

# Construindo Aplicações Corporativas com CAP Java

Dos Fundamentos à Produção: Um Guia Arquitetural

```
@Entity  
@Table(name = "Business_Object")  
public class ApplicationService {  
    ...  
  
    @Transient  
    public class ApplicationService {  
        return persistence.save(data);  
    }  
  
    public void etst( Date date {  
        return persistence.save(meetService.data);  
    }  
}
```

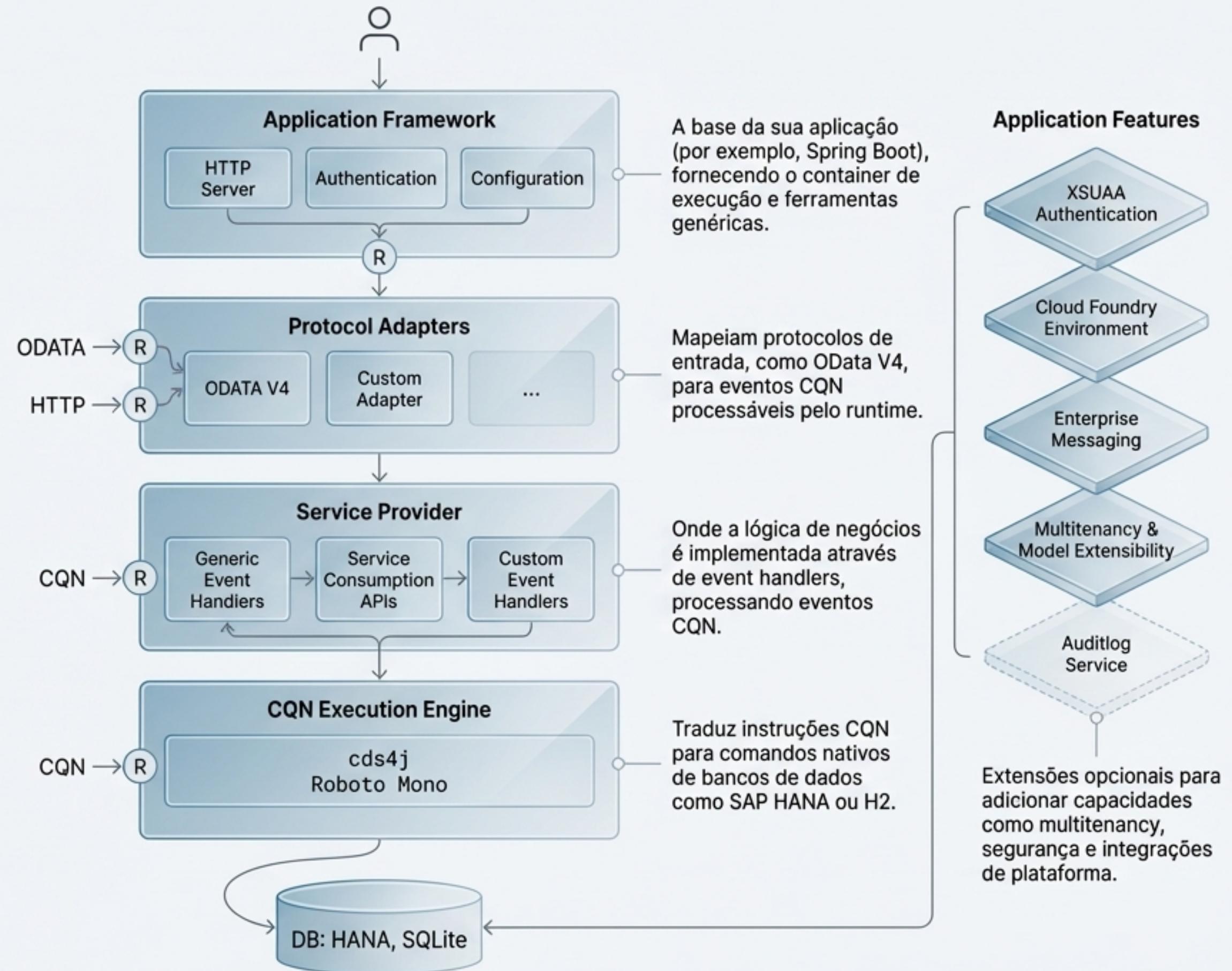


# A Arquitetura Modular do CAP Java: Uma Visão Geral

O CAP Java é projetado com uma arquitetura de pilha modular, separando funcionalidades em componentes independentes e intercambiáveis.

Este design oferece flexibilidade, simplifica testes e permite que as aplicações sejam agnósticas em relação à plataforma e aos serviços.

Vamos explorar as cinco principais áreas da pilha, que servem como os blocos de construção da nossa aplicação.



# Estabelecendo a Base do Projeto com Versões e Dependências

## Versionamento Semântico e Ciclo de Vida

O CAP Java segue o Versionamento Semântico (MAJOR.MINOR.PATCH).

- **Major** (#0A64A4): Lançadas anualmente ou mais, podem introduzir mudanças incompatíveis (ex: 2.0.0).
- **Minor** (#0A64A4): Lançadas mensalmente, trazem novas funcionalidades (ex: 2.7.0).
- **Patch** (#0A64A4): Lançadas sob demanda para correções críticas, sem novas funcionalidades (ex: 2.7.1).

Recomendamos fortemente consumir a última versão minor mensalmente para minimizar o esforço de migração futura.

## Codelines e Dependências Mínimas

- **Active Codeline** (#0A64A4): Recebe novas features e patches. A versão atual é a 'ativa'.
- **Maintenance Codeline** (#0A64A4): A versão major anterior entra em modo de manutenção, recebendo apenas patches críticos.
- **Dependências Chave** (#0A64A4): O CAP Java exige versões mínimas para dependências críticas, mas a recomendação é sempre usar as mais recentes.

JDK:	Mínimo 17	Recomendado 21
Spring Boot:	Mínimo 3.0	Recomendado o último 3.x
Node.js:	Mínimo 20	Recomendado 22

*Manter as dependências atualizadas garante acesso a correções de segurança e novas funcionalidades, mantendo o projeto saudável e seguro.*

# Garantindo a Consistência com Bill of Materials (BOM)

Misturar diferentes versões de artefatos de um mesmo SDK (como CAP Java ou Cloud SDK) pode causar erros de compilação e comportamento imprevisível em tempo de execução.

Para garantir a consistência, os SDKs fornecem 'Bill of Materials' (BOMs). Recomendamos fortemente importar os BOMs no `dependencyManagement` do seu `pom.xml`.

```
// pom.xml
<dependencyManagement>
  <dependencies>
    <!-- Garante versões consistentes para todos os artefatos do CAP Java -->
    <dependency>
      <groupId>com.sap.cds</groupId>
      <artifactId>cds-services-bom</artifactId>
      <version>${cds.services.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <!-- Alinha as versões para o SAP Cloud SDK -->
    <dependency>
      <groupId>com.sap.cloud.sdk</groupId>
      <artifactId>sdk-modules-bom</artifactId>
      <version>${cloud.sdk.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <!-- Gerencia dependências de segurança, como XSUAA -->
    <dependency>
      <groupId>com.sap.cloud.security</groupId>
      <artifactId>java-bom</artifactId>
      <version>${xsuaa.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
```

Garante versões consistentes para todos os artefatos do CAP Java

Alinha as versões para o SAP Cloud SDK

Gerencia dependências de segurança, como XSUAA

Em caso de uma vulnerabilidade de segurança descoberta entre os releases do CAP Java, você pode especificar explicitamente uma versão segura no início da seção `dependencyManagement` do seu POM.

# O Coração da Aplicação: Introspecção do Modelo CDS com a Reflection API

O modelo CDS é a fonte única da verdade para a estrutura da sua aplicação. A **Model Reflection API** fornece um conjunto de interfaces para introspectar programaticamente esse modelo. O ponto de partida é a interface `CdsModel`, que representa o modelo CDS completo.

## Acessando o `CdsModel`

O `CdsModel` pode ser obtido a partir do `EventContext` dentro de um handler ou injetado diretamente como um bean Spring.

### Snippet de Código (Obtenção via `EventContext`):

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

### Snippet de Código (Injeção via Spring):

```
// Injeção de dependência
@Autowired
CdsModel model;
```

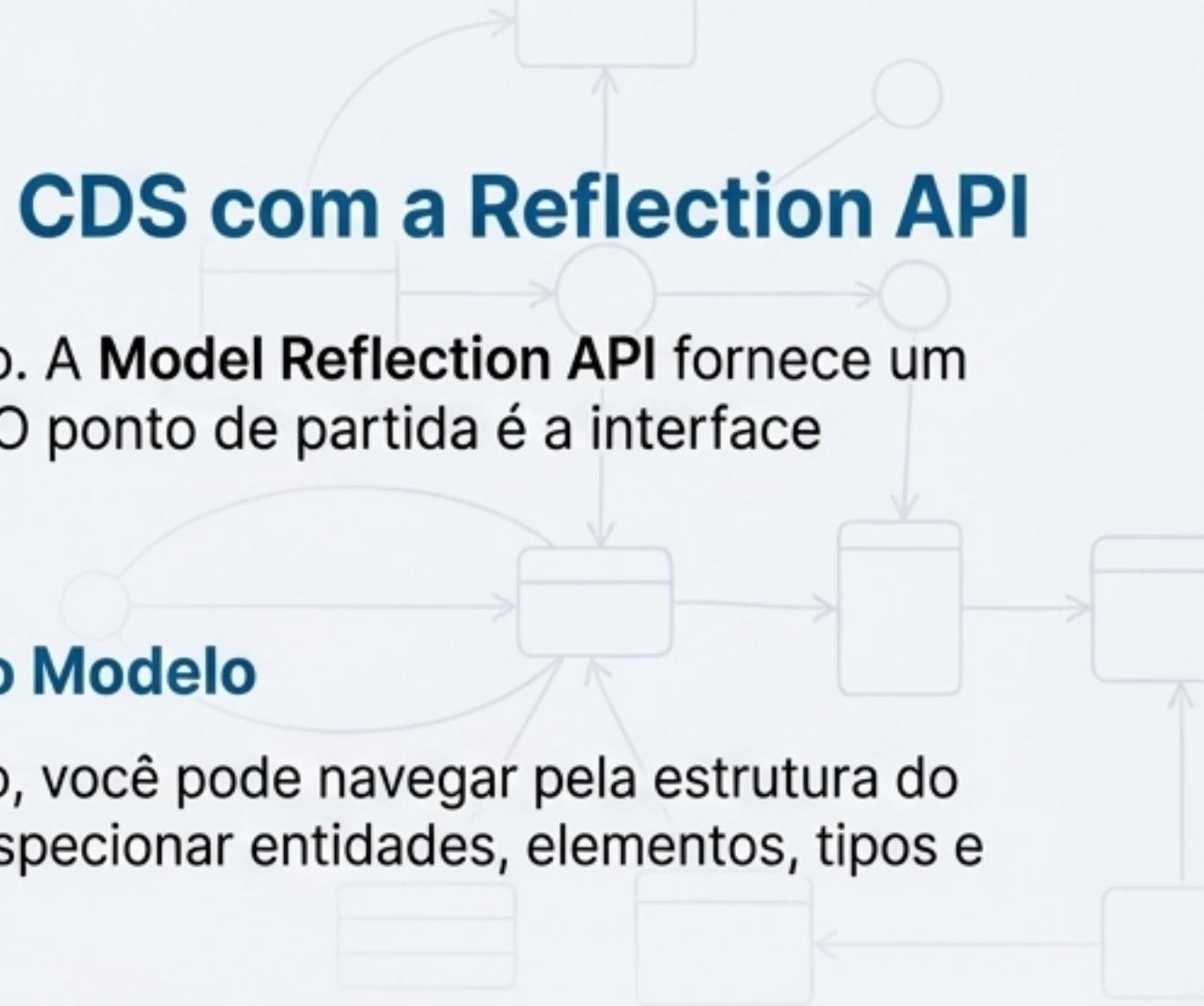
## Explorando o Modelo

Uma vez obtido, você pode navegar pela estrutura do modelo para inspecionar entidades, elementos, tipos e anotações.

### Snippet de Código (Inspecionando um Elemento):

```
// Exemplo: Inspecionando o elemento 'title' da entidade 'Books'
CdsEntity books = model.getEntity("my.bookshop.Books");
CdsElement title = books.getElement("title");

boolean localized = title.isLocalized(); // true
CdsSimpleType simpleType = title.getType().as(CdsSimpleType.class)
Integer length = simpleType.get("length"); // 111
```



## Dando Vida ao Modelo com Serviços Baseados em CQN

Serviços são um conceito central no CAP. Eles expõem a lógica de negócios através de uma API baseada em eventos e processam consultas usando o CAP Query Notation (CQN). Existem três tipos principais de serviços baseados em CQN, cada um com um propósito específico.



### `Application Services`

Definem as APIs expostas pela sua aplicação aos clientes. São baseados em uma definição de serviço no modelo CDS e geralmente atendidos por adaptadores de protocolo como OData. Delegam, por padrão, as operações para o Persistence Service.



### `Persistence Services`

São clientes de banco de dados que aceitam consultas CQN. O CAP Java fornece uma implementação JDBC para bancos de dados SQL (SAP HANA, H2) pronta para uso. Gerenciam transações e dados de forma transparente.



### `Remote Services`

São clientes para APIs remotas (por exemplo, OData V2/V4). Permitem integrar sua aplicação com outros microsserviços de forma síncrona ou assíncrona, traduzindo chamadas CQN para o protocolo remoto.

# Implementando Lógica de Negócios com Event Handlers

No CAP, tudo o que acontece em tempo de execução é um evento enviado a um serviço. Event Handlers são métodos Java que permitem estender ou sobrescrever o processamento desses eventos para adicionar sua lógica de negócios customizada.



## Snippet de Código Exemplo

```
@Component  
@ServiceName("AdminService")  
public class AdminServiceHandler implements EventHandler {  
  
    @Before(event = "CREATE", entity = Books_.CDS_NAME)  
    public void validateBook(Books book) {  
        // Lógica de validação antes da criação  
    }  
}
```

```
@On(event = "myCustomAction")  
public void performAction(MyCustomActionContext context) {  
    // Lógica principal da ação  
    context setResult("Action completed!");  
}  
  
@After(event = "READ", entity = Books_.CDS_NAME)  
public void enrichReadResult(List<Books> books) {  
    // Enriquece os dados após a leitura  
}
```

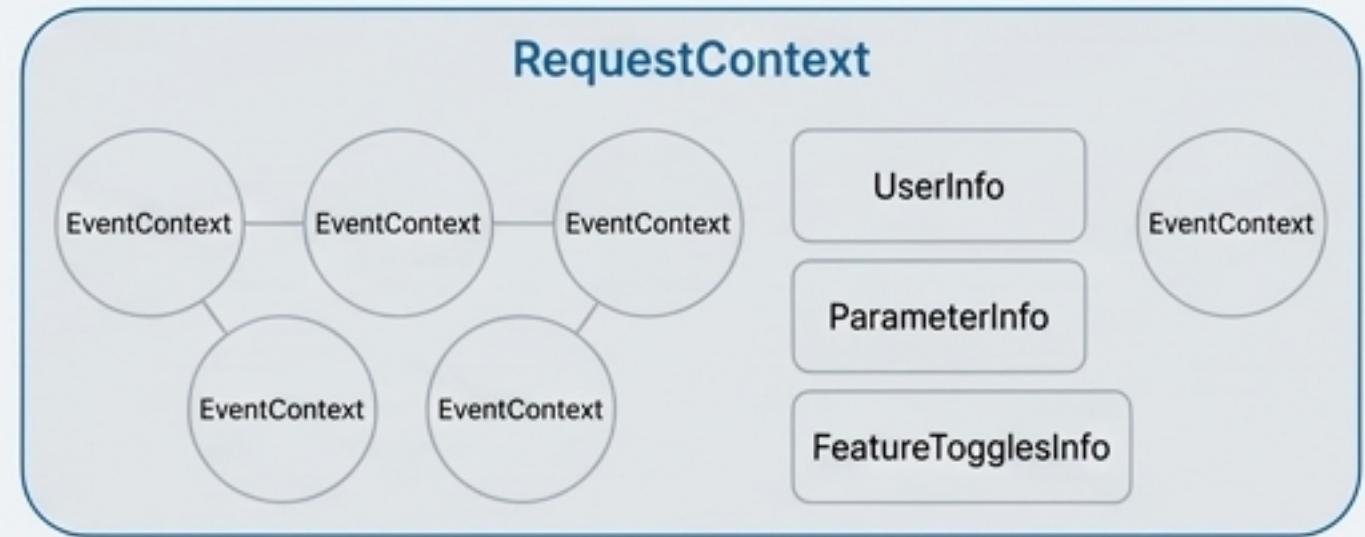
# Orquestrando o Fluxo de Execução com `RequestContext`

## Conceito Central

``RequestContext`` abrange a execução de múltiplos eventos (por exemplo, dentro de uma única requisição HTTP), fornecendo um contexto comum.

Ele carrega informações essenciais como ``UserInfo`` (usuário, tenant, roles), ``ParameterInfo`` (headers, locale) e ``FeatureTogglesInfo``.

Enquanto o ``EventContext`` é específico para um único evento, o ``RequestContext`` é mantido como um ``thread-local``, disponível para todos os serviços envolvidos.



## Acessando o Contexto

Você pode acessar as informações do contexto através do ``EventContext`` ou injetando-as diretamente com Spring.

### Snippet de Código (Injeção)

```
@Autowired  
private UserInfo userInfo;  
  
@Autowired  
private ParameterInfo parameterInfo;  
  
@Before(event = CqnService.EVENT_READ)  
public void beforeRead() {  
    boolean isAuthenticated = userInfo.isAuthenticated();  
    String tenant = userInfo.getTenant();  
    Locale locale = parameterInfo.getLocale();  
    // ...  
}
```

## Trocando de Contexto com ``RequestContextRunner``

Para cenários avançados, como jobs assíncronos ou chamadas a serviços técnicos, é necessário trocar ou definir um contexto.

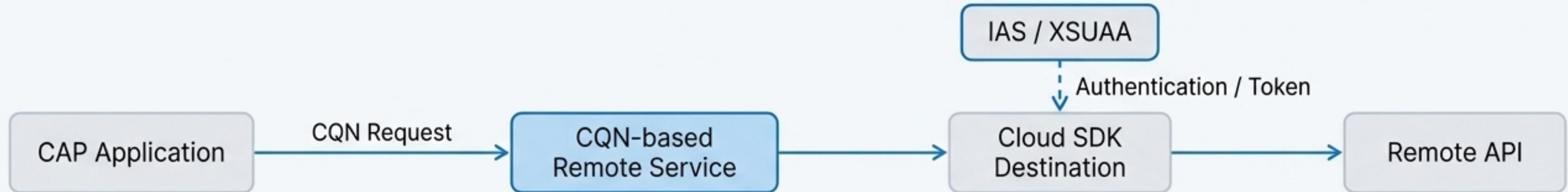
A API ``RequestContextRunner`` oferece métodos fluentes para isso.

Executar uma operação como um usuário técnico do tenant provedor.

```
@Autowired  
private CdsRuntime runtime;  
  
public void callTechnicalService() {  
    runtime.requestContext().systemUserProvider().run(reqContext -> {  
        // Este bloco de código executa em um novo RequestContext  
        // com um usuário técnico do tenant provedor.  
        // ... chamada ao serviço técnico ...  
    });  
}
```

# Conectando-se ao Ecossistema: Integração com `Remote Services`

`Remote Services` são clientes CQN para APIs remotas (OData V2/V4), permitindo que sua aplicação consuma outros serviços de forma transparente. O CAP Java utiliza o SAP Cloud SDK para gerenciar destinos (destinations) e conectividade, abstraindo detalhes de autenticação e propagação de usuário/tenant.



## Configuração via `application.yaml`

A configuração é declarativa. Você define o serviço remoto, seu tipo e o destino (destination) que contém os detalhes da conexão.

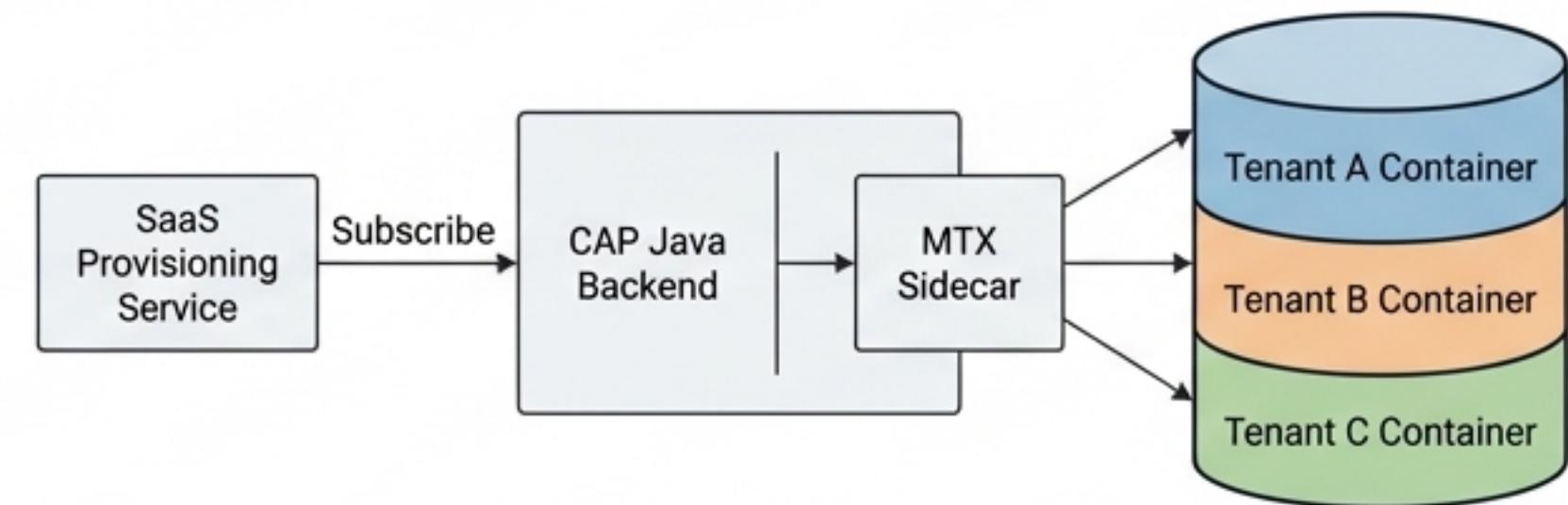
```
cds:  
  remote.services:  
    # Nome do serviço remoto usado no código  
    API_BUSINESS_PARTNER:  
      # Tipo do protocolo da API remota  
      type: "odata-v2"  
      # Configuração do destino via Cloud SDK  
      destination:  
        # Nome do destino no BTP Destination Service  
        name: "s4-business-partner-api"
```

A recomendação clara do CAP é usar `Remote Services` em vez de usar o SAP Cloud SDK diretamente, pois ele simplifica o código e mantém a extensibilidade.

# Escalando para Múltiplos Clientes com Multitenancy

## Visão Geral da Arquitetura

O CAP Java oferece suporte robusto para aplicações SaaS (Software as a Service), isolando os dados de múltiplos clientes (tenants) em containers de banco de dados dedicados. O MTX Sidecar é um componente chave que gerencia o ciclo de vida dos tenants, incluindo provisionamento de containers, extensões de modelo e feature toggles.



## Customizando o Ciclo de Vida do Tenant

O DeploymentService emite eventos que permitem customizar o processo de subscrição e cancelamento.

### Eventos Principais

- **SUBSCRIBE:** Disparado ao adicionar um novo tenant. Permite, por exemplo, especificar o database\_id se houver múltiplas instâncias de banco de dados.
- **UNSUBSCRIBE:** Disparado ao remover um tenant. Pode ser usado para executar lógicas de limpeza ou evitar a exclusão dos dados.
- **UPGRADE:** Disparado para aplicar atualizações de esquema de banco de dados em todos os tenants.

```

@Before(event = DeploymentService.EVENT_SUBSCRIBE)
public void beforeSubscription(SubscribeEventContext context) {
    // Se houver mais de uma instância de SAP HANA,
    // é necessário especificar qual usar.
    context.getOptions().put("provisioningParameters",
        Collections.singletonMap("database_id", "<your-database-id>"));
}
  
```

# Aprimorando a Experiência do Usuário com Fiori Drafts

## Conceito

O CAP oferece suporte nativo ao padrão de UI SAP Fiori Drafts, permitindo que os usuários salvem alterações em andamento sem afetar os dados ativos.

Quando uma entidade é habilitada para draft, o `DraftService` orquestra a leitura e escrita, mesclando dados da entidade ativa com os dados do rascunho.

## Fluxo de Eventos de Draft



## Adicionando Lógica Customizada

Você pode registrar event handlers para os eventos de draft para adicionar validações ou preencher valores padrão.

```
// Handler para preencher valores
// padrão em um novo rascunho
@On(event = DraftService.EVENT_DRAFT_CREATE, entity =
OrderItems_.CDS_NAME)
public void prefillDraftOrder(OrderItems orderItem) {
    // Lógica para preencher campos
    // com valores padrão
    if (orderItem.getQuantity() == null) {
        orderItem.setQuantity(1);
    }
}
```

A entidade ativa fica bloqueada para edição por outros usuários enquanto um rascunho existir, garantindo a integridade dos dados.

# Gerenciando Funcionalidades Dinâmicas com Feature Toggles

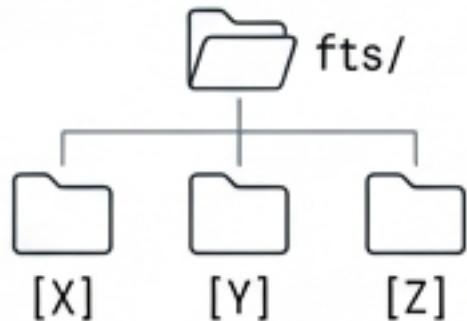
## Conceito

Feature Toggles permitem habilitar ou desabilitar partes de uma aplicação dinamicamente em tempo de execução, sem a necessidade de um novo deploy. Isso pode ser baseado no usuário, tenant, role ou qualquer outra informação da requisição.

## Arquitetura

### Estrutura de Arquivos

As features são definidas como modelos CDS em subpastas de um diretório `fts` no seu projeto. O nome da subpasta é o nome da feature.



### Runtime

A cada requisição, o `Model Provider Service` compila um modelo CDS efetivo que inclui apenas as features ativas para aquele contexto, garantindo que a lógica da aplicação e o acesso ao banco de dados respeitem os toggles.

## Implementando um Provedor Personalizado

O conjunto de features ativas é determinado por um `FeatureTogglesInfoProvider`. Você pode implementar um provedor personalizado para definir sua própria lógica de ativação.

```
@Component
public class RoleBasedFeatureProvider implements FeatureTogglesInfoProvider {

    @Override
    public FeatureTogglesInfo get(UserInfo userInfo, ParameterInfo paramInfo) {
        Map<String, Boolean> toggles = new HashMap<>();
        // Habilita a feature 'isbn' se o usuário tiver a role 'expert'
        if (userInfo.hasRole("expert")) {
            toggles.put("isbn", true);
        }
        return FeatureTogglesInfo.create(toggles);
    }
}
```

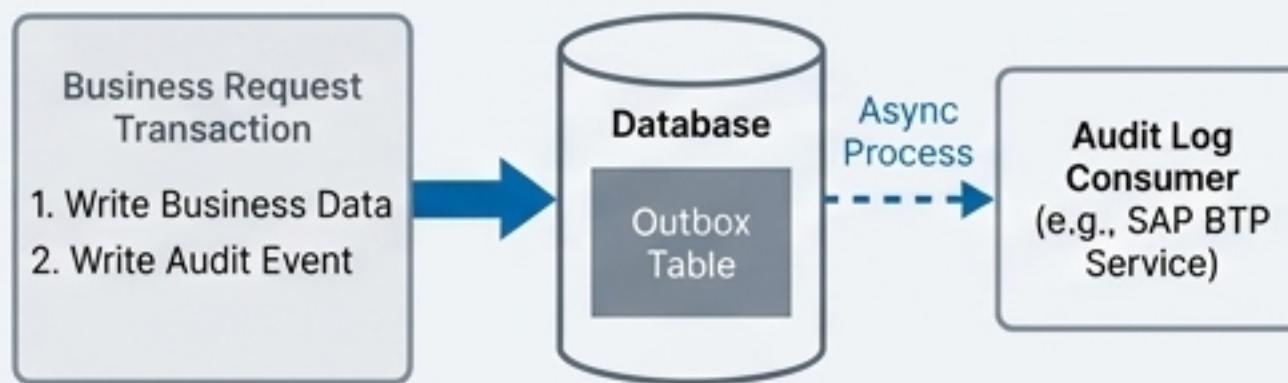
Usando Feature Toggles, você pode até mesmo controlar a visibilidade de elementos de UI em aplicações SAP Fiori Elements, já que a UI é definida por anotações no modelo CDS.

# Garantindo Conformidade e Rastreabilidade com Audit Logging

## Introdução

O `AuditLogService` fornece uma maneira padronizada de emitir eventos de auditoria para acesso a dados pessoais, modificações, alterações de configuração e eventos de segurança.

## Funcionamento Assíncrono e Transacional



Eventos de auditoria são armazenados na mesma transação da requisição de negócio. Se a transação falhar, os logs de auditoria também são revertidos, garantindo consistência.

Um processo assíncrono posterior envia os eventos para um consumidor central, como o serviço de AuditLog do SAP BTP.

## Utilizando o `AuditLogService`

O serviço pode ser injetado em qualquer bean Spring para ser utilizado na sua lógica de negócios.

```
@Autowired  
private AuditLogService auditLogService;  
  
public void modifyPersonalData(List<DataModification>  
modifications) {  
    // Lógica de negócio...  
  
    // Emite um evento de modificação de dados  
    auditLogService.logDataModification(modifications);  
}  
  
public void logLoginAttempt(String user) {  
    // Emite um evento de segurança  
    auditLogService.logSecurityEvent("login", user);  
}
```

**Handlers**: O CAP Java fornece handlers prontos que escrevem no log da aplicação (para desenvolvimento) ou se integram com o serviço de AuditLog v2 do SAP BTP (para produção).

# Estendendo o Framework: Construindo Plugins Modulares e Reutilizáveis

A arquitetura modular do CAP Java permite estender suas capacidades com código customizado e reutilizável. Você pode empacotar e compartilhar funcionalidades através de artefatos Maven.

## Compartilhando Modelos CDS

Você pode empacotar modelos CDS, dados CSV e arquivos i18n em um artefato Maven.

Consumidores simplesmente adicionam a dependência ao seu `pom.xml` e usam o goal `resolve` do `cds-maven-plugin` para disponibilizar os modelos.

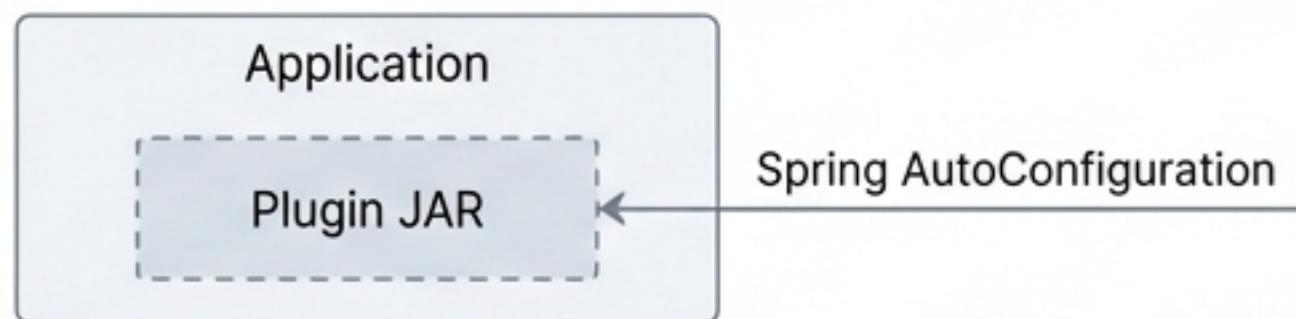
```
<dependency>
  <groupId>com.sap.capiре</groupId>
  <artifactId>bookshop-models</artifactId>
  <version>1.0.0</version>
</dependency>
<build>
  <plugins>
    <plugin>
      <groupId>com.sap.cds</groupId>
      <artifactId>cds-maven-plugin</artifactId>
      <version>5(cds.version)</version>
      <executions>
        <execution>
          <id>resolve</id>
          <goals>
            <goal>resolve</goal>
          </goals>
        </execution>
      </plugin>
    </plugins>
  </build>
```

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

## Empacotando Event Handlers

É possível criar bibliotecas com `EventHandlers` genéricos que podem ser automaticamente registrados em qualquer aplicação CAP Java.

A descoberta e registro pode ser feita via **Spring AutoConfiguration** (recomendado para handlers que precisam de injeção de dependência) ou via mecanismo `ServiceLoader` do Java para plugins mais simples.

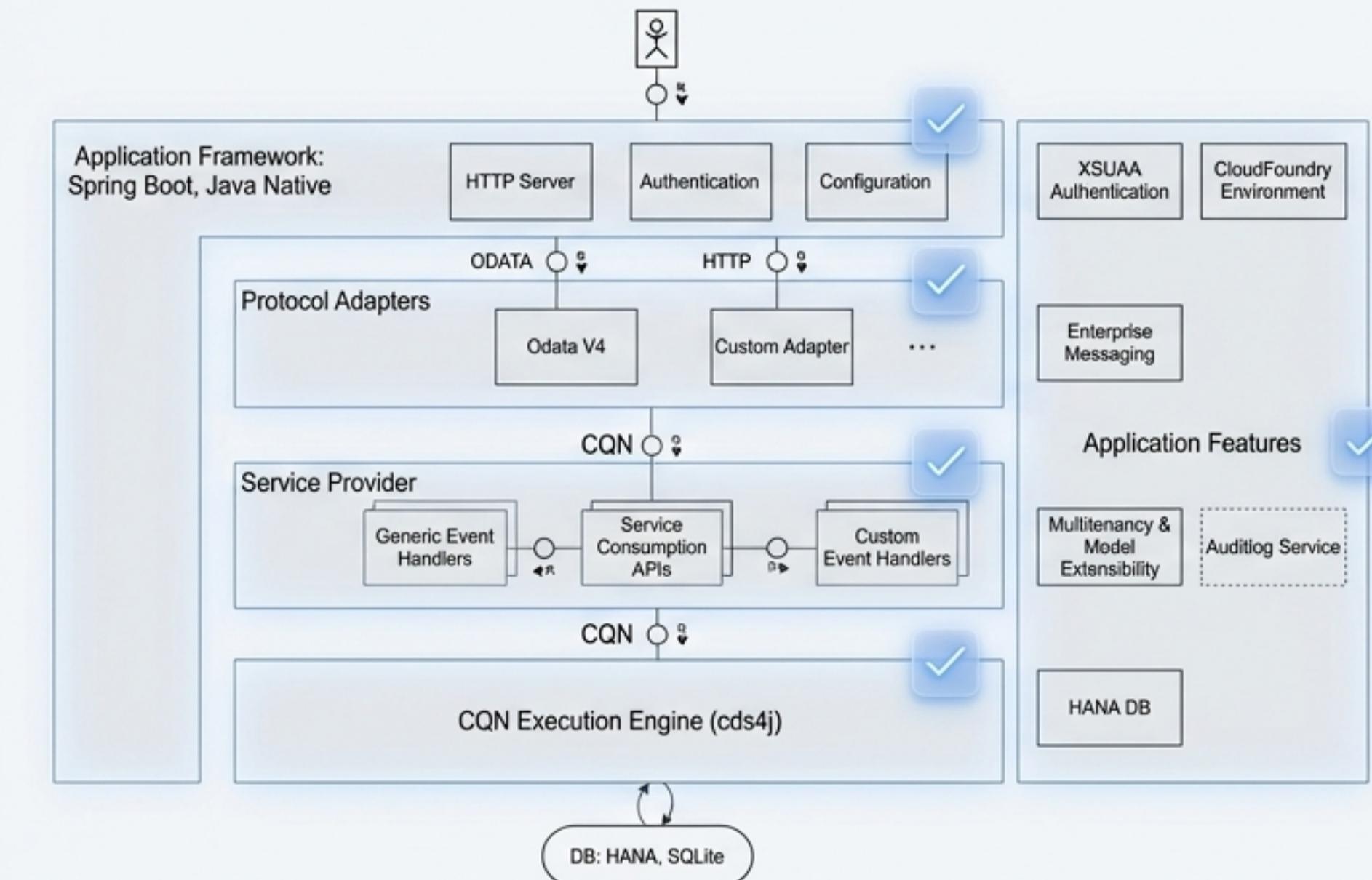


Plugins permitem a criação de ecossistemas de funcionalidades, promovendo a reutilização de código e a padronização em múltiplos projetos.

# CAP Java: Uma Arquitetura Completa para Aplicações Corporativas

Partindo de uma base sólida de gerenciamento de dependências e um modelo de dados bem definido, adicionam-se camadas de lógica de negócios, integrações e capacidades corporativas avançadas.

Cada componente, do `PersistenceService` ao `AuditLogService`, funciona como um bloco de construção em uma arquitetura coesa e modular.



O CAP Java oferece uma pilha opinativa e ao mesmo tempo aberta, combinando as melhores práticas da indústria com a flexibilidade para estender e customizar. Ele fornece uma base sólida e escalável para construir e evoluir aplicações prontas para o futuro.