📘 Apostila Técnica: Spring Store API (Expert Edition)

Versão do Projeto: 2.0 (Enterprise/Híbrido) **Stack:** Java 21 LTS, Spring Boot 3.4, PostgreSQL, Docker, JUnit 5.

Esta apostila consolida os padrões arquiteturais, decisões de design e soluções para problemas complexos (Concorrência, Segurança Híbrida, Infraestrutura) desenvolvidos durante a mentoria.

1. 🏗️ Arquitetura e Padrões de Projeto

Adotamos a **Layered Architecture** (Arquitetura em Camadas) com foco em desacoplamento e segurança.

1.1. O Fluxo de Dados

A regra de ouro é: **Nunca expor a Entidade (`@Entity`) diretamente no Controller.**

`Request (JSON)` ➡️ `Controller` ➡️ `DTO` ➡️ `Mapper` ➡️ `Service` ➡️ `Repository` ➡️ `Database`

1.2. Padrões Aplicados

- **Dependency Injection:** Via construtor (`@RequiredArgsConstructor`). Garante imutabilidade e facilita testes (evita `@Autowired` em campos).
- **DTO Pattern:** Blindagem da API contra *Mass Assignment* e vazamento de dados sensíveis (ex: senha do usuário).
- **Mapper Pattern:** Centralização da lógica de conversão.
- **Global Exception Handling:** Uso de `@RestControllerAdvice` para capturar erros e retornar JSON padronizado (RFC 7807), evitando "Whitelabel Error Pages".

2. 🛡️ Segurança Avançada (Hybrid Auth)

Implementamos um sistema de segurança **Stateless** (sem sessão no servidor) que aceita múltiplos provedores.

2.1. Fluxo Híbrido (JWT + OAuth2)

O sistema aceita login via:

1. **Email/Senha:** Validação via `AuthenticationManager` e `BCrypt` .
2. **Google (OAuth2):** Delegação de autenticação e geração automática de conta.

Ambos os fluxos resultam na emissão de um **Token JWT** assinado pela nossa API.

2.2. Configuração do Porteiro (`SecurityFilterChain`)

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http, SecurityFilter securi
    return http
        .csrf(csrf -> csrf.disable()) // Stateless não precisa de CSRF
        .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPoli
```

```
            .authorizeHttpRequests(authorize -> authorize
                // Rotas Públicas
                .requestMatchers(HttpMethod.POST, "/api/auth/**").permitAll()
                .requestMatchers("/v3/api-docs/**", "/swagger-ui/**").permitAll()
                .requestMatchers("/error").permitAll() // Importante para não mascarar erro
                // Rotas Privadas
                .anyRequest().authenticated()
            )
            // Habilita Login Social
            .oauth2Login(oauth2 -> oauth2.successHandler(successHandler))
            // Instala nosso filtro de JWT antes do filtro padrão
            .addFilterBefore(securityFilter, UsernamePasswordAuthenticationFilter.class)
            .build();
    }
```

2.3. O Mito do CORS

Aprendemos que erros de CORS no Swagger muitas vezes são **Erros 403 (Forbidden)** mascarados pelo navegador. A solução real geralmente é corrigir a autenticação, não apenas os headers CORS.

3. 📦 Modelagem de Domínio e Concorrência

3.1. Relacionamentos JPA

Modelagem de um sistema de Pedidos (`Order`) complexo.

- **User 1:N Order:** Um usuário tem muitos pedidos.
- **Order 1:N OrderItem:** Um pedido tem itens exclusivos (Cascade ALL, Orphan Removal).
- **OrderItem N:1 Product:** Muitos itens referenciam um produto.

3.2. Controle de Concorrência (O Problema do Último Estoque)

Para evitar que dois usuários comprem o mesmo item simultaneamente, usamos **Optimistic Locking**.

Na Entidade:

```
@Entity
public class Product {
    // ...
    @Version // O Hibernate checa isso antes de qualquer update
    private Long version;
}
```

No Service (Lógica de Baixa):

```
// Se a versão no banco mudou desde a leitura, lança ObjectOptimisticLockingFailureExce
productRepository.save(product);
```

3.3. Estorno e Edição

Ao atualizar um pedido, implementamos a lógica de **Estorno**:

1. Devolvemos os itens antigos para o estoque (soma).
2. Limpamos a lista do pedido.
3. Adicionamos os novos itens e baixamos o estoque novamente.

4. 🐳 Infraestrutura (DevOps)

4.1. Docker Multi-Stage Build

Criamos uma imagem otimizada (~150MB) separando a compilação da execução.

```
# Estágio 1: Build (Maven + JDK)
FROM eclipse-temurin:21-jdk-alpine AS builder
WORKDIR /app
COPY . .
RUN ./mvnw clean package -DskipTests

# Estágio 2: Run (Apenas JRE)
FROM eclipse-temurin:21-jre-alpine
WORKDIR /app
COPY --from=builder /app/target/*.jar app.jar
ENTRYPOINT ["java", "-jar", "app.jar"]
```

4.2. Docker Compose e Profiles

Ambiente híbrido configurado via variáveis de ambiente.

- **Perfil** `default` **:** Banco H2 (Arquivo).
- **Perfil** `pg` **:** Banco PostgreSQL (Docker).

Snippet do `docker-compose.yml` **(Ponte de Variáveis):**

```
environment:
  - SPRING_PROFILES_ACTIVE=pg
  # Injeta as variáveis do .env para dentro do Java
  - JWT_SECRET=${JWT_SECRET}
  - GOOGLE_CLIENT_ID=${GOOGLE_CLIENT_ID}
```

5. 🧪 Qualidade e Testes

5.1. Testes Unitários

Foco em testar a **Regra de Negócio** (Service), mockando o Banco e o Mapper.

- **Ferramentas:** JUnit 5, Mockito.
- **Padrão:** AAA (Arrange, Act, Assert).
- **Dica:** Usar `verify(repository, never()).save(any())` para garantir que dados inválidos não tocam no banco.

5.2. Data Seeder

Criamos um `CommandLineRunner` com perfil `!prod` para popular o banco automaticamente com usuários, produtos e pedidos históricos, facilitando o desenvolvimento e testes de relatórios.

6. 📊 Performance (Paginação e Relatórios)

6.1. Paginação Inteligente

Substituímos `List<T>` por `Page<T>` para evitar *Out of Memory*. No Service, usamos `.map()` para converter DTOs mantendo os metadados da página.

6.2. Native Queries (Relatórios)

Para agregações complexas (Soma de Vendas por Produto), ignoramos o JPQL e usamos SQL Nativo para performance máxima.

```
@Query(value = """
    SELECT p.name, SUM(oi.total)
    FROM order_items oi
    JOIN products p ON ...
    GROUP BY p.name
    ORDER BY SUM(oi.total) DESC
""", nativeQuery = true)
List<SalesReportDTO> getRelatorio();
```

7. 📝 Glossário Técnico do Projeto

- **Idempotência:** Garantia de que repetir uma operação (como pagar um pedido já pago) não gera efeitos colaterais indesejados.
- **Stateless:** A API não guarda estado do cliente. Toda requisição deve conter o Token.
- **Bean:** Um objeto gerenciado pelo Spring (Injeção de Dependência).
- **Managed Entity:** Um objeto que está sendo observado pelo Hibernate (mudanças nele refletem