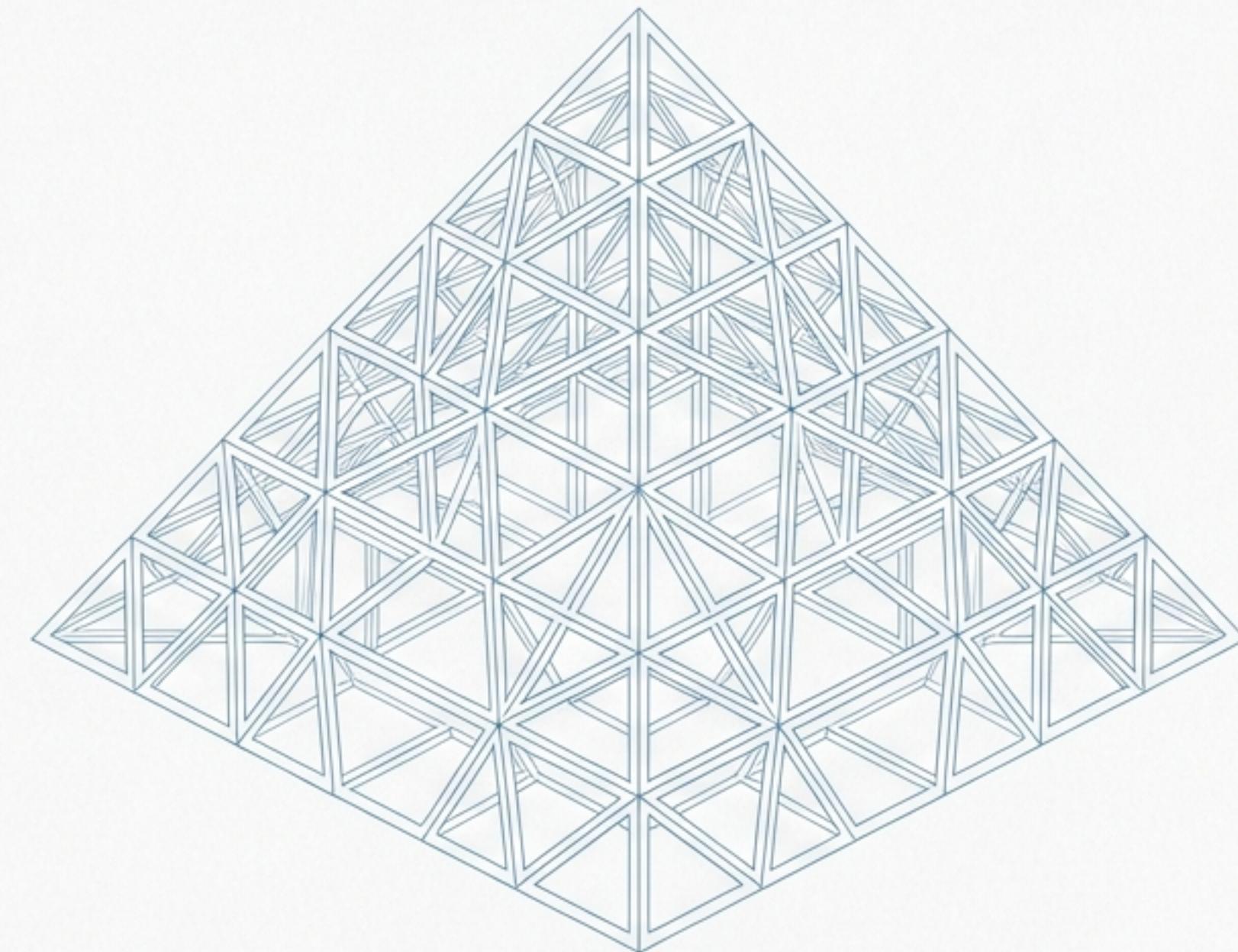


Construindo Aplicações CAP Java Robustas

Uma Estratégia de Testes em Camadas Guiada pela Pirâmide

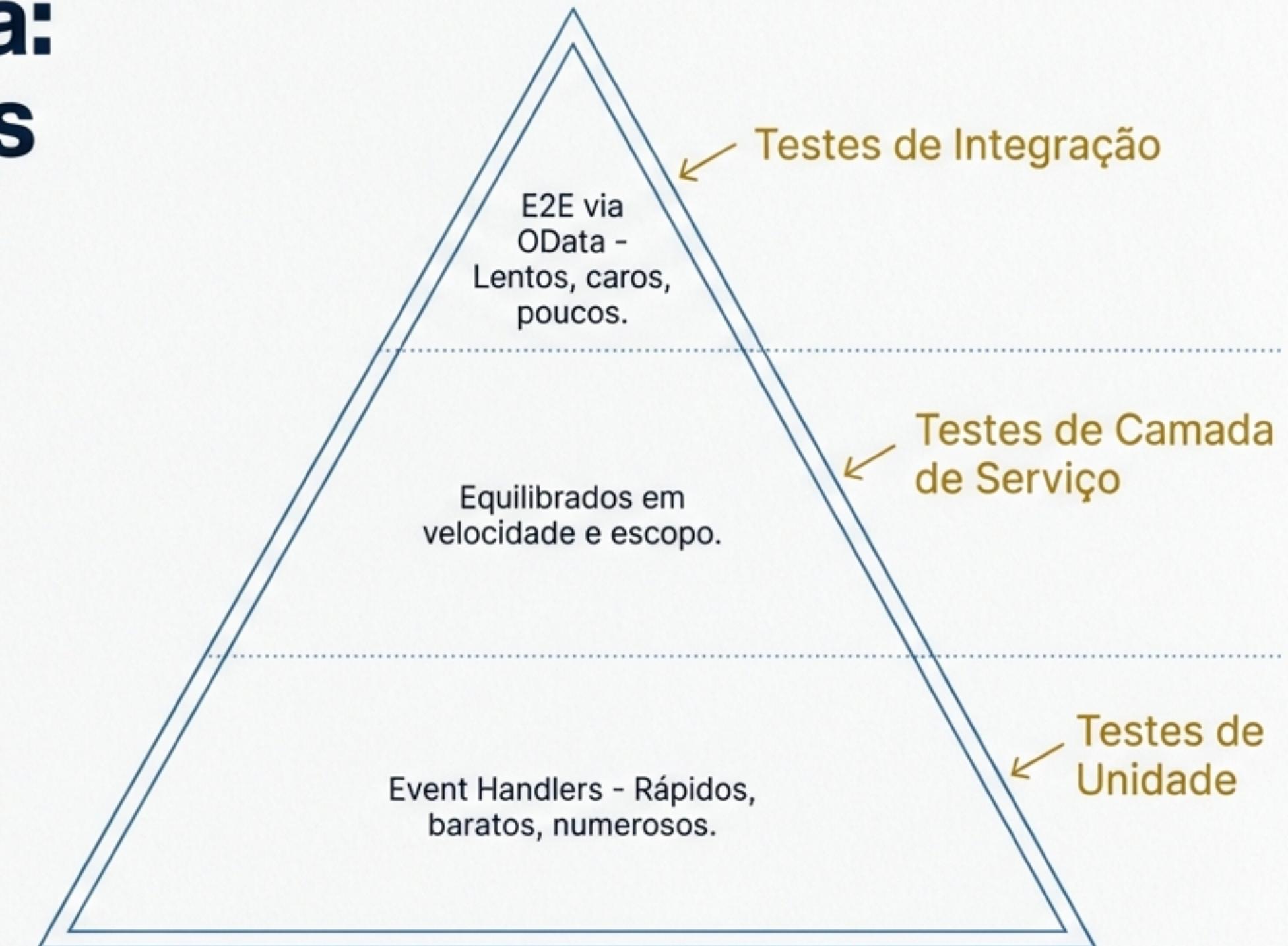


Este guia apresenta as melhores práticas para testar aplicações CAP Java, organizando os testes em uma estrutura lógica que garante cobertura, manutenibilidade e confiança no código.

Nossa Metáfora Guia: A Pirâmide de Testes

A arquitetura de componentes fracamente acoplados do CAP Java permite um escopo de teste preciso.

Para estruturar nossa abordagem, adotamos a Pirâmide de Testes, um modelo que prioriza testes rápidos e isolados na base e testes mais lentos e integrados no topo.



Recomendamos o uso do framework Spring e JUnit para conveniência em testes de unidade e integração, aproveitando recursos como injeção de dependência e MockMvc.

A Anatomia do Nosso Serviço de Exemplo: `CatalogServiceHandler`

Para ilustrar cada camada da pirâmide, analisaremos dois métodos de um `CatalogServiceHandler`: um ideal para testes de unidade e outro que exige uma abordagem mais integrada.

Lógica de Negócio Pura: `discountBooks`

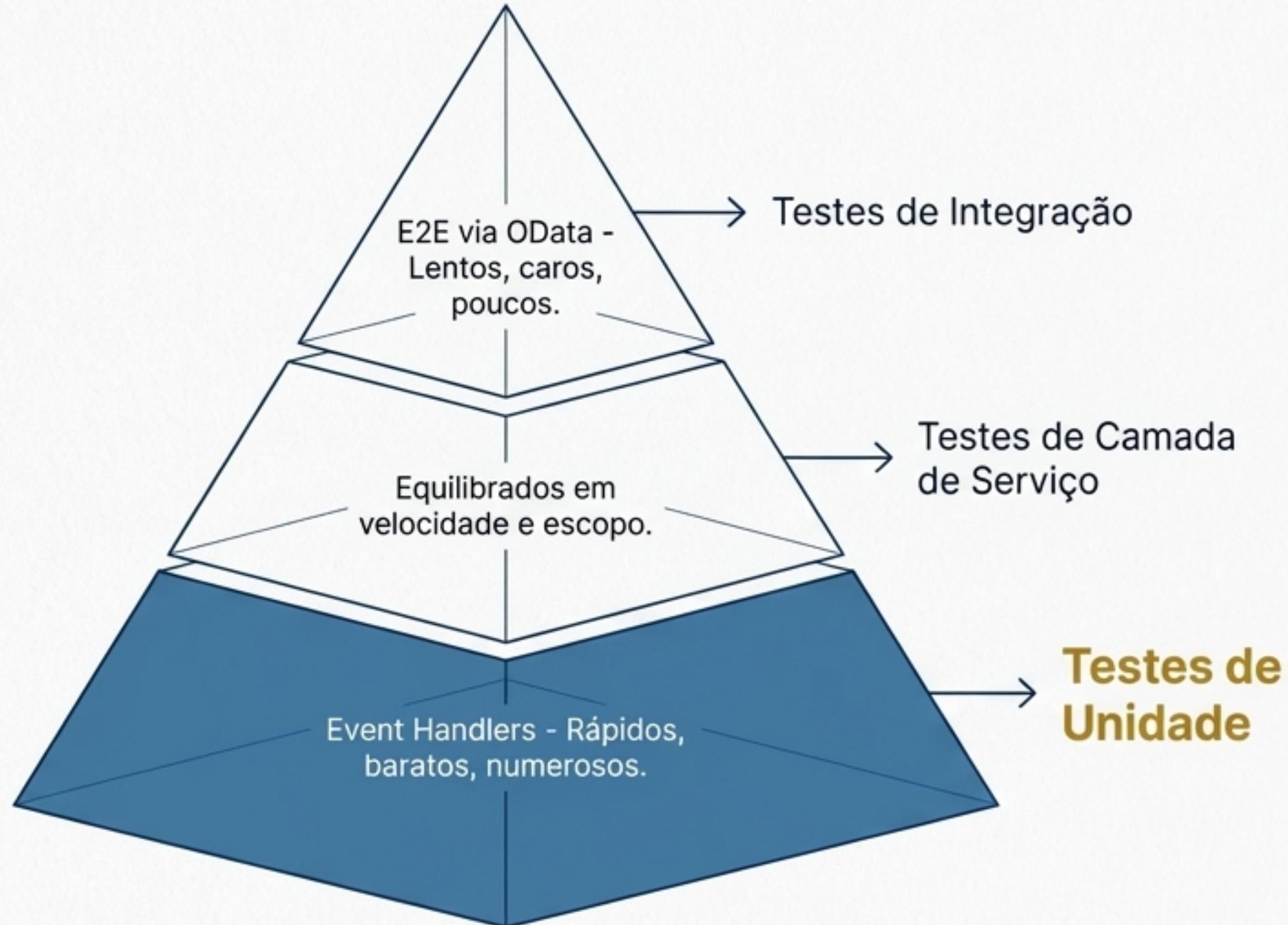
```
@After(event = CqnService.EVENT_READ)
public void discountBooks(Stream<Books> books) {
    books.filter(b -> b.getTitle() != null).forEach(b -> {
        loadStockIfNotSet(b);
        discountBooksWithMoreThan111Stock(b);
    });
}
```

→ Aplica um desconto no título do livro se o estoque for maior que 111. Não depende diretamente do `PersistenceService` para sua lógica principal.

Lógica com Dependência: `onSubmitOrder`

```
@On
public void onSubmitOrder(SubmitOrderContext context) {
    Integer quantity = context.getQuantity();
    String bookId = context.getBook();
    Optional<Books> book = db.run(Select.from(BOOKS)
        .columns(Books_::stock).byId(bookId));
    // ... (error handling and stock update logic)
}
```

← Reduz o estoque de um livro após um pedido. Interage diretamente com o `PersistenceService` para ler e atualizar o banco de dados.



A Base da Pirâmide: Testes de Unidade Rápidos e Isolados

A primeira camada de defesa é testar a lógica de processamento pura de um `Event Handler`. Ao "mockar" (simular) dependências como o `PersistenceService`, podemos verificar o comportamento de um método em total isolamento. Isso resulta em testes extremamente rápidos e focados.

Ferramenta em Destaque

O Mockito, já incluído no `spring-boot-starter-test`, é a ferramenta ideal para criar mocks de dependências.

Quando usar

Ideal para handlers ou métodos que não dependem de interações complexas com banco de dados ou outros serviços para executar sua lógica principal.

Testando a Lógica de Desconto em Isolamento

O método `discountBooks` é um candidato perfeito para um teste de unidade. Sua lógica de aplicar desconto depende apenas dos dados de entrada. Mockamos o `PersistenceService` para satisfazer a construção do handler, mas a lógica em si não o utiliza.

```
@ExtendWith(MockitoExtension.class)
public class CatalogServiceHandlerTest {
    @Mock
    private PersistenceService db; ← A dependência é simulada com Mockito. Nenhuma conexão real com o banco de dados é necessária.

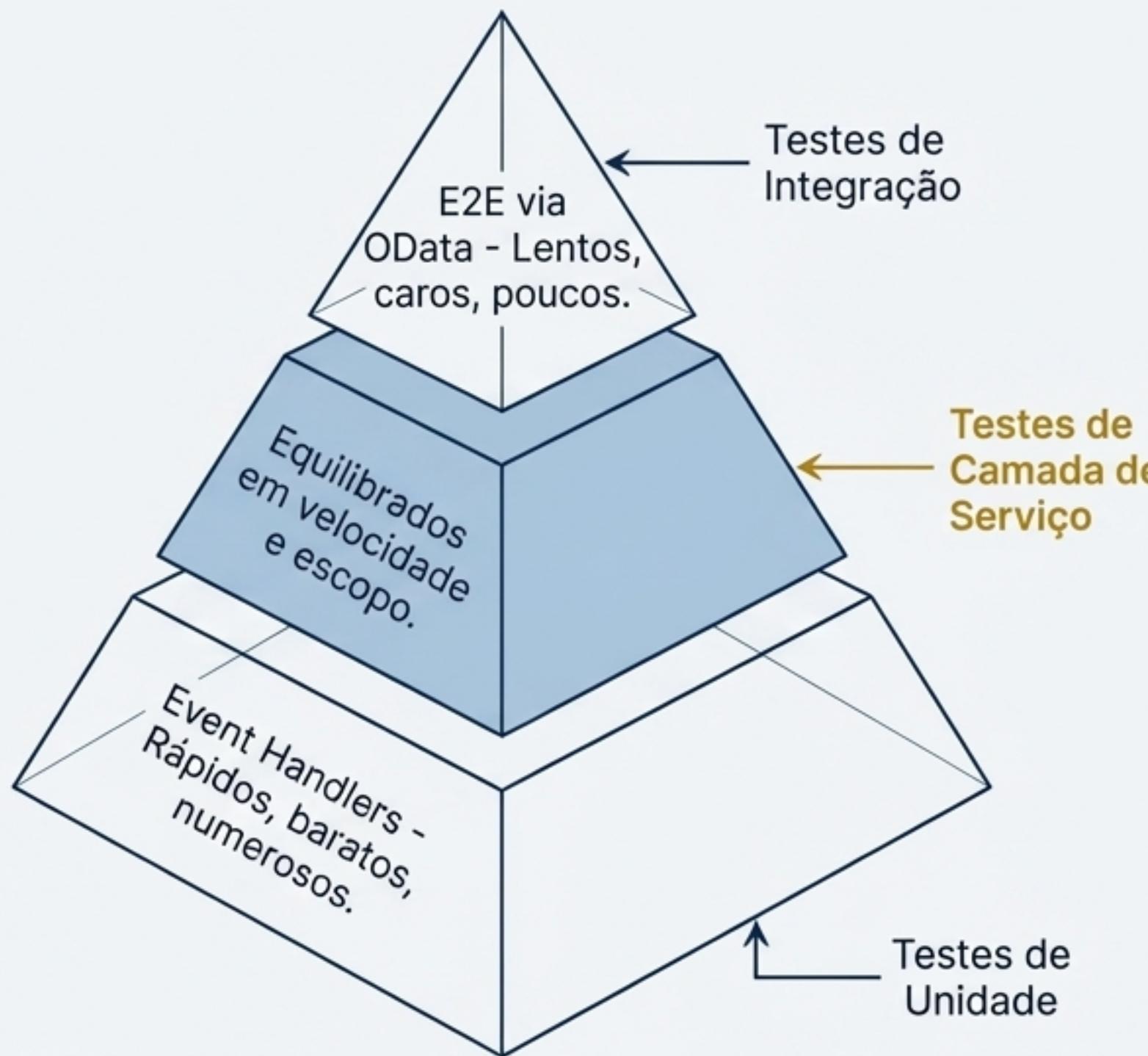
    @Test
    public void discountBooks() {
        Books book1 = Books.create();
        book1.setTitle("Book 1");
        book1.setStock(10);

        Books book2 = Books.create();
        book2.setTitle("Book 2");
        book2.setStock(200);

        CatalogServiceHandler handler = new CatalogServiceHandler(db);
        handler.discountBooks(Stream.of(book1, book2)); ← Invocamos o método diretamente com dados controlados.

        assertEquals("Book 1", book1.getTitle(), "Book 1 was not discounted");
        assertEquals("Book 2 -- 11% discount", book2.getTitle(), "Book 2 was discounted"); ← Verificamos se o título foi alterado apenas para o livro que atende à condição de estoque.
    }
}
```

O Meio da Pirâmide: Testando a Camada de Serviço



Para testar handlers que interagem com outras partes do framework, como o 'PersistenceService', subimos um nível. **Em vez de invocar o método diretamente, interagimos com a camada de serviço. Isso nos permite verificar o comportamento do handler em um ambiente mais realista, mas ainda sem o overhead de requisições HTTP.**

- **Execução de CQN:** Executar statements CQN (como Select) diretamente contra a interface do serviço para verificar os resultados de handlers que modificam dados.
- **Emissão de Eventos:** Disparar um EventContext (como SubmitOrderContext) através do método emit() para acionar a execução de um handler específico.

Verificando a Redução de Estoque via Emissão de Evento

O método `onSubmitOrder` precisa do `PersistenceService` para funcionar. Para testá-lo, disparamos um evento `SubmitOrderContext` através da camada de serviço, que acionará a execução do nosso handler.

```
@SpringBootTest
public class CatalogServiceTest {
    @Autowired
    @Qualifier(CatalogService_.CDS_NAME)
    private CqnService catalogService;

    @Test
    public void submitOrder() {
        SubmitOrderContext context = SubmitOrderContext.create();
        // ID de um livro conhecido por ter estoque de 22
        context.setBook("4a519e61-3c3a-4bd9-ab12-d7e0c5329933");
        context.setQuantity(2);

        catalogService.emit(context);
    }
}
```

Preparamos o `EventContext` com os dados necessários para o teste.

O método `emit` aciona o handler `onSubmitOrder` correspondente.

A assertão verifica se o estoque retornado no contexto do resultado foi corretamente atualizado.

Garantindo o Tratamento de Erros: Estoque Insuficiente

Uma estratégia de testes completa também deve validar o comportamento em caso de falha. Verificamos se uma `ServiceException` é lançada quando a quantidade do pedido excede o estoque disponível.

```
@SpringBootTest
public class CatalogServiceTest {
    // ... (Autowired catalogService)
    @Test
    public void submitOrderExceedingStock() {
        SubmitOrderContext context = SubmitOrderContext.create();
        // ID do mesmo livro com estoque de 22
        context.setBook("4a519e61-3c3a-4bd9-ab12-d7e0c5329933");
        context.setQuantity(30); ←

        assertThrows(ServiceException.class, () -> {
            catalogService.emit(context);
        });
    }
}
```

Configuramos uma condição de falha: pedido maior que o estoque.

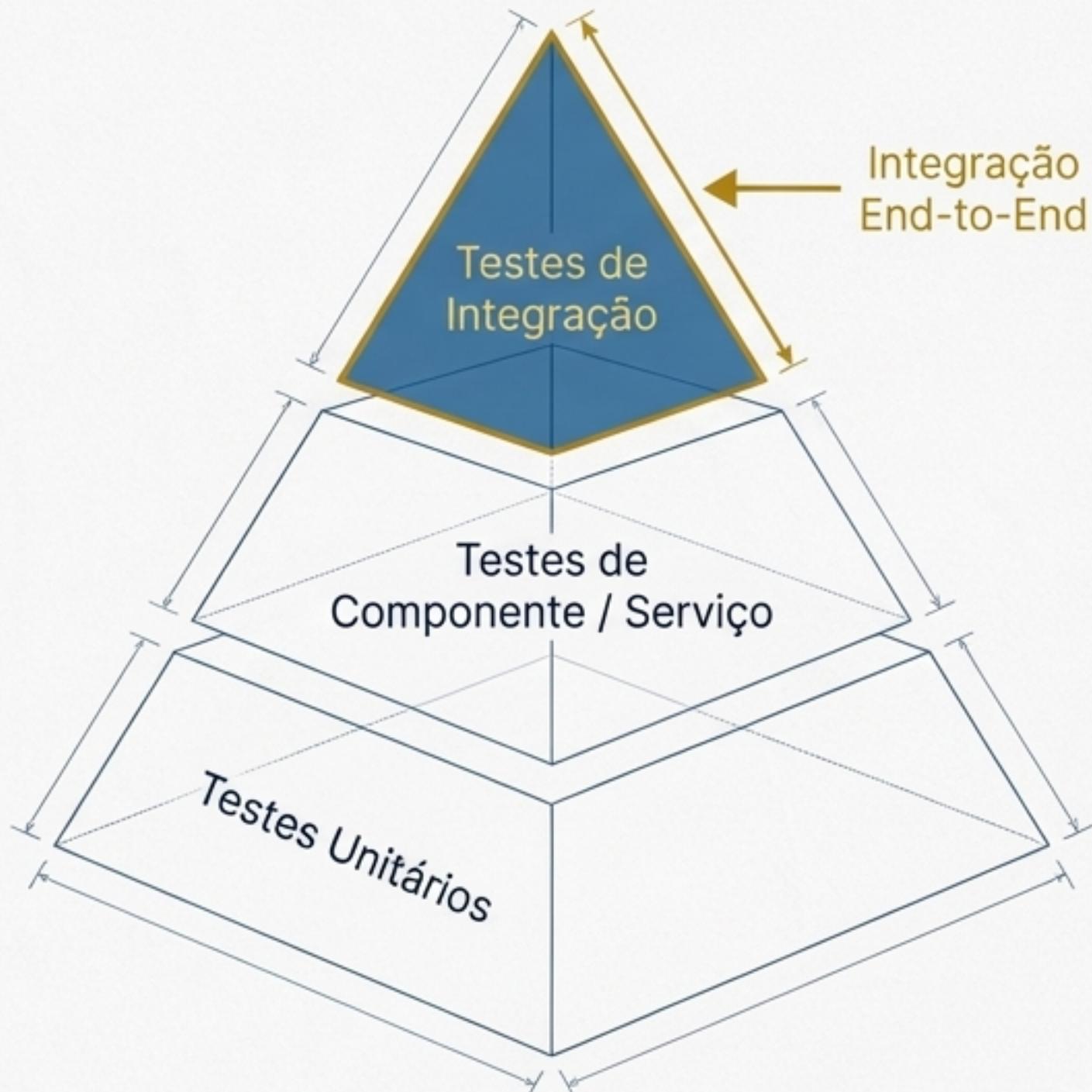
Usamos `assertThrows` do JUnit para confirmar que a execução do `emit` resulta em uma `ServiceException`, como esperado.

O Topo da Pirâmide: Testes de Integração End-to-End

No topo da pirâmide, validamos o comportamento da aplicação em um ‘roundtrip’ completo. Simulamos requisições HTTP para os endpoints OData expostos e verificamos a resposta. Isso garante que todas as camadas — do protocolo ao handler e ao banco de dados — estão funcionando corretamente em conjunto.

Ferramenta em Destaque

O `MockMvc` do Spring é a ferramenta chave aqui. Ele permite invocar requisições a endpoints OData sem a necessidade de um servidor web real, tornando os testes mais rápidos e confiáveis.



Validando a Lógica de Desconto Através do Endpoint OData

Vamos testar o handler `discountBooks` novamente, mas desta vez através de uma requisição `GET` ao endpoint `/api/browse/Books`, simulada com `MockMvc`.

```
@SpringBootTest  
@AutoConfigureMockMvc  
public class CatalogServiceITest {  
    private static final String booksURI = "/api/browse/Books";  
    @Autowired private MockMvc mockMvc;  
  
    @Test  
    public void discountApplied() throws Exception {  
        mockMvc.perform(get(booksURI + "?$filter=stock gt 200&top=1"))  
            .andExpect(status().isOk())  
            .andExpect(jsonPath("$.value[0].title", containsString("-- 11% discount")));  
    }  
  
    @Test  
    public void discountNotApplied() throws Exception {  
        mockMvc.perform(get(booksURI + "?$filter=stock lt 100&top=1"))  
            .andExpect(status().isOk())  
            .andExpect(jsonPath("$.value[0].title", not(containsString("-- 11% discount"))));  
    }  
}
```



Simula uma requisição GET para buscar um livro com estoque alto.

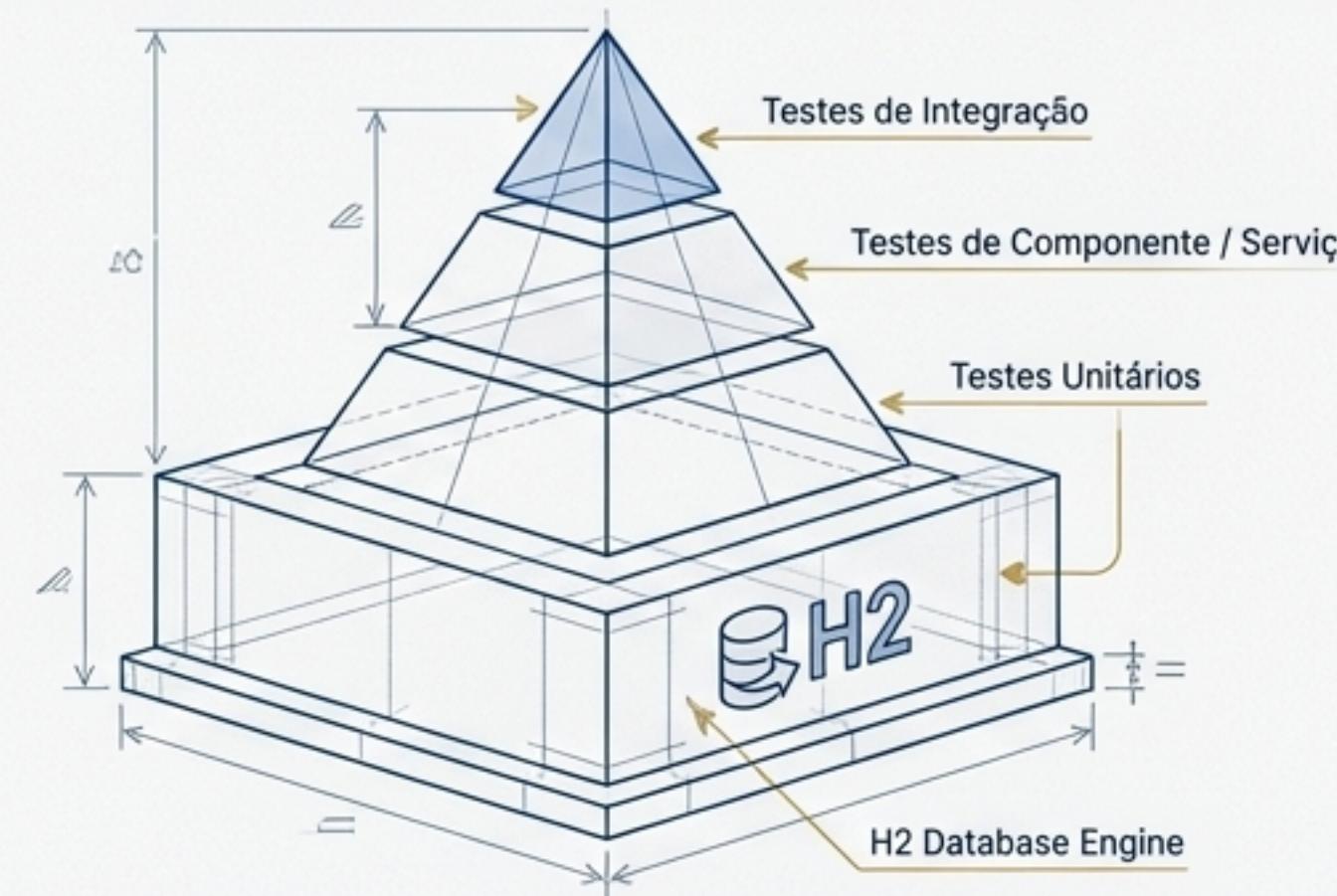
Verifica se a resposta JSON contém o texto do desconto no título.

Simula uma requisição para um livro com estoque baixo.

Verifica se o texto do desconto NÃO está presente no título.

A Fundação da Pirâmide: H2 como Banco de Dados para Testes Locais

Uma estratégia de testes eficaz precisa de uma base sólida e rápida. Para o CAP Java, essa base é o H2, o banco de dados preferido para desenvolvimento e testes locais.



Acesso Concorrente

Suporta múltiplas conexões com travamento em nível de linha, permitindo testes paralelos seguros.



Nativo Java e Open Source

Integração ótima com aplicações Java e mantido por uma comunidade ativa.



Ferramentas de Administração

Inclui um console web para fácil administração e inspeção de dados.

Configurando o H2 em seu Projeto

A configuração inicial do H2 é simples, especialmente ao usar o Maven Archetype do CAP. Para configuração manual, siga estes dois passos:

Passo 1: Adicionar a Dependência Maven

Inclua o driver JDBC do H2 no seu `pom.xml` com escopo de `runtime`.

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Passo 2: Configurar o Perfil `default` no Spring

No `application.yaml`, ative o H2 para o perfil `default`, usado em desenvolvimento e testes. O Spring o inicializará automaticamente como um banco de dados em memória.

```
spring:
  config.activate.on-profile: default
  sql.init.platform: h2
cds:
  data-source:
    auto-config.enabled: false
```

Conhecendo as Limitações e a Solução Híbrida

Limitações do H2

É crucial entender as restrições do seu banco de dados de teste. O H2 possui características próprias de performance, tipos de dados e transações.

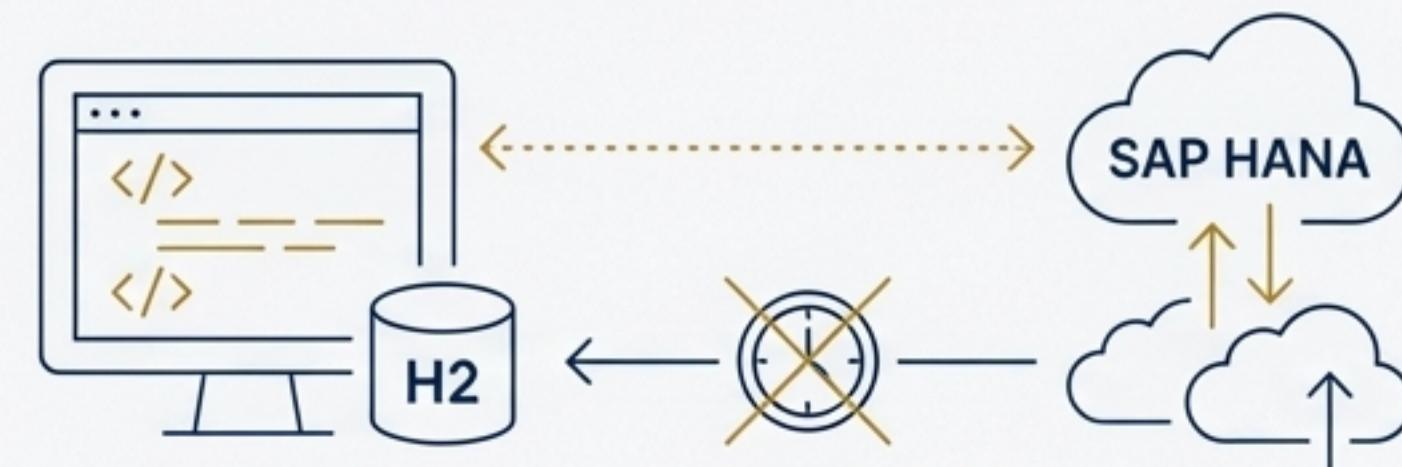
Ponto Crítico: Não é possível usar o H2 para testar cenários de **multitenancy e extensibilidade (MTXS)** em um ambiente local.



A Solução: Hybrid Testing

Quando as limitações do H2 se tornam um bloqueio, o **Hybrid Testing** é a resposta.

Permite que você continue desenvolvendo e testando localmente, mas conectando-se seletivamente a serviços na nuvem (como um SAP HANA) quando necessário. Isso evita os longos ciclos de deploy na nuvem para cada verificação.



Otimizando o Fluxo de Trabalho de Desenvolvimento Local

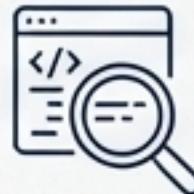


Integrando H2 com Spring DevTools

Problema: O reinício automático do Spring DevTools apaga os dados do H2 quando configurado em memória.

Solução: Use o modo de H2 baseado em arquivo para persistir os dados no disco entre reinitializações.

```
spring:  
  config.activate.on-profile: default  
  datasource:  
    url: "jdbc:h2:file:/data/testdb" # Caminho para o arquivo do DB  
    # ... (resto da configuração)
```



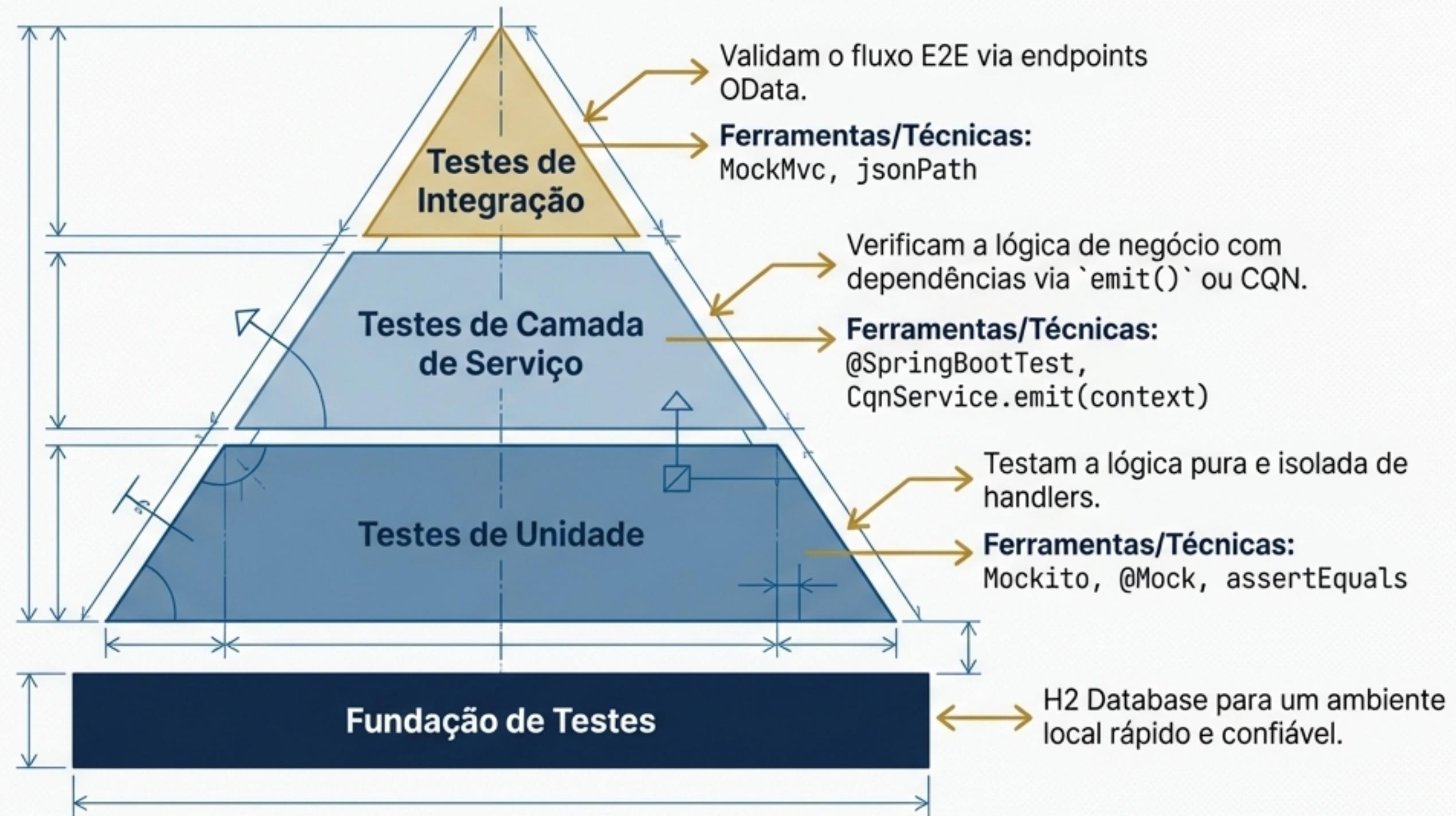
Visualizando Statements SQL

Necessidade: Para depurar ou entender o comportamento em tempo de execução, é útil ver os comandos SQL gerados.

Solução: Ative o log `DEBUG` para o pacote de persistência do CDS.

```
logging:  
  level:  
    com.sap.cds.persistence.sql: DEBUG
```

O Blueprint da Estratégia de Testes para Aplicações CAP Java



Adotar essa abordagem em camadas não apenas aumenta a cobertura, mas cria uma suíte de testes robusta, rápida e de fácil manutenção, resultando em aplicações CAP Java mais confiáveis.