



Dominando o Contexto no CAP Java

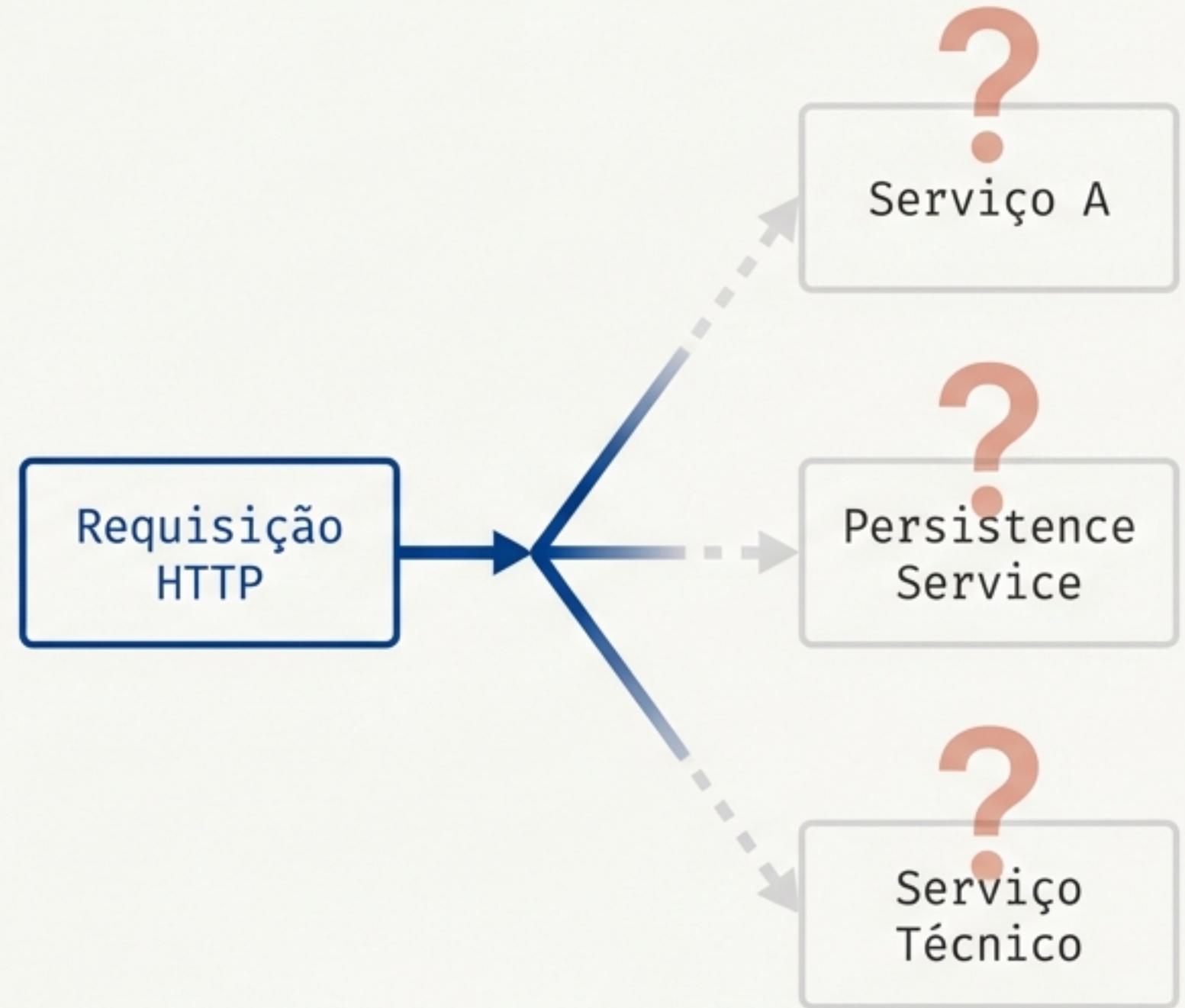
Um Guia Profundo sobre `RequestContext`

No coração de toda aplicação CAP robusta está o `RequestContext`. Ele é a maleta que carrega as informações essenciais—usuário, tenant, locale—através de cada etapa da sua lógica de negócios. Vamos aprender a não apenas usá-la, mas a dominá-la.

O Desafio: Mantendo o Estado em um Mundo de Microserviços

Uma única requisição HTTP ou um evento assíncrono aciona uma cascata de eventos em múltiplos serviços. Como garantimos que cada um deles saiba:

- ? Quem é o usuário atual?
- ? Em qual tenant estamos operando?
- ? Qual é o locale para a tradução?
- ? Como o Persistence Service sabe qual banco de dados específico do tenant consultar?



A Solução do CAP: `RequestContext` como um `ThreadLocal`

O CAP Java SDK gerencia e expõe essas informações através de instâncias de `RequestContext`. Pense nele como um escopo, geralmente atrelado a uma única requisição HTTP, que está sempre disponível.

RequestContext	EventContext
<p>Thread</p> 	 <p>Handler 1 → Handler 2</p> <p>Handler 1 → Handler 2</p> <pre>graph LR; H1[Handler 1] --> H2[Handler 2]; H1 -- gear --> H2;</pre>

- Mantido como `ThreadLocal`.
- Acessível em qualquer ponto da pilha de chamadas na mesma thread.
- Contém informações abrangentes (usuário, tenant, headers).
- Essencial para serviços técnicos como o Persistence Service.

- Passado explicitamente para os event handlers.
- Contém informações específicas do evento atual.
- Fornece acesso ao `RequestContext` ativo.

Nota de Destaque: Dentro de um event handler, é garantido que um `RequestContext` está disponível.

O Que Há Dentro da Maleta do Contexto?



`UserInfo`

Expõe a API para buscar dados do usuário autenticado: ID, nome, papéis (roles) CAP e o tenant.



`ParameterInfo`

Fornece acesso a dados adicionais da requisição: valores de header, parâmetros de query e o `locale`.



`AuthenticationInfo`

Contém as "claims" de autenticação.
Contém as 'claims' de autenticação.
Permite acesso ao token JWT bruto para propagação de usuário ou cenários de autenticação customizados.



`FeatureTogglesInfo`

Permite verificar quais feature toggles estão ativados para a requisição atual, possibilitando lógicas de negócios dinâmicas.

Acessando o Contexto na Prática

Opção 1: Via Injeção de Dependência (Spring)

A forma mais limpa quando se usa Spring.

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

Opção 2: A Partir do `EventContext`

Universal e sempre disponível em um handler.

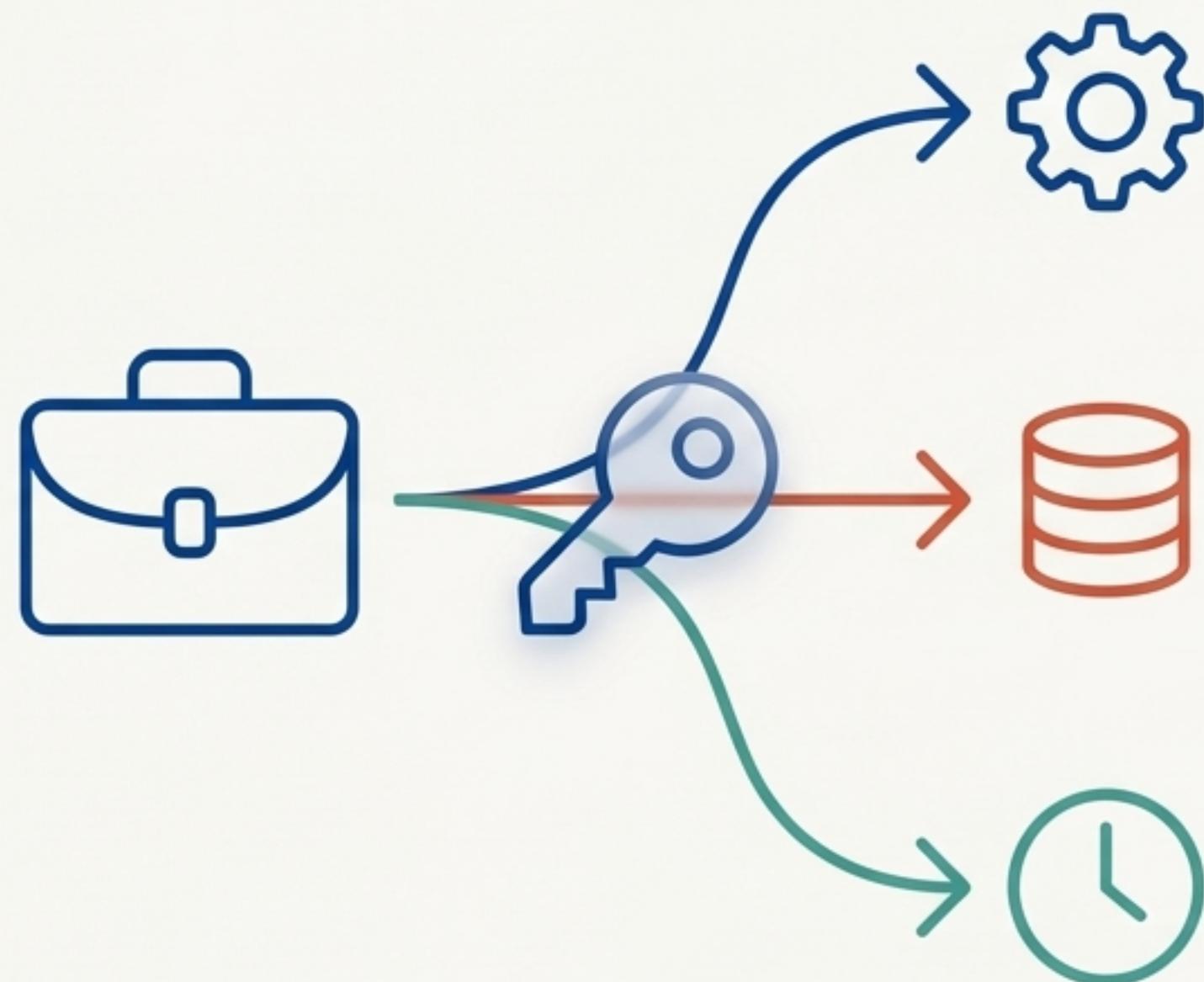
```
@Before(event = CqnService.EVENT_READ)  
public void beforeRead(CdsReadEventContext context) {  
    UserInfo userInfo = context.getUserInfo();  
    ParameterInfo parameterInfo =  
        context.getParameterInfo();  
    AuthenticationInfo authInfo =  
        context.getAuthenticationInfo();  
    // ...  
}
```

A Necessidade de Mudar de Perspectiva

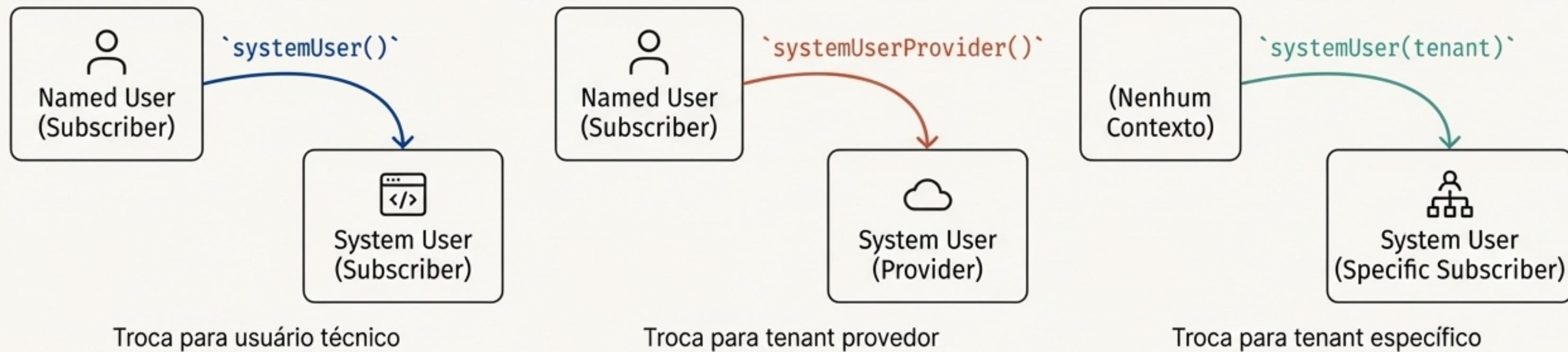
Às vezes, o contexto inicial de um usuário nomeado não é suficiente. Precisamos realizar ações que exigem uma identidade ou escopo diferente, como:

- Executar uma tarefa de fundo (background job) para um tenant específico.
- Chamar um serviço técnico interno com privilégios de sistema.
- Acessar dados compartilhados que residem no tenant provedor.

Para esses cenários, o CAP fornece a API ``RequestContextRunner``, uma forma fluente e segura de criar e executar código dentro de um novo contexto aninhado.



Os Três Cenários Essenciais de Troca de Contexto

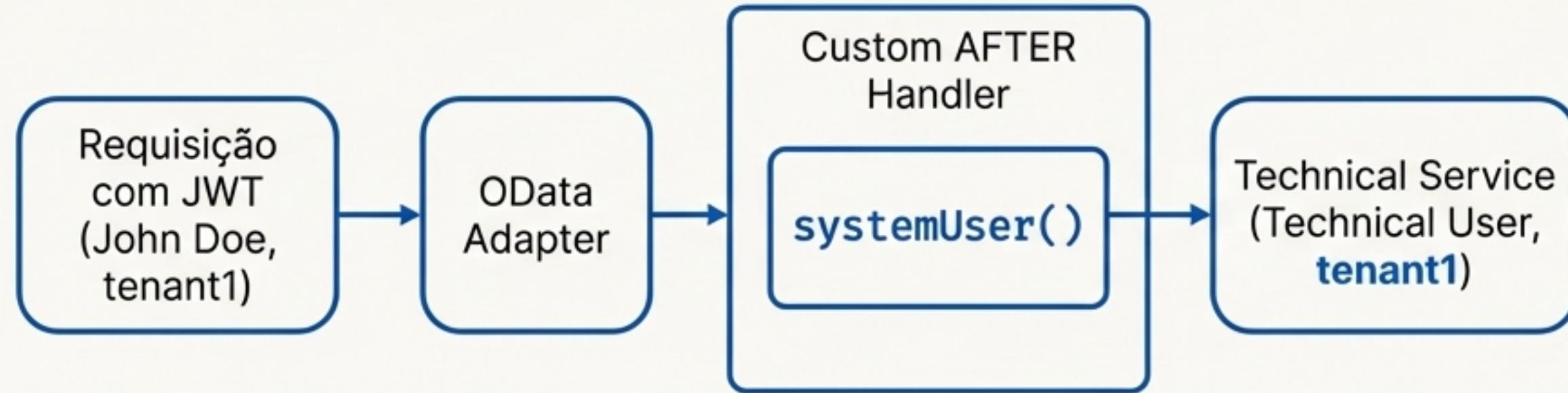


O `RequestContextRunner` oferece métodos de conveniência para as transições mais comuns em aplicações multitenant.

Método	Descrição
<code>'systemUser()'</code>	Troca para um usuário técnico, mas preserva o tenant atual. Ideal para rebaixar privilégios.
<code>'systemUserProvider()'</code>	Troca para um usuário técnico no escopo do tenant provedor. Usado para acessar recursos compartilhados.
<code>'systemUser(tenant)'</code>	Troca para um usuário técnico em um tenant específico. Essencial para jobs assíncronos e tarefas de sistema.

Cenário 1: Agindo como Usuário de Sistema no Mesmo Tenant

O caso de ‘rebaixamento de privilégios’ com ‘systemUser()’

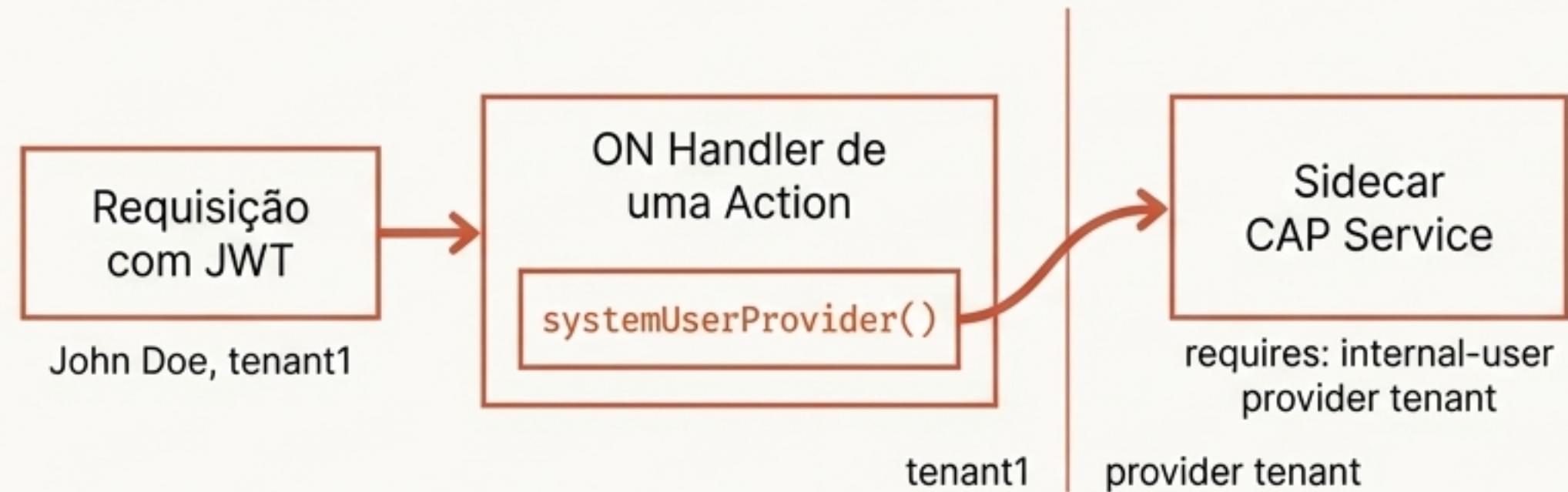


Um handler ‘AFTER’ precisa chamar um serviço técnico interno sem propagar as permissões do usuário original. O `systemUser()` remove o usuário nomeado, mantendo o contexto do tenant.

```
// No After handler
runtime.requestContext().systemUser().run(reqContext -> {
    // Chamar serviço técnico...
    // Aqui, userInfo.isAuthenticated() seria false.
    // O tenant, no entanto, é preservado do contexto original.
});
```

Cenário 2: Acessando Recursos do Tenant Provedor

Acessando o 'cofre' compartilhado com `systemUserProvider()`

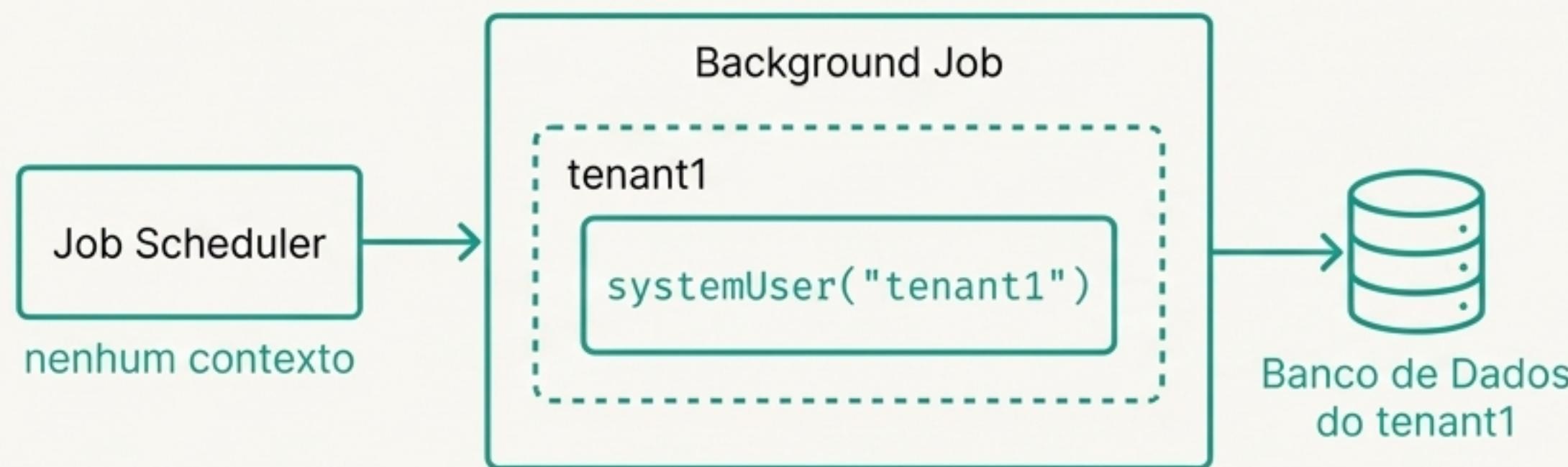


Sua aplicação precisa chamar um serviço auxiliar (sidecar) ou ler dados de configuração que são compartilhados entre todos os tenants.
`systemUserProvider()` muda o contexto para o tenant provedor, permitindo esse acesso.

```
// No ON handler de uma Action
runtime.requestContext().systemUserProvider().run(reqContext → {
    // Chamar serviço remoto no tenant provedor...
    // O acesso à persistência aqui usaria a conexão do provedor.
});
```

Cenário 3: Executando Tarefas em Nome de um Inquilino

O poder do `systemUser(tenant)` para Background Jobs



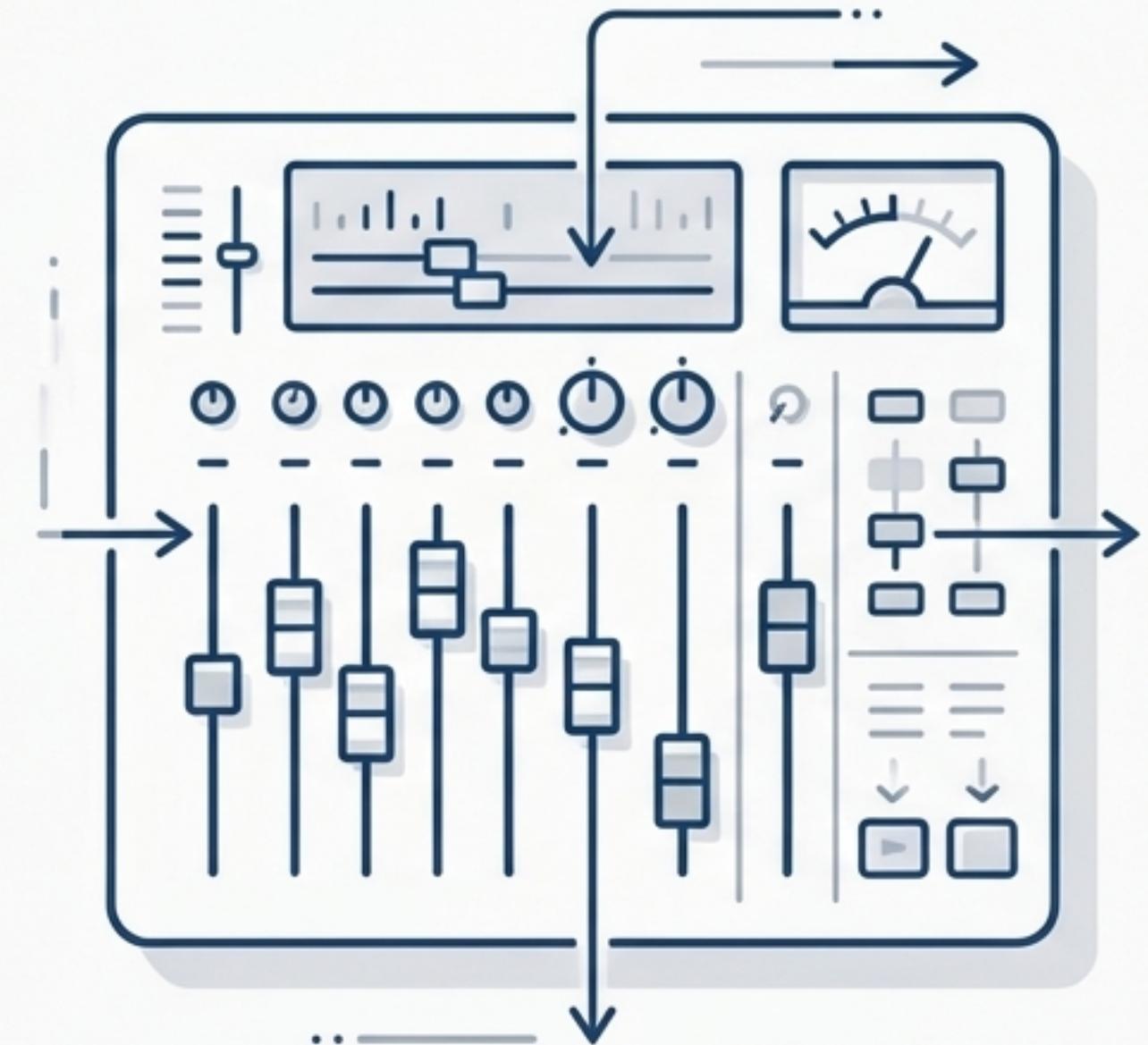
Um job agendado precisa processar dados para um tenant específico. Como o job roda fora de um contexto de requisição, você deve criar explicitamente um contexto para garantir que todas as operações, especialmente as de persistência, ocorram no tenant correto.

```
// Dentro de um background job
runtime.requestContext().systemUser(tenantId).run(reqContext -> {
    return persistenceService.run(Select.from(Books_.class))
        .listOf(Books.class); // Esta query será executada no DB do tenantId
});
```

Além do Padrão: Modificando Contextos com Precisão Cirúrgica

O `RequestContextRunner` não serve apenas para trocar de usuário ou tenant. Ele oferece um conjunto de APIs fluentes para modificar, adicionar ou remover atributos específicos de um contexto aninhado, sem afetar o original.

- `modifyParameters(param -> ...)`: Adiciona, modifica ou remove parâmetros como headers ou o locale.
- `modifyUser(user -> ...)`: Modifica atributos do `UserInfo`, como remover um role ou alterar o tenant.
- `clearParameters()`: Limpa todos os parâmetros.



Isso permite criar um “sanduíche” de contexto, onde você adiciona uma camada de modificação temporária para uma operação específica.

A Regra da Herança e Suas Exceções Críticas

Um novo contexto aninhado **herda** cópias de todos os valores do seu contexto pai. As modificações que você aplica são feitas sobre essa cópia herdada.



O Que NÃO Pode Ser Modificado

O **Modelo CDS** e os **Feature Toggles** são determinados apenas no contexto inicial da requisição e não podem ser alterados em um contexto aninhado.

A Grande Exceção

...a menos que você modifique o tenant do usuário! Alterar o tenant **redetermina** o modelo CDS para aquele tenant.



Dica Essencial: Transações e Tenants

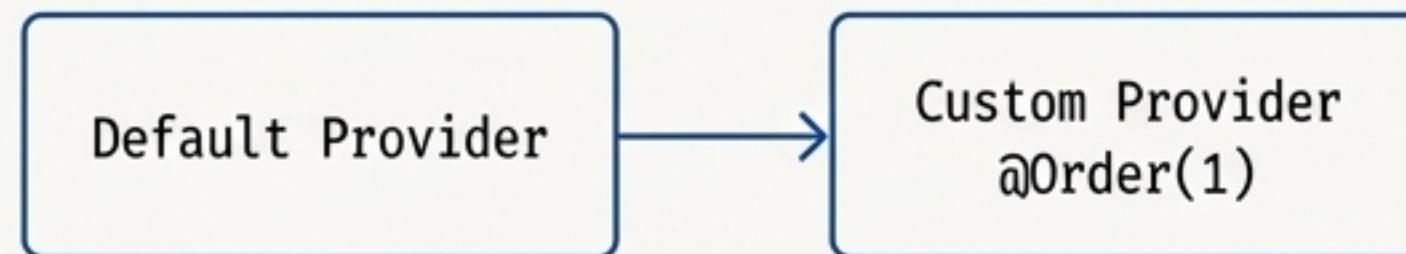
Ao mudar o tenant do usuário, é **obrigatório** abrir um novo `ChangeSet`. O CAP Java SDK detecta essa condição e previne o acesso ao banco de dados para evitar vazamento de dados entre tenants se você esquecer.

Customizando a Origem: Provedores Globais de Contexto

Como o CAP determina as informações iniciais de usuário e parâmetros? Através de provedores (`UserInfoProvider`, `ParameterInfoProvider`). Você pode registrar suas próprias implementações para substituir ou estender o comportamento padrão.

Caso de Uso

Imagine um cenário onde as informações do usuário não vêm de um JWT, mas de headers HTTP customizados. Você pode criar um `UserInfoProvider` para ler esses headers.



```
@Component  
@Order(1) // A ordem é importante!  
public class HeaderBasedUserInfoProvider implements  
UserInfoProvider {  
    @Autowired  
    HttpServletRequest req;  
  
    @Override  
    public UserInfo get() {  
        // Lógica para ler 'custom-tenant-header', etc.  
        return UserInfo.create()  
            .setTenant(req.getHeader("custom-tenant-header"))  
            .setName(req.getHeader("custom-username-header"))  
    }  
}
```

Lê header customizado

Conceito Chave

Encadeie provedores usando `@Order` e `setPrevious()` para criar pipelines de modificação (ex: normalizar nomes de usuário).

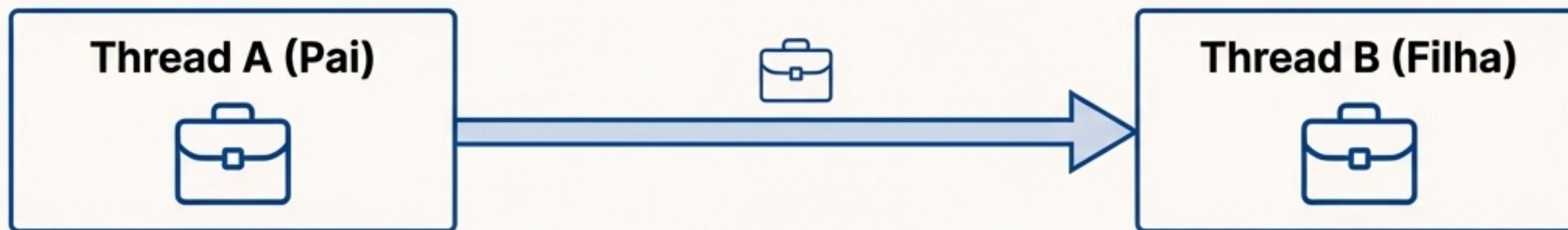
Concorrência Segura: Propagando o Contexto para Novas Threads

O Problema

Ao iniciar uma nova thread (ex: com `CompletableFuture` ou `ExecutorService`), o `RequestContext` **não** é propagado automaticamente, pois ele é um `ThreadLocal`. Chamadas a serviços na nova thread falharão ou usarão o contexto errado.

A Solução

Capture o `RequestContextRunner` na thread pai e use seu método `run()` dentro da thread filha para estabelecer o contexto corretamente.



```
// Na thread principal (pai)
RequestContextRunner runner = runtime.requestContext();

// Na thread filha
Future<CdsResult> result = Executors.newSingleThreadExecutor().submit(() -> {
    // Envolve a lógica da thread filha com runner.run()
    return runner.run(threadContext -> {
        return persistenceService.run(Select.from(Books_.class));
    });
});
```

De Conceito à Maestria: Seu Domínio sobre o `RequestContext`



Entender



Aplicar



Customizar



O `RequestContext` é seu `ThreadLocal` de confiança

Ele carrega o estado da requisição (usuário, tenant, etc.) de forma implícita e segura.



Use o `RequestContextRunner` para mudar de perspectiva

Domine os três cenários (`systemUser`, `systemUserProvider`, `systemUser(tenant)`) para cobrir 90% das suas necessidades.



Controle os detalhes quando necessário

Modifique parâmetros, entenda a herança e crie provedores personalizados para cenários avançados.



Propague o contexto em threads

Nunca se esqueça de usar o `runner.run()` ao trabalhar com concorrência.

Compreender o `RequestContext` é a chave para construir aplicações CAP Java seguras, escaláveis e prontas para multitenancy.