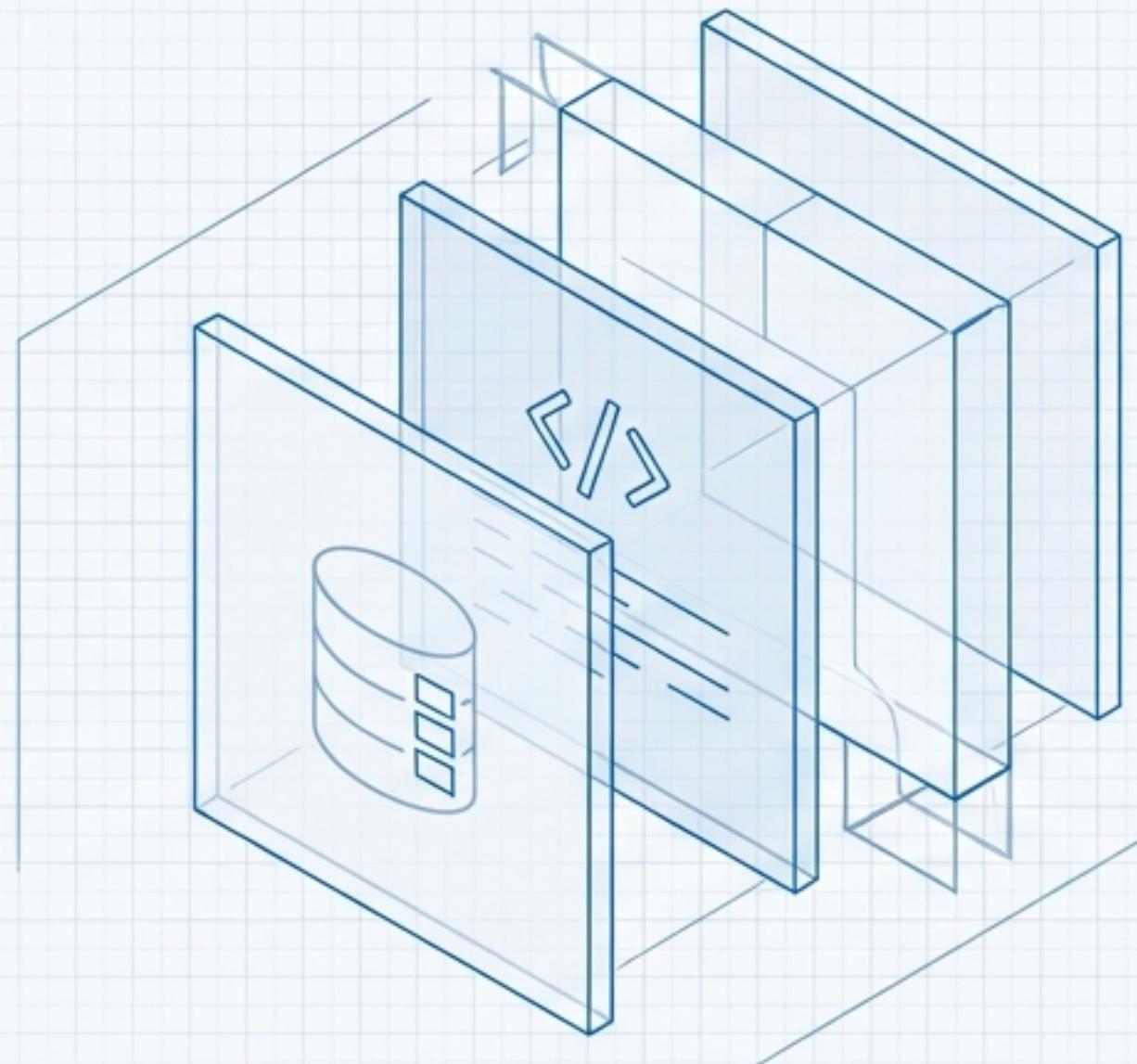


# **Dominando a Persistência de Dados no CAP Java**

## **Um Guia do Arquiteto para Persistence Services**

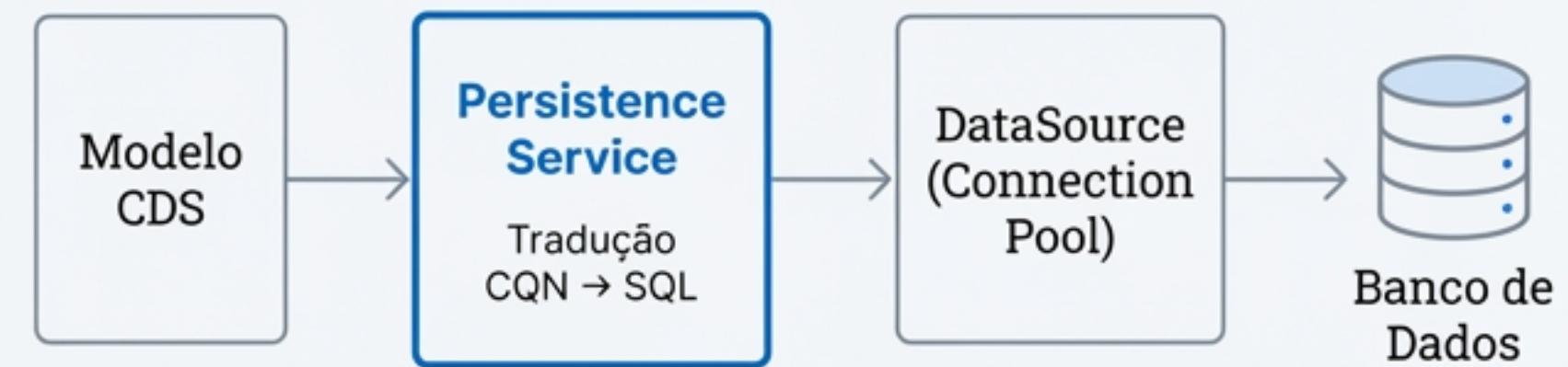


Para desenvolvedores Java que buscam extrair o máximo do SAP Cloud Application Programming Model.

# O Que é um Persistence Service?

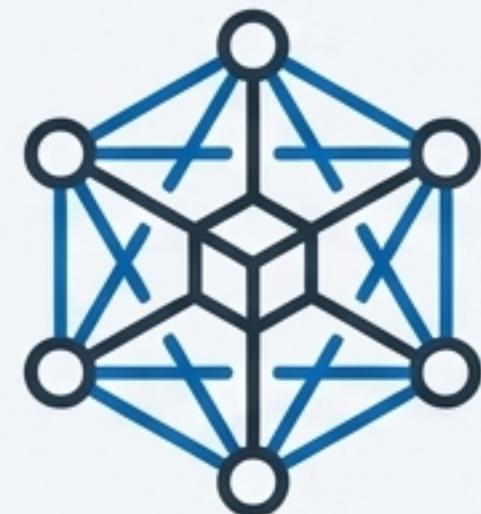
Persistence Services são clientes de banco de dados baseados em CQN (CDS Query Notation). Eles atuam como uma camada de abstração sobre um `DataSource` JDBC, com duas funções principais:

1. **Tradução de CQN para SQL**: Converte as consultas do modelo CDS para o dialeto SQL nativo do banco de dados.
2. **Gerenciamento de Transações**: Cuida da inicialização e manutenção de transações de banco de dados como parte do contexto de `ChangeSet` ativo.



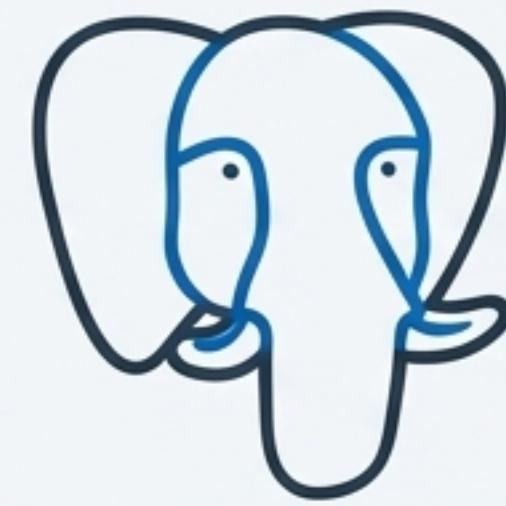
# O Ecossistema de Bancos de Dados Suportados

O CAP Java oferece suporte integrado para diversos tipos de banco de dados. A escolha correta depende do seu ambiente (desenvolvimento ou produção) e dos requisitos da sua aplicação.



## SAP HANA

O padrão para uso produtivo, com suporte completo a evolução de schema e multitenancy.



## PostgreSQL

Uma alternativa robusta para produção, com algumas limitações específicas.



## H2

Recomendado para desenvolvimento local e testes de CI. Não é para uso empresarial.



## SQLite

Recomendado estritamente para desenvolvimento e testes, especialmente em cenários mais simples.

# A Decisão Estratégica: Comparativo dos Bancos de Dados

Feature / Característica	SAP HANA Cloud	PostgreSQL	H2	SQLite
Caso de Uso Ideal	Produção Enterprise	Produção / Dev Avançado	Desenvolvimento / Testes	Desenvolvimento / Testes
Evolução de Schema	Completo	Limitado	N/A	N/A
Multitenancy	Completo	Não Suportado	N/A	N/A
Ordenação por Localidade	Completo (Configurável)	Depende do DB	Padrão ASCII	Case-insensitive
Views Atualizáveis	Avançado	Suporte Básico	Suporte CAP	Suporte CAP
Locking Pessimista	Completo	N/A	Locks compartilhados viram exclusivos	Não Suportado

# Foco em Produção: SAP HANA Cloud

A opção padrão do CAP para aplicações produtivas que exigem escalabilidade, multitenancy e evolução de schema.

## Performance de Ordenação

A ordenação por localidade em elementos `String` pode impactar a performance. Desabilite quando não for necessário.

## Modo de Otimização SQL

Por padrão, o CAP gera SQL otimizado para o motor HEX do HANA Cloud. Para compatibilidade com HANA HANA 2.x, use o modo legado.

## Associações Nativas

Recomenda-se desabilitar a geração de associações no DDL, pois não são utilizadas pelo CAP Java.

CDS	YAML
isbn: String(40) @cds.collate: false;	cds.sql.hana. ignoreLocale: true

YAML
cds.sql.hana.optimizationMode: legacy

JSON
{ "sql": { "native_hana_associations": false } }

# Alternativa Robusta: PostgreSQL

Suportado no CAP Java (testado na v15), o PostgreSQL é uma excelente opção para muitos cenários. Contudo, é crucial estar ciente das suas limitações atuais.

## Limitações Conhecidas

- 1 Sem Ordenação por Localidade:** O comportamento da ordenação (`sort`) depende exclusivamente da configuração do banco de dados.
- 2 Views Atualizáveis Limitadas:** Apenas views que o próprio PostgreSQL considera atualizáveis ou que o runtime do CAP consegue resolver são suportadas para escrita.
- 3 Sem Suporte a Multitenancy e Extensibilidade:** Funcionalidades essenciais para SaaS ainda não estão disponíveis.



**\*\*Nota de Produção\*\*:** A implantação automática de schema não é recomendada para produção. Utilize ferramentas como **Flyway** ou **Liquibase**.

# Para Desenvolvimento e Testes: H2 e SQLite

## H2

### Finalidade

Ideal para desenvolvimento local e testes de CI (em memória ou baseado em arquivo).

### Limitações Importantes

- Collation apenas a nível de banco de dados (padrão ASCII).
- Locks compartilhados são substituídos por locks exclusivos.
- Não suporta checagem de constraints deferida (`deferred checking`).



Requer H2 v2.2.x ou superior para suporte a dados localizados e temporais via variáveis de contexto de sessão.

## SQLite

### Finalidade

Simples e rápido para desenvolvimento local.

### Configuração Crítica

Limite o pool de conexões a 1 para serializar transações e evitar problemas de concorrência.

```
hikari:  
    maximum-pool-size: 1
```

### Limitações Importantes

- Não suporta locking pessimista.
- Streaming de LOBs (cds.LargeString, cds.LargeBinary) pode impactar a performance.
- Views são somente leitura (com suporte do CAP para atualização de algumas).

# O Blueprint da Conexão: Configurando DataSources

O CAP Java se integra ao ecossistema Java padrão, utilizando `java.sql.DataSource` para se conectar aos bancos. A configuração pode ser automática (via service bindings) ou explícita (via Spring Boot).

## Gerenciamento de Conexões com Connection Pools

Pools Suportados:

- **Single-tenant:** `hikari`, `tomcat`, `dbcp2`
- **Multitenant:** `hikari`, `tomcat`, `atomikos`

É necessário ter a dependência do pool no classpath.

cds:

dataSource:

<service-instance-name>: ←

hikari:

maximum-pool-size: 20

data-source-properties:

packetSize: 300000 # Propriedade do driver

Nome do seu  
service binding

# Configuração na Prática: SAP HANA e PostgreSQL

## SAP HANA (via Service Binding)

### Mecanismo

Autoconfigurado a partir de `VCAP\_SERVICES` (Cloud) ou `default-env.json` (Local).

### Precedência

Uma configuração explícita de `spring.datasource.url` sempre terá prioridade sobre os service bindings.

## PostgreSQL (via Configuração Explícita)

### Passo 1: Adicionar Suporte

```
cds add postgres
```

### Passo 2: Gerar Schema SQL (Maven)

```
<execution>
  <id>schema.sql</id>
  <goals><goal>cds</goal></goals>
  <configuration>
    <commands>
      <command>deploy --to postgres --dry --out ...</comm
    </commands>
  </configuration>
</execution>
```

### Passo 3: Configurar Conexão

```
spring:
  config.activate.on-profile: postgres
  datasource:
    url: <url>
    username: <user>
    password: <password>
    driver-class-name: org.postgresql.Driver
```

# Configuração para Desenvolvimento: H2 e SQLite

## H2 (Em Memória)

### Mecanismo

O Spring Boot autoconfigura um banco H2 em memória se ele estiver no classpath.

### Geração de Schema

Use o dialeto `h2` no comando `cds deploy`.

```
cds deploy --to h2 --dry
```

## SQLite (Em Memória vs. Arquivo)

### \*\*Baseado em Arquivo\*\*

O schema é gerenciado pelo `cds deploy`, então a inicialização do Spring deve ser desabilitada.

```
spring:  
  config.activate.on-profile: sqlite  
  sql.init.mode: never  
  datasource:  
    url: "jdbc:sqlite:sqlite.db"  
    hikari:  
      maximum-pool-size: 1
```

### \*\*Em Memória\*\*

O Spring gerencia o schema. Para evitar que o Hikari destrua o banco ao fechar a última conexão, configure o `max-lifetime`.

```
spring:  
  config.activate.on-profile: default  
  sql.init.mode: always  
  datasource:  
    url: "jdbc:sqlite:file::memory:?cache=shared"  
    hikari:  
      maximum-pool-size: 1  
      max-lifetime: 0
```

# Arquitetura com Múltiplos Bancos de Dados

## O Desafio

Uma aplicação com múltiplos `DataSources` (por exemplo, múltiplos service bindings) falhará na inicialização. O CAP Java precisa saber qual deles deve ser usado para o `PersistenceService` padrão, que é utilizado pelos handlers genéricos para as operações de CRUD.

## A Solução

Designar explicitamente um `DataSource` como o principal.



## Duas Estratégias para Definir o Padrão

1. **\*\*Via Service Binding\*\*:** Use a propriedade `cds.dataSource.binding` para especificar o nome do service binding.

```
cds.dataSource.binding: "my-primary-db-binding"
```

2. **\*\*Via Spring Bean\*\*:** Anote o `Bean` do `DataSource` com `@Primary`.

```
@Bean @Primary  
public DataSource myPrimaryDataSource() {  
    ...  
}
```

# Padrão Avançado: Multitenancy com Dados Compartilhados

Um cenário comum para uma aplicação multitenant precisa de:

- Um banco de dados por tenant (gerenciado via **Service Manager**) para dados isolados.
- Um banco de dados adicional e compartilhado entre todos os tenants para dados globais.

## Implementação em Produção

1. Faça o bind de ambos os serviços (Service Manager e o HDI container compartilhado) à aplicação.
2. Configure o service binding do Service Manager como o primário para que ele se torne o **PersistenceService** padrão (para dados dos tenants).
3. O CAP criará automaticamente um **PersistenceService** nomeado para o segundo binding, que pode ser usado para acessar os dados compartilhados.

```
1 spring:  
2   config.activate.on-profile: cloud  
3   cds:  
4     dataSource:  
5       binding: "my-service-manager-binding"
```

# Além do CQN: SQL Nativo e Modelos Estáticos

O CQN cobre a maioria dos casos de uso, mas o CAP Java oferece flexibilidade para cenários avançados.

## Opção 1: Descer para o SQL Nativo com `JdbcTemplate`

### Quando usar

Para executar stored procedures ou consultas SQL complexas que não podem ser expressas em CQN.

### Vantagem

Integra-se perfeitamente com o gerenciamento de transações e conexões do Spring.

```
@Autowired  
JdbcTemplate jdbcTemplate;  
  
public void callProcedure(int id, int stock) {  
    jdbcTemplate.update("call setStockForBook(?,?)", id, stock);  
}
```

## Opção 2: Subir para um Modelo Estático Tipado (Type-safe)

### Quando usar

Para construir consultas CQN de forma segura, com autocompletar e verificação em tempo de compilação, evitando erros com nomes de entidades/elementos em strings.

# O Poder da Tipagem: Construindo Queries com o Modelo Estático

O CDS Maven Plugin pode gerar um conjunto de interfaces Java que espelham seu modelo de dados, permitindo a construção de queries fluentes e type-safe.

## Query Dinâmica (Baseada em Strings)

- ✗ Sem autocompletar.
- ✗ Erros de digitação só são encontrados em tempo de execução.

```
1 // "Books", "author", "name" são apenas strings
2 Select.from("my.bookshop.Books")
3   .columns("title")
4   .where(book ->
book.to("author").get("name").eq("Edgar Allan
Poe"));
```

## Query Estática (Type-Safe)

- ✓ Autocompletar no seu IDE.
- ✓ Verificação de tipos e nomes em tempo de compilação.
- ✓ Código mais legível e seguro.

```
1 // Usa a interface Books_ gerada
2 Select<Books_> query = Select.from(Books_.class)
3   .columns(b -> b.title())
4   .where(b -> b.author().name().eq("Edgar Allan
Poe"));
```

# Da Fundação à Maestria: Sua Arquitetura de Persistência

Dominar os Persistence Services no CAP Java significa fazer escolhas de arquitetura deliberadas e utilizar a ferramenta certa para cada tarefa.

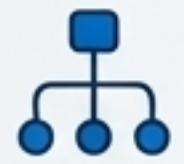
## Resumo dos Pontos-Chave



- **Estratégia Primeiro:** Escolha seu banco de dados (HANA, PostgreSQL, H2, SQLite) com base no seu caso de uso.



- **Configuração é Poder:** Domine a configuração de `DataSources` via service bindings e Spring para criar conexões robustas.



- **Escale com Complexidade:** Utilize múltiplos Persistence Services para cenários avançados como multitenancy.



- **Escreva Código Melhor:** Adote o modelo estático para construir queries seguras, legíveis e livres de erros em tempo de execução.

