Aula 4: Persistência de Dados e Lógica de Negócios

Esta aula aprofunda a camada de dados e de modelo, focando em como as aplicações Java interagem com bancos de dados de forma eficiente e organizada.

Conceitos Fundamentais de Persistência

- **Serialização**: É o processo de converter um objeto Java em uma sequência de bytes, essencial para armazená-lo em arquivos, transmiti-lo pela rede ou salvá-lo em um banco de dados. Isso é feito implementando a interface `Serializable`, que, embora vazia, sinaliza à JVM que a classe pode ser serializada.
- **Java Persistence API (JPA)**: É uma especificação padrão do Java que simplifica a persistência de dados, fornecendo uma ponte entre o mundo orientado a objetos do Java e o modelo relacional dos bancos de dados.
- **Object-Relational Mapping (ORM)**: É a técnica implementada pela JPA (e por frameworks como o Hibernate) que mapeia classes Java para tabelas de banco de dados e objetos para linhas nessas tabelas. O ORM resolve o problema da "incompatibilidade de impedância objeto-relacional", permitindo que desenvolvedores manipulem objetos em vez de escrever consultas SQL diretamente.

Mapeamento de Entidades com Anotações JPA

- **Entidades**: São classes Java simples (POJOs) que representam tabelas no banco de dados. A anotação **`@Entity`** é usada para marcar uma classe como uma entidade persistente.
- **Chave Primária**: Toda entidade precisa de uma chave primária, definida pela anotação **`@Id`**. A geração automática de seu valor pode ser configurada com **`@GeneratedValue`**.
- **Outras Anotações**:
    - `@Table`: Permite especificar um nome diferente para a tabela no banco de dados.
    - `@Column`: Permite customizar o mapeamento de um atributo para uma coluna da tabela, definindo nome, tamanho, nulidade, etc..
    - `@Transient`: Indica que um atributo da classe não deve ser persistido no banco de dados.
    - Anotações de Relacionamento (`@OneToOne`, `@ManyToOne`, etc.): Mapeiam as associações entre entidades, como relacionamentos unidirecionais e bidirecionais.

Organização da Lógica de Dados e Negócios

- **Padrão DAO (Data Access Object)**: É um padrão de projeto que **separa a lógica de acesso a dados da lógica de negócios**. Ele fornece uma interface abstrata para realizar operações CRUD (Create, Read, Update, Delete), tornando a aplicação mais modular e flexível a mudanças no mecanismo de persistência.
- **Classe de Serviço (`@Service`)**: No contexto do Spring MVC, a classe de serviço **encapsula a lógica de negócios da aplicação**. Ela atua como uma camada intermediária entre o Controller e a camada de acesso a dados (DAO ou Repositório), promovendo reutilização e organização do código.
- **Gerenciamento de Transações (`@Transactional`)**: Garante a **integridade e consistência dos dados** durante operações que envolvem múltiplas etapas no banco de dados. Uma transação assegura que todas as operações sejam concluídas com sucesso (commit) ou que, em caso de falha, todas sejam revertidas (rollback). A fonte recomenda aplicar a anotação `@Transactional` na classe de serviço para garantir que operações complexas, que envolvem múltiplos DAOs, sejam tratadas como uma única transação atômica, evitando inconsistências.

Aula 5: Testes de Software e Validação

Esta aula aborda a importância dos testes para garantir a qualidade do software, cobrindo desde testes funcionais de APIs até testes unitários de código e validação de dados de entrada.

Fundamentos e Estratégia de Testes

- **Verificação vs. Validação**: Verificação garante que o software implementa uma função corretamente, enquanto a validação garante que o software atende aos requisitos do cliente.
- **Níveis de Teste**: Os testes são organizados em uma estratégia espiral, começando pelos testes de menor escopo e avançando para os de maior escopo:
    - **Teste de Unidade**: Foca em pequenas partes do código (métodos, classes) isoladamente. Geralmente é de responsabilidade do desenvolvedor.
    - **Teste de Integração**: Verifica se diferentes unidades do sistema se comunicam e funcionam corretamente juntas.
    - **Teste de Validação**: Assegura que o software atende aos requisitos funcionais especificados.

    - **Teste de Sistema**: Testa o software como um todo, em conjunto com outros elementos do sistema.

Testes Funcionais e Ferramentas

- **Testes Funcionais**: Focam em verificar se as funcionalidades do software operam conforme as especificações e requisitos, simulando cenários de uso reais.
- **Postman**: É uma ferramenta popular para realizar **testes funcionais em APIs e serviços web**. Ele permite criar e enviar requisições HTTP (GET, POST, etc.), automatizar testes com scripts JavaScript e validar as respostas recebidas.

Validação de Dados: Front-end e Back-end

A validação é crucial para garantir a integridade, formato e segurança dos dados.

- **Validação Front-end**: Ocorre no navegador do usuário, proporcionando feedback imediato. Pode ser implementada com:
    - **JavaScript**: Oferece validação customizada e dinâmica.
    - **HTML5**: Usa atributos como `type="email"`, `pattern` (para expressões regulares) e `required` para validações mais simples e declarativas.
    - **Desvantagens**: É insegura, pois pode ser facilmente contornada pelo usuário desativando o JavaScript ou manipulando o código.
- **Validação Back-end**: Ocorre no servidor e é a **camada de segurança essencial**.
    - **Spring Validation**: No ecossistema Spring, a validação back-end é feita com a especificação **Bean Validation**. Anotações como `@NotNull`, `@Size` e `@Email` são adicionadas diretamente às classes de modelo. Ao usar a anotação `@Valid` em um método do Controller, o Spring automaticamente valida o objeto recebido antes de executar a lógica de negócios.

Teste Unitário com JUnit

- **JUnit**: É o framework padrão para testes unitários em Java. Ele permite que desenvolvedores criem casos de teste de forma simples, usando anotações como `@Test` e métodos de asserção (ex: `assertEquals`) para verificar se o comportamento do código está correto. A execução automatizada dos testes ajuda a identificar problemas de forma precoce e a garantir a estabilidade do código ao longo do tempo.

Aula 6: APIs, Web Services e Documentação

A última aula foca na comunicação entre sistemas, explicando os conceitos de API e Web Services, a arquitetura REST, o formato JSON e a importância da documentação.

Diferença entre API e Web Service

- **API (Interface de Programação de Aplicações)**: É um conceito amplo que define um conjunto de regras e padrões para que diferentes softwares possam interagir entre si. Uma API pode operar localmente (como a JDBC para bancos de dados) ou pela rede.
- **Web Service**: É um **tipo específico de API que opera na web**, utilizando protocolos como HTTP para permitir a comunicação entre sistemas distribuídos. **Todo Web Service é uma API, mas nem toda API é um Web Service**.

Arquiteturas e Tecnologias de Web Services

- **SOAP (Simple Object Access Protocol)**: Um protocolo mais antigo e rígido, baseado em XML, para troca de mensagens estruturadas. É útil em cenários que exigem um alto grau de padronização.
- **REST (Representational State Transfer)**: É um **estilo arquitetural**, não um protocolo, que define um conjunto de restrições para criar serviços web escaláveis e simples. Uma API é considerada **RESTful** se seguir princípios como arquitetura cliente-servidor, comunicação *stateless* (sem estado), uso de cache e uma interface uniforme.
- **JSON (JavaScript Object Notation)**: É um formato leve e de fácil leitura para troca de dados, baseado em uma estrutura de pares chave-valor. Tornou-se o padrão de fato para APIs REST devido à sua simplicidade e compatibilidade com diversas linguagens.

Métodos e Documentação de APIs REST

- **Métodos HTTP**: APIs REST utilizam os verbos padrão do protocolo HTTP para realizar operações em recursos, como:
    - `GET`: Recuperar dados.
    - `POST`: Criar um novo recurso.
    - `PUT`: Atualizar um recurso existente (substituição completa).
    - `PATCH`: Atualizar um recurso existente (modificação parcial).
    - `DELETE`: Excluir um recurso.

- **Documentação de API**: É um guia essencial que descreve como utilizar uma API, detalhando endpoints, métodos, parâmetros e exemplos de respostas. Como muitas APIs não possuem uma interface gráfica, a documentação é a principal ferramenta para os desenvolvedores entenderem como interagir com ela. Ferramentas como o **Postman** podem ser usadas não apenas para testar, mas também para criar e manter a documentação de uma API de forma prática.