma dependência de desenvolvimento do Node.js.

Abordagem Recomendada: `package.json` + `npm ci`

Passo 1: Configuração

Declare o `@sap/cds-dk` na seção `devDependencies` do seu `package.json`.

```
{  
  "devDependencies": {  
    "@sap/cds-dk": "^9"  
  }  
}
```

Passo 2: Execução

O `cds-maven-plugin` executa `npm ci` durante o build, usando o `package-lock.json` para instalar a versão exata do `cds-dk` e suas dependências, garantindo builds reproduzíveis.

- ⓘ Esta é a configuração padrão para projetos criados com `cds-services-archetype` a partir da versão 3.6.0.

⚠️ Para aplicações multitenant, garanta que a versão do `@sap/cds-dk` no `sidecar` esteja em sincronia.

Abordagem Antiga (Depreciada)

Projetos mais antigos podem usar o goal `install-cdssdk`. Recomenda-se a migração para a abordagem com `npm ci`.

Otimizando seu Fluxo de Trabalho de Build

Por padrão, o build é **autossuficiente**, baixando uma instância local do Node.js e do `@sap/cds-dk`. Para acelerar o processo, você pode utilizar as ferramentas globais já instaladas na sua máquina.



Build Autossuficiente (Padrão)

Garante a reproduzibilidade do build em qualquer máquina, sem pré-requisitos externos. Ideal para pipelines de CI/CD.

```
mvn spring-boot:run
```

Velocidade

Build Acelerado (Perfil `cdsdk-global`)

Omite o download e a instalação local, usando a instalação global do Node.js e do `@sap/cds-dk`. Significativamente mais rápido para desenvolvimento local.

Pré-requisitos:

1. `@sap/cds-dk` instalado globalmente (`npm i -g @sap/cds-dk`).
2. Node.js disponível no PATH do sistema.

```
mvn spring-boot:run -P cdsdk-global
```