



Programação servidor com Java

A linguagem Java é muito popular e amplamente usada em grandes empresas para criar aplicativos empresariais e sistemas importantes. Isso significa que saber Java é uma habilidade muito procurada no mercado de trabalho. Dominar essa linguagem vai te preparar para atender às demandas do mercado e te dar uma base sólida para construir sua carreira como desenvolvedor de sistemas.

Prof. Marcos Alexandre Pinto de Castro Junior

Propósito

Antes de iniciar o conteúdo, configure o ambiente com a instalação do JDK e Apache NetBeans, definindo a plataforma de desenvolvimento. Também é necessário instalar o Web Server Tomcat e o Application Server GlassFish, definindo o ambiente de execução e testes. Além disso, configure a porta do servidor Tomcat como 8084 para evitar conflitos com o GlassFish na porta 8080.

Objetivos

- Identificar características de Web Servers no ecossistema Java.
- Identificar características de Application Servers no ecossistema Java.
- Empregar as tecnologias Servlet e JSP na construção de aplicativos servidores.
- Empregar a tecnologia JDBC para viabilizar o acesso ao banco de dados.

Introdução

No contexto do ambiente de desenvolvimento de sistemas web, vamos analisar os princípios funcionais de sistemas cliente-servidor, no modelo web, e veremos como são configurados os servidores Tomcat (Web Server) e GlassFish (Application Server), bem como os aplicativos executados nas duas plataformas.

Após compreender o ambiente de execução, estudaremos os componentes Servlet e JSP, utilizados para implementar aplicativos servidores. A partir disso, aprenderemos como conectar nossos aplicativos a bancos de dados usando o JDBC (Java Database Connectivity).

Introdução



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

O protocolo HTTP

O protocolo HTTP (Hypertext Transfer Protocol) pertence à camada de aplicação do modelo OSI (Open System Interconnection), definido originalmente para suportar páginas de hipertexto baseadas na sintaxe HTML. É muito comum que nossas aplicações tenham a necessidade de se comunicar com outras, normalmente via web. Na maioria das situações os desenvolvedores optam por implementar comunicação via protocolo HTTP. Por isso, o conhecimento do protocolo de comunicação é tão importante para os profissionais de desenvolvimento.

Neste vídeo, você verá o protocolo HTTP, o mais utilizado para aplicações web, e entenderá a importância do conhecimento desse protocolo para desenvolvimento de servidores.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Métodos HTTP

No protocolo HTTP, as informações podem ser transmitidas de diferentes maneiras. Isso é determinado pelo método de transmissão escolhido. Os métodos mais comuns são GET e POST. No GET, as informações são enviadas por meio do próprio endereço da requisição. Já o POST permite que as informações sejam transferidas no corpo da requisição; dessa forma os dados trafegados não ficam explícitos, possibilitando algum nível de segurança na transferência dos dados.

Por exemplo, ao verificar o histórico de navegação de um usuário, é possível notar os dados trafegados por meio de requisições do tipo GET, o que não ocorre com aqueles do tipo POST.

Além dos métodos GET e POST, existem diversos outros, dentre eles: DELETE, PUT e PATCH. Os métodos HTTP também são conhecidos como verbos. Isso se deve à ação esperada para cada tipo de requisição. Por exemplo, ao realizar uma requisição do tipo DELETE, o próprio verbo (ou método) da requisição indica que o cliente deseja excluir algum recurso ou informação do servidor. Portanto, para que o cliente e outras aplicações possam usar os recursos que desenvolvemos, é necessário que nossas aplicações sigam as regras do protocolo HTTP e implementem suas funcionalidades de maneira adequada.

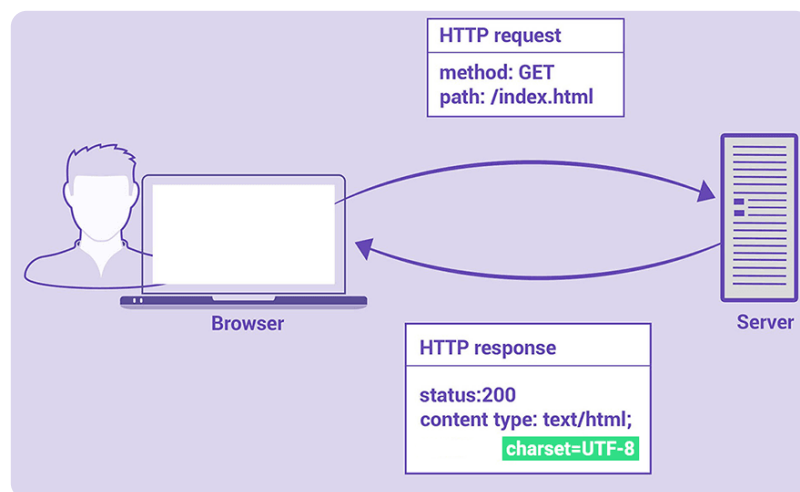


Diagrama de requisição HTTP.

Quando enviamos os dados para o servidor, temos uma requisição HTTP, que poderá iniciar algum processamento. Ao final, esse servidor deverá retornar uma resposta, que normalmente contém uma representação em formato HTML, XML ou JSON. Dentro do modelo de requisição e resposta, proporcionado

pelo HTTP, o conteúdo recebido constitui um ambiente independente do anterior, ou seja, é um **protocolo** que **não oferece** a manutenção de estados.

HTTP em aplicações web

No início da internet, tudo que tínhamos eram conjuntos de páginas estáticas com possibilidade de navegar entre elas utilizando hiperlinks; logo, o máximo de dinamismo possível era a execução de rotinas em linguagem JavaScript.

Contudo, as necessidades evoluíram levando a um novo patamar de desenvolvimento web. Passamos a gerar respostas dinamicamente, podendo utilizar dados disponíveis no servidor, como tabelas dos bancos de dados ou repositórios de arquivos. Com isso o perfil dos dados transferidos se diversificou. Atualmente, adotamos diversos formatos, como XML e JSON, utilizados principalmente na representação de dados.

Além da visualização gerada no ambiente do cliente e dos dados trafegados entre aplicações, é necessário implementar as regras de negócio. Para este fim, podemos empregar diversas linguagens de programação no servidor. Veja algumas delas a seguir!

- PHP
- Python
- Java

Essa estrutura permite a integração entre diferentes aplicações.

Atividade 1

Considerando o emprego dos métodos HTTP, analise as afirmações:

- Com as evoluções do protocolo HTTP, a utilização de métodos é opcional.
- Os métodos/verbos HTTP restringem a comunicação, pois um servidor não poderá lidar com mais de um método.
- Ao escrever aplicações, é importante utilizar os métodos apropriados a cada tipo de operação que será realizada.

A

As afirmações II e III são verdadeiras.

B

As afirmações I e III são verdadeiras.

C

Somente a III é verdadeira.

D

Somente a II é verdadeira.

E

Somente a I é verdadeira.



A alternativa C está correta.

No protocolo HTTP, todas as requisições utilizam métodos. Mesmo quando não especificado (como no caso do uso de navegadores), é comum utilizar o método GET. O servidor web deve processar requisições coerentemente com seus métodos.

Ambiente servidor para web

Existem diversos tipos de servidores. Dependendo da necessidade, podemos empregar servidores HTTP simples, capazes de servir páginas e conteúdos estáticos. Já quando precisamos manipular requisições e respostas, normalmente utilizamos servidores web ou de aplicativos. Um bom exemplo de servidor web é o Apache Tomcat.

Neste vídeo, você verá o fluxo que uma comunicação segue a partir de uma requisição enviada a um servidor Java. Além disso, irá conferir as tarefas realizadas pelo servidor web e pela aplicação Java.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Em geral, as requisições passam por uma série de etapas. Primeiramente recebemos a requisição do cliente, que é interceptada pelo servidor web e entregue à aplicação. O aplicativo captura os possíveis dados enviados na requisição e executa alguma rotina na linguagem adotada pelo servidor. Por fim, é construída a resposta, que será encaminhada ao cliente. A seguir, trataremos do caso específico de uma aplicação e um servidor web em um ambiente Java. Vamos lá!

Responsabilidades do servidor e da aplicação

O servidor web instancia um objeto do tipo **HttpServletRequest**, que representa a requisição HTTP e permite efetuar qualquer tipo de processamento. Esse processamento ocorrerá por meio de classes que implementam a lógica da aplicação. A resposta é representada por um objeto do tipo **HttpServletResponse**, que também pode ser processado de diversas formas. Observe no exemplo um trecho de código que processa requisições e respostas.

```
java

public class MinhaAplicacao {
    protected void processa(HttpServletRequest request, HttpServletResponse response) {
        String parametro = request.getParameter("parametro");
        response
            .getWriter()
            .println(
                "Parametro: " +
                parametro +
                "");
    }
}
```

O código acima extrai o valor do "parametro" da requisição e insere um texto HTML na resposta que será retornada ao cliente. Repare que a classe não possui um método "main" ou um retorno, pois o responsável por instanciar e executar a classe será o servidor web.



Dica

Tendo em vista que o protocolo HTTP não permite o armazenamento de estados entre requisições, a melhor solução para manter os estados são as sessões, que correspondem a objetos alocados no servidor, fazendo referência à conexão. Enquanto o usuário estiver no site, todos os dados atribuídos serão mantidos, sendo eliminados após a perda da conexão.

No ambiente Java, um objeto da classe **HttpSession** permite a gerência de sessões HTTP, gerando consumo de memória do servidor. Essa prática deve ficar restrita a finalidades específicas.

Atividade 2

Avalie as afirmações e assinale a resposta correta:

- I. Servidores web como Tomcat possuem limitações que nos permitem apenas servir páginas estáticas.
- II. Ao desenvolver aplicações para web é necessário sempre desenvolver a camada de comunicação entre essas aplicações e os clientes.
- III. Uma aplicação web consiste em classes, métodos e objetos capazes de manipular requisições e respostas HTTP.

A

As afirmações II e III são verdadeiras

B

As afirmações I e III são verdadeiras.

C

Somente a III é verdadeira.

D

Somente a II é verdadeira.

E

Somente a I é verdadeira.



A alternativa C está correta.

Além de servir páginas estáticas e diretórios, os servidores Web Java abstraem parte do tratamento das requisições e respostas por meio das classes **HttpServletRequest** e **HttpServletResponse**. Isso permite que

o servidor utilize aplicações web para implementar todo tipo de processamento. Assim, o desenvolvedor volta-se apenas para a implementação da lógica.

Web Server Tomcat

O Tomcat é um projeto da Apache Software Foundation voltado para a definição de um servidor web, com código aberto e uso de tecnologias Java.

Neste vídeo, você verá o conceito de servidor web ou container. Para isso, será utilizado como exemplo o servidor web Apache Tomcat, por ser um dos mais utilizados no mercado. Além disso, entenderá os principais diretórios e seus respectivos papéis em um ambiente que utiliza Tomcat, com ênfase no diretório webapps e seu papel no deploy de aplicações.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

O Tomcat oferece um ambiente consistente para manipular a comunicação HTTP, além de suportar tecnologias Java nativas no container web. Por ser um produto de código aberto, tornou-se o padrão para hospedagem de sistemas Java para web, oferecendo a possibilidade de executar de modo totalmente independente, ou atuando como módulo plugável em servidores de aplicativos Java, uma estratégia adotada pelos servidores JBoss e GlassFish.



Atenção

Um componente que implemente a interface Servlet é uma classe que permite ampliar as funcionalidades básicas de um servidor, ou seja, na prática, é um aplicativo plugável, que deve ser executado em ambiente específico, como o container web oferecido pelo Tomcat. Por meio de um Servlet, temos o processamento no servidor, permitindo a geração dinâmica de conteúdo para a resposta HTTP.

Já as páginas JSP (Java Server Pages) permitem uma sintaxe com base em HTML e XML. Assim, trechos dinâmicos são intercalados por meio de fragmentos de código Java denominados Scriptlets, que são executados no servidor. Apenas a forma de escrita é modificada, pois as páginas JSP são transformadas em Servlets pelo container web, quando ocorre o primeiro acesso, o que nos leva ao entendimento de que servidores como o Tomcat sempre utilizam Servlets para a geração de conteúdo dinâmico.

Diretórios do Tomcat

A complexidade da arquitetura do Tomcat é refletida na grande quantidade de arquivos e diretórios gerados na instalação padrão. Conheça a seguir a lista com os principais diretórios do Web Server Tomcat e suas respectivas utilizações.

Bin

Binários estruturais do servidor, agrupados em arquivos no formato jar, e scripts para inicialização ou término da execução.

Conf

Arquivos de configuração, como server.xml, que guardam os parâmetros gerais do Tomcat, incluindo a porta utilizada para comunicação.

Lib

Bibliotecas de inicialização do servidor, no formato jar, que também ficam disponíveis para todos os aplicativos do ambiente.

Logs

Arquivos de log, extremamente úteis para a identificação dos erros que ocorreram durante a execução do servidor.

Webapps

Diretório de base para a instalação dos aplicativos Java web, ou seja, define a raiz do site. Para incluir um novo aplicativo, basta copiar um arquivo de extensão ".war" neste diretório; o Tomcat fará o "deploy" do aplicativo, criando o respectivo diretório e servindo a aplicação.

Atividade 3

O servidor Apache Tomcat é muito utilizado no mundo Java para desenvolvimento web. Contudo, não é o único container existente no mercado. Nesse contexto, assinale a alternativa correta.

A

O Tomcat é bastante utilizado por ser de código fechado, provendo suporte a baixo custo.

B

Aplicações desenvolvidas para o Apache Tomcat não funcionam em outros tipos de containers, devido às suas especificidades.

C

Servidores web implementam diferentes especificações de Servlet, o que dificulta a mudança de servidor.

D

O Apache Tomcat implementa a especificação "Servlet", amplamente difundida no mundo Java.

E

O Tomcat permite execução de código Java, enquanto os outros containers possuem apenas suporte a páginas HTML.



A alternativa D está correta.

No geral, os servidores web do mundo Java devem implementar a especificação Servlet. Portanto, o Apache Tomcat segue o mesmo padrão, entregando as interfaces previstas nesse tipo de especificação, como `HttpServletResponse` e `HttpServletRequest`.

Estrutura de um aplicativo web

Os aplicativos que criaremos aqui deverão obedecer à estrutura exigida pelo Tomcat, composta por um diretório de base, com o nome do aplicativo, contendo um subdiretório de nome `WEB-INF` e outro com o nome `META-INF`. Acompanhe mais detalhes no vídeo.

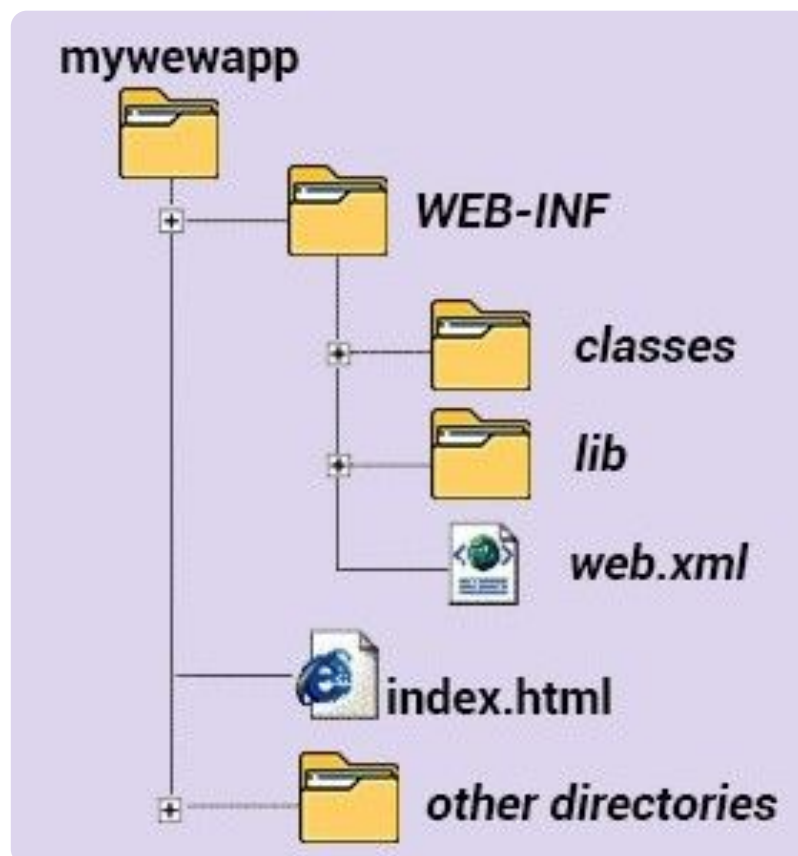
Neste vídeo, você verá a estrutura necessária para que um projeto web possa ser integrado ao Tomcat. Além disso, irá conferir como a nossa IDE pode facilitar a criação dessa estrutura e a integração com o container Apache Tomcat.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Observe como se organiza a estrutura de um aplicativo Java web:



Estrutura de um aplicativo web.

O Tomcat fornece uma interface padronizada, facilitando o trabalho do desenvolvedor. Contudo, também exige que a aplicação respeite determinada estrutura.

Vamos tratar agora dos principais componentes de um projeto web baseado no Tomcat. Acompanhe!

Diretório raiz

Neste diretório, temos páginas JSP, HTML, XML, e outros formatos interpretados, além de podermos acrescentar subdiretórios para a organização geral do conteúdo. É comum definir diretórios para conteúdo estático, como mídia e estilização, permitindo que as bibliotecas e temas sejam isolados, como no caso do JQuery, o que facilita a atualização de versões e a mudança do aspecto do sistema.

Diretório WEB-INF

Neste diretório, encontramos o arquivo de configuração com o nome web.xml, um subdiretório com o nome classes para os arquivos compilados do Java e um subdiretório lib com as bibliotecas. Quando criamos um Servlet, o arquivo compilado da classe Java é copiado para classes e o reconhecimento pelo servidor pode ser configurado, utilizando o arquivo web.xml para mapeamento.

Diretório META-INF

Neste diretório, são definidas as configurações de ambiente, com recursos oferecidos para o aplicativo a partir do Tomcat. Podemos ter arquivos como context.xml, com informações gerais de contexto.

Atividade 4

Quando criamos um aplicativo web na plataforma Java, o Tomcat é muito utilizado como Web Server, e a preferência é justificada por algumas de suas características estruturais e funcionais. Avalie as afirmações a seguir e marque a opção correta.

I. O Tomcat dá suporte a diversas tecnologias Java, incluindo Servlets e JSPs.

II. O Tomcat funciona apenas de maneira isolada e não permite trabalhar como módulo "plugável" do GlassFish.

III. O Tomcat apresenta uma configuração muito flexível, a partir de arquivos XML.

A

As afirmações II e III são verdadeiras

B

As afirmações I e III são verdadeiras.

C

Somente a III é verdadeira.

D

Somente a II é verdadeira.

E

Somente a I é verdadeira.



A alternativa B está correta.

O Tomcat suporta tecnologias como Servlet e JSP, simplificando o desenvolvimento web. Além de funcionar de maneira isolada, o Tomcat pode ser incorporado a servidores de aplicação, como o Glassfish. Além da interface gráfica e da integração com IDEs para a configuração do Tomcat, podemos utilizar os arquivos XML, dando mais flexibilidade na configuração.

Primeira aplicação web com Java

Além de entender como funciona o ecossistema Java, precisamos nos beneficiar dessa gama de tecnologias para resolver problemas reais. Como já vimos, as diversas tecnologias se integram para facilitar cada vez mais o desenvolvimento. Porém o desenvolvedor precisa entender a estrutura do aplicativo para, então, utilizar abstrações para ganhar eficiência. Assim, criaremos manualmente a estrutura de diretórios e os arquivos, para que possamos compreender a estrutura da aplicação web.

Neste vídeo, você verá a criação de uma aplicação simples para web com Tomcat. Essa implementação será realizada de duas formas, primeiramente, com a criação dos diretórios e arquivos de forma manual; e depois, por meio da utilização dos recursos visuais da IDE Apache netbeans.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

Uma aplicação Java compatível com o Tomcat deve respeitar uma estrutura de diretórios padronizada. Agora vamos testar como isso funciona "por baixo dos panos", antes de criar um projeto no IDE. Na verdade, é sugerido que você não use sua IDE neste momento. Você pode criar as pastas e os arquivos em qualquer outro editor de código.

Primeiramente, crie um diretório raiz chamado "teste_tomcat". Dentro desse diretório, teremos os diretórios "WEB-INF" e "META-INF". Por fim, criaremos os arquivos seguindo alguns passos. Vamos conferi-los!

1. "WEB-INF/web.xml": Insira a tag `<?xml?>` e a tag `<web-app>` vazia.
2. Inclua o arquivo de contexto "META-INF/contexto.xml" vazio.
3. Crie um arquivo chamado "index.html" no diretório raiz do projeto.
4. Copie o diretório raiz "teste_tomcat" para o diretório "webapps" do seu Tomcat.
5. Reinicie o Tomcat e verifique o seu browser por meio do endereço `http://localhost:8080/teste_tomcat`.

O resultado dessa prática será:

Arquivo WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>

</web-app>
```

"index.html"

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Olá mundo!</title>
</head>
<body>
<h1>Olá mundo!</h1>
</body>
</html>
```

Estrutura de arquivos e diretórios



Faça você mesmo!

Qual a estrutura de diretórios e arquivos necessária para criar uma aplicação Java compatível com o Tomcat?

A

Diretório raiz "teste_tomcat", "WEB-INF/web.xml", "META-INF/contexto.xml" e "index.html".

B

Arquivos "WEB-INF", "META-INF", "web.xml" e "index.html".

C

Diretório raiz "teste_tomcat", "WEB-INF/web.xml" e "index.html".

D

Arquivos "WEB-INF", "META-INF" e "contexto.xml".

E

Diretório raiz "teste_tomcat" e "index.html".



A alternativa A está correta.

Para que a página funcione da maneira esperada, precisamos criar a estrutura de diretórios e arquivos idêntica à do exemplo, sendo o arquivo "META-INF/contexto.xml" um item opcional.

Servidores de aplicativos

Application Servers ou servidores de aplicativos são plataformas capazes de suportar projetos de grande porte, com alto nível de conectividade, processamento paralelo e distribuído entre diversos outros elementos comuns para um ambiente empresarial complexo.

Neste vídeo, você verá o conceito de servidores de aplicativos, a partir da citação de algumas das principais demandas de um ambiente corporativo, e entenderá como esse tipo de servidor entrega soluções para tais demandas.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

O objetivo de um servidor de aplicativos é tratar as necessidades comuns dos ambientes de execução, como segurança, balanceamento de carga e alta disponibilidade. Eles oferecem um conjunto de containers que aplicam essas soluções, permitindo ao programador se concentrar nas regras de negócio e abstraindo a responsabilidade sobre as funcionalidades comuns.

Servidores de aplicativos podem trabalhar de forma conjunta, com base em protocolos voltados para o processamento distribuído, o que traz mais dificuldades na **gerência de exceções**, já que podem ocorrer de forma local ou remota. Com a possibilidade da ocorrência de exceções em vários locais distintos, surge a necessidade de bibliotecas voltadas para **transações distribuídas**.



Atenção

A transação é um processo atômico e isolado, que deve ser executado de forma consistente, e com resultados duráveis.

Uma necessidade básica do ambiente corporativo é a **gerência da segurança**, exigindo ferramentas robustas para a **autenticação** e **autorização** de usuários. Além de gerenciar o acesso, nosso servidor deve ser capaz de oferecer protocolos seguros, como TLS (Transport Layer Security).

Como os servidores de aplicativos devem lidar com diversos fluxos de dados que fazem parte do ambiente corporativo, um requisito primário é a presença de um grande conjunto de bibliotecas de middleware, garantindo a interoperabilidade com outros sistemas. Pools de conexões e canais de mensagerias são exemplos de componentes de middleware.

Devido à grande quantidade de componentes e recursos compartilhados, precisamos de individualização. Para criar nomes únicos e gerenciá-los, utilizamos os **serviços de nomes** e **diretórios**. Para servidores criados na

plataforma Java, o JNDI (Java Naming and Directory Interface) funciona como um canal de identificação entre os aplicativos, tanto interna quanto externamente, centralizando toda a integração entre plataformas.



Usuário utilizando servidor de aplicativo.

Temos diversas opções de servidores de aplicativos Java, como o GlassFish, sempre com base no JEE (Java Enterprise Edition), uma arquitetura de referência para ambientes corporativos, com suporte a objetos distribuídos.

Atividade 1

Com a necessidade de diversos serviços integrados, é comum a utilização de Application Servers. Esse tipo de serviço é bastante diferente dos servidores web. Nesse contexto, analise as afirmações a seguir e assinale a alternativa correta.

- I. Servidores web possuem mais recursos, sendo ideais para ambientes corporativos.
- II. Servidores de aplicação possuem diversos componentes, como Web Servers.
- III. Servidores web possuem serviços de diretórios para o gerenciamento de seus componentes.

A

As afirmações II e III são verdadeiras.

B

As afirmações I e III são verdadeiras.

C

Somente a III é verdadeira.

D

Somente a II é verdadeira.

E

Somente a I é verdadeira.



A alternativa D está correta.

Servidores web são limitados, implementando apenas a especificação Servlet. Alguns servidores podem ser utilizados como módulos dos servidores de aplicação. Por serem mais simples, os servidores web não disponibilizam recursos para gerenciamento de serviços de diretório.

Tecnologias de objetos distribuídos

Um ponto central na maioria dos Application Servers são sistemas de objetos distribuídos, como CORBA (Common Object Request Broker Architecture), ou Microsoft DCOM (Distributed Component Object Model). No caso do Java, temos os componentes do tipo EJB (Enterprise Java Bean).

Neste vídeo, você irá conferir a existência de diversos recursos distribuídos que podem ser utilizados em uma aplicação web com base no ecossistema Java. A partir disso, entenderá os serviços protocolos e descritores, que podem variar bastante conforme cada tecnologia.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Processamentos distribuídos são serviços oferecidos de forma remota, usando protocolos próprios, como RPC (Remote Procedure Call) e RMI (Remote Method Invocation). Vejamos alguns elementos comuns à maioria das arquiteturas:

- Protocolo de comunicação
- Serviço de registro e localização
- Descritor de serviços

Entre os processamentos distribuídos, temos os **Web Services**, como o SOAP (Simple Object Access Protocol), que utiliza um protocolo de mesmo nome; o **WSDL** (Web Service Description Language), usado para descrever os serviços; e o **UDDI** (Universal Description, Discovery and Integration), utilizado nas tarefas de registro e localização. Essas tecnologias trabalham com **XML** e têm como objetivo **garantir** a interoperabilidade entre diferentes sistemas.

Os objetos distribuídos possuem o mesmo tipo de fornecimento de serviços, de forma distribuída, mas com a execução efetuada a partir de pools. No CORBA temos a utilização do protocolo de rede IIOP (Internet Inter-ORB Protocol), descritor IDL (Interface Definition Language) e registro via COS (CORBA Object Services) Naming.

Tecnologia	Registro e localização	Descritor	Protocolo
CORBA	COS Naming	IDL	IIOP
SOAP	UDDI	WSDL	SOAP
RMI	JNDI	Interface Java	RMI
EJB	JNDI	Interface Java	RMI-IIOP

Tabela: Componentes arquiteturais para serviços distribuídos.

Marcos Alexandre Pinto de Castro Junior.

O registro dos EJBs utiliza JNDI e a comunicação utiliza o protocolo misto RMI-IIOP. Quanto ao descritor de serviços, utilizamos interfaces Java, mas pode ser gerado o IDL a partir das interfaces.

Agora, acompanhe um fluxo comum de utilização dos serviços distribuídos. Vamos lá!

1. Localizar o serviço ou objeto distribuído a partir do serviço de nomes.
2. Recuperar o descritor a partir da localização encontrada.
3. Gerar o cliente de comunicação a partir do descritor.
4. Transmitir as solicitações, por meio do cliente, com base no protocolo.

O cliente é gerado de forma automática a partir do descritor, permitindo que nossas chamadas ao cliente se assemelhem a solicitações locais, abstraindo a comunicação remota.

Durante a execução dos objetos, o container fornece todos os recursos necessários para autenticação, autorização, acesso à **middleware** entre outros. Com base em um container, temos a execução controlada e robusta dos processos de negócio do sistema.

Confira, logo após a atividade, os componentes arquiteturais mais comuns.

Atividade 2

Marque a alternativa que contém o protocolo utilizado pelo CORBA.

A

IIOP

B

SOAP

C

RMI

D

RMI-IIOP

E

ACI



A alternativa A está correta.

No ambiente CORBA, temos o descritor de serviços do tipo IDL, o registro e a localização pelo COS Naming, e o protocolo IIOP.

Application Server GlassFish

Enquanto o Tomcat suporta apenas Servlets e JSPs, atuando como um Web Server, o GlassFish vai além, oferecendo suporte às tecnologias Java de objetos distribuídos, no caso, os EJBs, sendo classificados como Application Server.

Neste vídeo, você irá conferir uma introdução ao servidor de aplicação GlassFish, observando seus principais componentes e sua estrutura de diretórios.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

O Tomcat é utilizado pelo GlassFish como módulo interno, delegando para o Web Server a comunicação HTTP e tratamento de Servlets e JSPs, enquanto o GlassFish trata das diversas tecnologias do JEE (Java Enterprise Edition).

Com base no GlassFish, somos capazes de criar sistemas mais complexos, com uso de EJBs e transações distribuídas, além de obtermos ferramentas para gerenciamento de componentes corporativos. O servidor também disponibiliza um ambiente de testes simplificado para **Web Services** do tipo SOAP.



Atenção

O ponto central do servidor é o container EJB, que gerencia os objetos distribuídos e facilita o acesso aos recursos. A configuração dos recursos para os EJBs é feita por meio de arquivos XML e anotações de código, utilizando identificadores JNDI.

Como todas as bibliotecas de middleware são gerenciadas pelo servidor, o processo é o mesmo para a maioria dos recursos. De forma geral, temos duas etapas: a **criação** do recurso e o **registro** no JNDI. Depois disso, o container pode utilizar o recurso com base nas configurações definidas para os EJBs.

Entre as bibliotecas e os elementos de middleware do GlassFish, podemos destacar o JDBC (Java Database Connectivity) para bancos de dados; e o JMS (Java Message Service), gerenciando mensagens a partir de canais e filas, além de bibliotecas para o controle de transações locais e distribuídas, como JTA (Java Transaction API) e JTS (Java Transaction Service).

A distribuição padrão do GlassFish inclui o serviço de mensageria GlassFish Message Queue, o banco de dados Derby (ou Java DB) e uma ferramenta de inicialização pelo console.

Agora, vamos conhecer os diretórios do Application Server GlassFish!

Bin

Diretório que contém a ferramenta asadmin para inicialização pelo console.

Glassfish

Diretório do servidor de aplicativos, com os componentes estruturais, arquivos de configuração e domínios.

Javadb

Distribuição do banco de dados Derby, contendo os subdiretórios bin e lib, com scripts de gerenciamento e bibliotecas.

Mq

Diretório que contém os componentes da mensageria GlassFish MQ e diversos exemplos de utilização.

Ao trabalhar com o servidor GlassFish, precisamos definir **domínios**, que são ambientes completos e independentes para a instalação e execução de aplicativos.

Atividade 3

O GlassFish permite acesso a um grande conjunto de tecnologias Java, disponibilizadas a partir de container próprio. Qual das tecnologias é voltada para a comunicação com mensagens?

A

JDBC

B

JMS

C

JTA

D

JTS

E

JVMS



A alternativa B está correta.

O JMS (Java Message Service) é o middleware do ambiente Java para recursos de mensagerias.

Estrutura do aplicativo corporativo

Um aplicativo corporativo comporta elementos como EJBs (Enterprise Java Beans), Servlets e JSPs. Geralmente, os objetos de negócio e as classes de acesso ao banco de dados são compactados em um arquivo jar, enquanto a interface web é organizada em um arquivo war, sendo os dois arquivos compactados em um terceiro, com extensão **ear**.

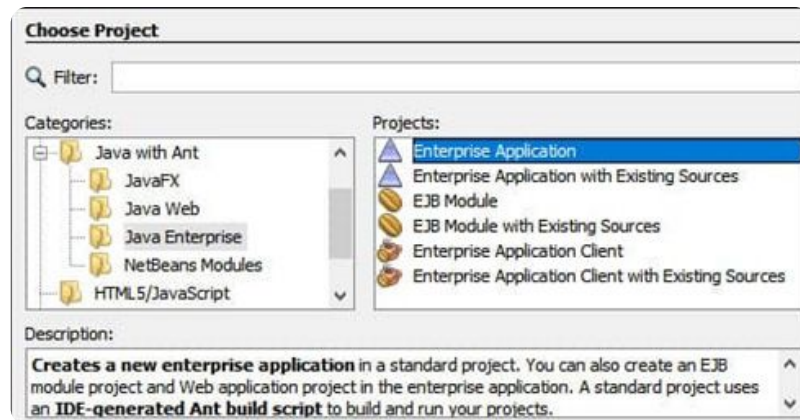
Neste vídeo, você verá a estrutura de um aplicativo corporativo para "deploy" em um servidor de aplicação. Também verá a necessidade de uma solução mais robusta em detrimento de uma mais simples (baseada em Servlets e JSP).



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

A estrutura padrão para implantação é obedecida pelo NetBeans, quando criamos um projeto do tipo **Enterprise Application**. Veja!



"Wizard" de criação de aplicações.

Na prática, são criados um projeto principal, com extensão ear, e dois projetos internos, com as extensões jar e war. Para implantar e testar o aplicativo, devemos utilizar sempre o projeto principal, representado por um triângulo.

Geralmente desenvolvemos as classes DAO (Data Access Object) e EJBs no projeto jar. Como o projeto trata da camada de negócio, não devem ser definidos elementos de interface nesse nível. Já a camada de interface web deve ficar alocada no projeto war, contendo elementos como Servlets e JSPs.



Comentário

Projetos mais simples, que não envolvem EJBs e demais elementos corporativos podem implementar apenas o projeto com extensão war. Nesse caso, temos todas as classes no mesmo projeto, organizadas por meio de pacotes.

Atividade 4

Com relação à divisão de tarefas dos componentes de uma aplicação corporativa, analise as afirmações e assinale a alternativa correta.

- I. As classes DAO devem ser definidas no projeto war, pois fazem parte do aplicativo web.
- II. Aplicativos simples podem ser definidos apenas com EJBs.
- III. A interface web deve ser implementada na camada war do projeto.

A

Apenas a afirmação III está correta.

B

As afirmações II e III estão corretas.

C

Todas as afirmações estão corretas.

D

As afirmações I e II estão corretas.

E

Apenas a afirmação II está correta.



A alternativa A está correta.

Classes DAO e EJBs devem ser implementados na camada EJB, enquanto a interface web é implementada por meio da camada war, com a exceção de aplicativos simples, que podem conter apenas a camada war.

Primeiros passos com um Application Server

Entender completamente os recursos de um servidor de aplicação pode ser desafiador inicialmente. No entanto, com o tempo e a prática, seu uso se tornará mais natural. Além disso, é possível começar a usar servidores de aplicação mesmo em projetos de aprendizado.

Neste vídeo, você verá a criação de um aplicativo simples baseado no Application Server GlassFish. A partir disso, entenderá como a integração com servidor de aplicação e a própria criação do aplicativo podem se tornar algo bastante interessante devido aos recursos presentes na IDE NetBeans.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

Utilizaremos aqui o servidor GlassFish para implementar uma página que exibirá uma mensagem (“Olá Mundo!!!”) gerada por uma classe Java. O aplicativo deve possuir uma mensagem de texto em memória, que será criada na camada EJB. Já a apresentação consistirá em uma página responsável pela apresentação. Para isso, seguiremos os seguintes passos:

1. Definição de um EJB responsável por criar a mensagem.
2. Criação de uma página JSP capaz de exibir essa mensagem.

Parece uma atividade bem simples, mas envolve conceitos avançados como EJBs e JSP. Portanto, aproveite a atividade para se familiarizar com essas tecnologias.

Vejamos o resultado dessa prática!

Arquivo index.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%%>
<%@page import="javax.Message"%%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"%%>
<title>JSP Page</title>
</head>
<body>
<% out.println(new Message().mensagem);%>
</body>
</html>
```

Arquivo Mensagem.java

```
package ejb;  
import javax.ejb.Stateless;  
import javax.ejb.LocalBean;  
@Stateless  
@LocalBean  
public class Mensagem {  
    public String mensagem = "Olá mundo!!!";  
}
```

Saída do aplicativo



Atividade 5

No exemplo, utilizamos o método "retornaMensagem" da classe "Mensagem" dentro de uma página JSP. Como podemos realizar a importação de outra classe chamada "Empresas", localizada no pacote "negocio" do subprojeto EJB?

A

import negocio

B

<%@page import="Empresa"%>

C

<%@page import="negocio.Empresa"%>

D

<%@page import="ejb.negocio.Empresa"%>

E

<%@import "negocio.Empresa"%>



A alternativa C está correta.

Para importar os pacotes do subprojeto EJB, utilizamos a sintaxe <%@page import="" %> com um import indicando a classe e o pacote utilizados.

Aplicativos web no Netbeans

Nesse contexto de estudos, um aplicativo corporativo não será necessário, já que estamos analisando tecnologias voltadas para a camada web. Precisaremos apenas de um aplicativo web, com extensão war, e o uso de um servidor, como Tomcat ou GlassFish. Acompanhe mais detalhes no vídeo!

Neste vídeo, você verá a criação de um projeto que servirá de base para o restante do estudo. Visando facilitar o desenvolvimento e a visualização do projeto, será empregado o "autodeploy", utilizando a própria IDE para gerenciar a aplicação e o servidor Glassfish.



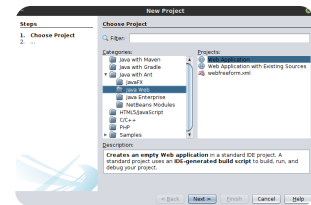
Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Vamos criar um projeto de testes conforme os passos a seguir.

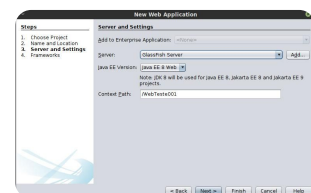
Escolha o projeto do tipo Web Application, na categoria Java web

"Wizard" de criação de projetos do NetBeans.



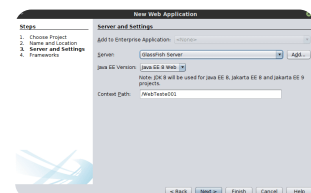
Defina o nome do projeto (WebTeste001)

"Wizard" de criação de projetos do NetBeans.



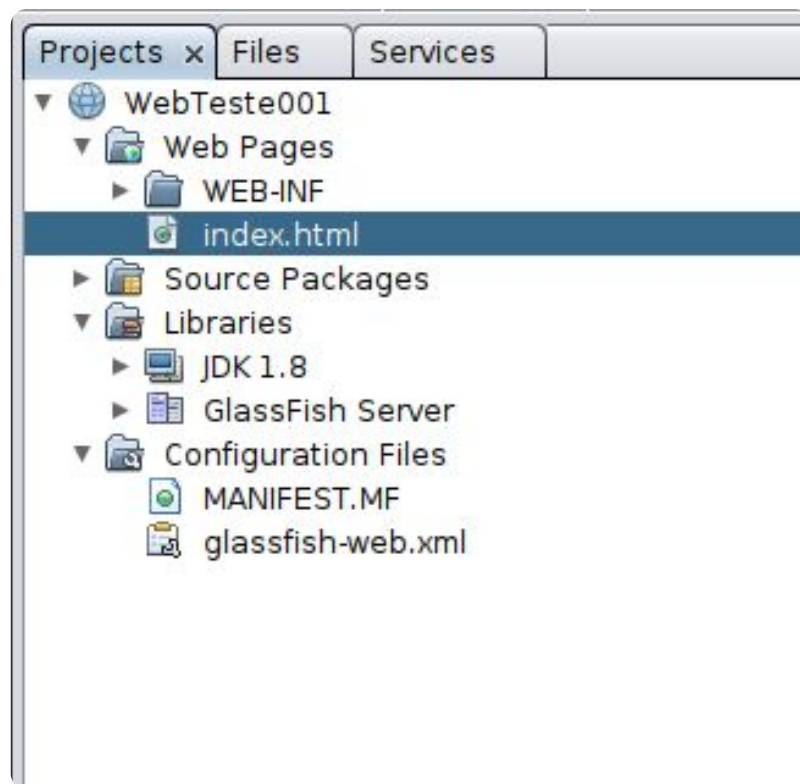
Escolha o servidor (GlassFish) e a versão do JEE (Java EE 8 Web)

"Wizard" de criação de projetos do NetBeans.



Ao término da execução desses passos, o novo projeto será apresentado na aba Projects do NetBeans. Pode ser necessário acrescentar manualmente a biblioteca Java EE 8 API, caso não esteja instalada.

Observe que o arquivo web.xml não é apresentado na estrutura do projeto, o que se deve ao fato de termos utilizado o JEE versão 8, onde as configurações são efetuadas, em sua maioria, por meio de anotações no código Java.



Navegador do NetBeans – aba de projetos.

Entenda a definição de cada um destes diretórios:

Web Pages

Elementos interpretados, como XHTML, JSP (Java Server Pages), bibliotecas Java Script e folhas de estilo CSS.

Source Packages

Elementos compilados, definidos em termos de pacotes e classes Java, incluindo nossos Servlets.

Libraries

Bibliotecas Java requeridas pelo aplicativo web.

Configuration files

Arquivos de configuração, como MANIFEST.MF e glassfish-web.xml.

Agora podemos começar a criar os elementos constituintes de nosso sistema, incluindo Servlets e JSPs, e executar o projeto. Na execução, teremos a geração do arquivo war, o servidor será iniciado, caso ainda não esteja ativo, e o arquivo compilado será copiado para o diretório de autodeploy, ocorrendo a implantação do aplicativo e abertura do navegador no endereço correspondente ao index.html.

Atividade 1

Para incluir páginas na nossa aplicação, utilizamos a divisória Web Pages. Nela, podemos implementar os componentes de apresentação do aplicativo. Marque a alternativa correta.

A

A página "index" serve como entrada, podendo ser apenas do tipo "html".

B

As Web Pages também podem conter classes Java com definição de Servlets.

C

Em Web Pages, podemos implementar apenas conteúdo estático.

D

Utilizamos arquivos do tipo "jsp" para criar páginas dinâmicas.

E

Não devemos utilizar Web Pages, pois possuem apenas páginas de administração.



A alternativa D está correta.

As JSPs (Java Server Pages) misturam sintaxe Java com estrutura HTML para criar páginas dinâmicas.

Servlets

São componentes que implementam serviços remotos, estendendo as funcionalidades de algum servidor. Dessa forma, eles representam uma solução genérica para a definição de aplicativos web, baseados principalmente no protocolo HTTP.

Confira no vídeo o conceito de Servlets, bem como a explicação do exemplo utilizado. Além disso, veremos que os servlets também são uma especificação Java. Portanto, para que um container dê suporte aos Servlets, ele deverá implementar algumas interfaces.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

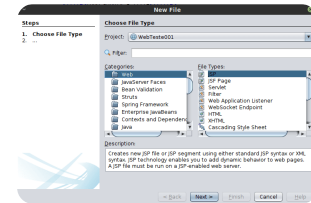
A classe `HttpServlet`, filha de `GenericServlet`, integra-se ao ambiente de execução do servidor por meio do container, utilizando tecnologias Java. Quanto à comunicação com o usuário, a classe **`HttpServlet`** apresenta métodos para responder às chamadas via protocolo HTTP.

Para processar requisições, definimos uma subclasse de `HttpServlet`, alterando os métodos `doGet` e `doPost` para personalizar este processamento.

Para definir um **Servlet no NetBeans**, adicione um **novo arquivo** e siga os passos apresentados:

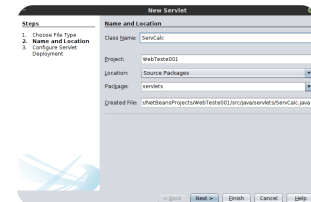
Escolha o arquivo do tipo Servlet na categoria web

"Wizard" de criação de Servlets do NetBeans.



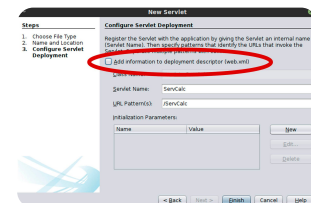
Defina o nome do Servlet (ServCalc) e do pacote (Servlets)

"Wizard" de criação de Servlets do NetBeans.

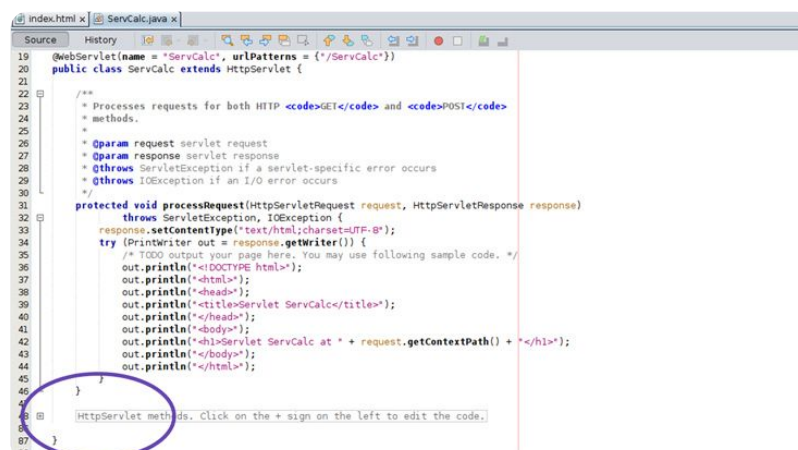


Finalize sem adicionar as informações ao arquivo web.xml

"Wizard" de criação de Servlets do NetBeans.



Teremos o código de ServCalc no editor de código do NetBeans, contendo código oculto (editor-fold) por meio de comentários no início e fim do bloco. Confira!



Código oculto pelo recurso do NetBeans.

O código oculto, engloba os métodos **doGet** e **doPost**, responsáveis pelo tratamento de chamadas HTTP. Ambos os métodos executam **processRequest**.

O próximo passo será a alteração do método **processRequest**, para que corresponda ao processamento necessário no aplicativo.

```

java

protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    try (PrintWriter out = response.getWriter()) {
        Integer a = Integer.parseInt(request.getParameter("a"));
        Integer b = Integer.parseInt(request.getParameter("b"));
        out.println("");
        out.println("A soma de " +
            a + " e " + b +
            " será " + (a + b));
        out.println("");
    }
}

```

A assinatura dos métodos `processRequest`, `doGet` e `doPost`, recebe dois tipos de parâmetros. Vamos conhecê-los!

HttpServletRequest

Representa a requisição do usuário.

HttpServletResponse

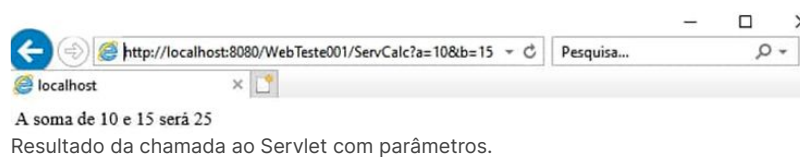
Representa a resposta HTTP.

Iniciamos a implementação de `processRequest` com a definição do tipo de saída (HTML) e codificação (UTF-8). Em seguida, iniciamos um bloco protegido, do tipo `try-with-resources`, em que o objeto de nome `out` recebe o canal de saída de resposta.

No bloco protegido recuperamos os parâmetros `a` e `b` com `getParameter`, convertendo para valores inteiros. Como o protocolo HTTP trabalha com texto, qualquer outro formato exigirá conversões no Java.

Agora vamos montar a resposta usando o `println`. Vamos criar uma página HTML que exibirá a soma dos valores fornecidos.

Para testar nosso Servlet, executamos o projeto, efetuando a chamada no endereço a seguir.



No formato original de mapeamento, acrescentaríamos uma entrada no arquivo `web.xml`, mas nas versões mais recentes do container, utilizamos uma anotação `WebServlet` na classe `ServCalc`. Veja!

plain-text

```

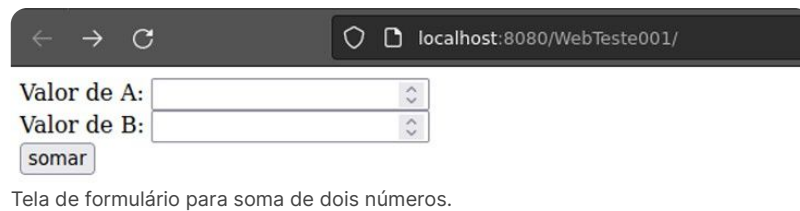
@WebServlet(name = "ServCalc", urlPatterns = {"/ServCalc"}) public class ServCalc extends
HttpServlet {

```

Para que não seja necessário escrever o endereço completo a cada chamada, podemos alterar o arquivo `index.html` da seguinte maneira:

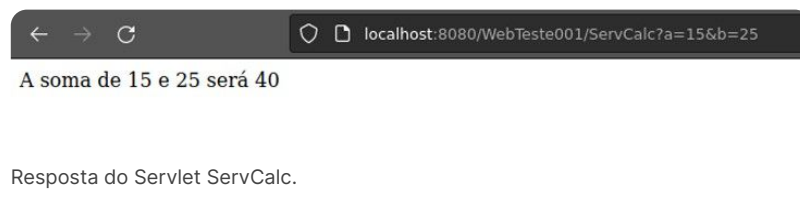
plain-text

Finalizando a modificação, teremos um formulário para a entrada dos valores e o botão de envio para o Servlet. Esse formulário é acessado pelo endereço `http://localhost:8080/WebTeste001`.



A screenshot of a web browser window. The address bar shows 'localhost:8080/WebTeste001/'. The page content includes two input fields labeled 'Valor de A:' and 'Valor de B:'. Below these fields is a button labeled 'somar'. Underneath the button, there is a line of text: 'Tela de formulário para soma de dois números.'

Informe dois números e selecione o botão "somar". Veja o resultado!



A screenshot of a web browser window. The address bar shows 'localhost:8080/WebTeste001/ServCalc?a=15&b=25'. The page content displays the result: 'A soma de 15 e 25 será 40'. Below this, there is a line of text: 'Resposta do Servlet ServCalc.'

Atividade 2

Nas versões atuais de aplicativos Java Web usamos a anotação `@WebServlet` para efetuar o mapeamento. Qual atributo da anotação define o endereço para o Servlet?

A

name

B

class

C

urlPatterns

D

Link

E

urlLink



A alternativa C está correta.

Por meio do atributo urlPatterns é definido o endereço a partir de onde o fluxo de execução será direcionado para os métodos do Servlet.

Tecnologia de JSPs

Construir páginas complexas utilizando o **println** pode ser bastante problemático devido à mistura de linguagens. Para isso, utilizamos o JSP (Java Server Pages) para construir conteúdo HTML ou XML utilizando blocos de sintaxe Java, gerando conteúdo dinamicamente.

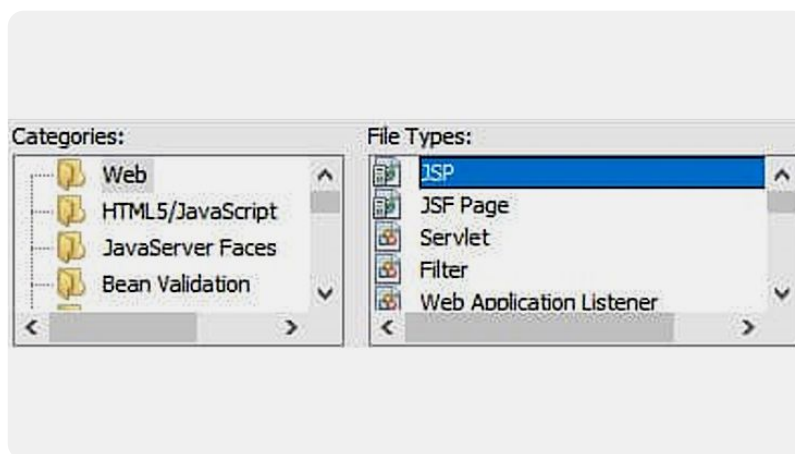
Neste vídeo, você verá a criação de uma página utilizando a tecnologia JSP. Para isso, será criado um arquivo de extensão ".jsp", que, ao ser requisitado, dará origem a um servlet, renderizando conteúdo em sintaxe java – os scriptlets



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Para criar uma página JSP, adicionamos um arquivo na categoria web. É necessário definir o nome do arquivo, sem acrescentar a extensão, e o novo componente ficará disponível na divisão **Web Pages** do projeto, ou em um subdiretório especificado. Confira!



"Wizard" de criação de arquivos do NetBeans.

Após o procedimento, vamos criar uma página JSP simples, utilizando o nome **ListaCores**. Ela não possui funcionalidade relevante, mas permitirá que analisemos a estrutura das páginas JSP. Veja!

java

```
<%@page contentType="text/html" pageEncoding="UTF-8" %>
```

```
<%  
String[] cores={"vermelho","verde","azul"};  
for(String x: cores) {  
    out.println("  
• "+x+"  
");  
}  
%>
```

Agora, vamos conhecer os elementos ou recursos utilizados no código anterior, escrito em HTML/Java Server Pages (JSP).

Diretivas

São utilizadas para importar bibliotecas e definir a página de erro, entre diversas outras opções. A primeira linha desse código é uma diretiva e indica o tipo de conteúdo adotado.

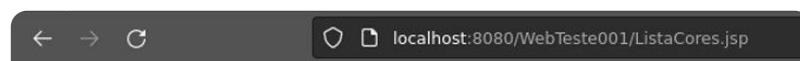
Código HTML

Define a estrutura básica da página e a inclusão de uma lista, com o uso do marcador ul. À exceção da primeira linha, todas as demais contêm código HTML.

Scriptlet

Trata-se do bloco delimitado por "<%" no início e "%>" no final, localizado internamente ao marcador de lista, que define um trecho de código de sintaxe Java executado no servidor. No trecho de código do exemplo, temos a definição de um vetor de texto com nomes de cores a serem impressas no conteúdo da página através do objeto out, implícito nas páginas JSP, como itens de lista.

Para testar nossa página JSP, vamos executar o projeto e efetuar a chamada correta a partir do navegador, como veremos no exemplo:



- vermelho
- verde
- azul

Resposta da requisição realizada ao ListaCores.jsp.

Os objetos **request** e **response** também são implícitos para as páginas JSP, podendo ser utilizados da mesma forma que nos métodos doGet e doPost dos Servlets. Inclusive, devemos nos lembrar de que a página JSP é

convertida em Servlet no primeiro acesso; logo, o que muda é a forma de programar, e não a funcionalidade original.

Atividade 3

A tecnologia de páginas JSP permite mesclar HTML e Java em meio ao mesmo código, constituindo uma opção particularmente interessante para a definição do design. Qual o nome utilizado para os trechos de código escritos em Java nas páginas?

A

Diretiva

B

Expression

C

Comentário

D

Scriptlet

E

Servlet



A alternativa D está correta.

Os trechos de código escritos em Java são chamados de Scriptlets, e os conteúdos são transcritos diretamente para o Servlet, gerado pelo container, diferentemente da parte HTML, que é transformada em instruções do tipo out.print.

Sessões e redirecionamentos

Sessões HTTP são de grande utilidade no ambiente web, provendo uma forma de manutenção de estados na troca de páginas, tendo em vista que o protocolo HTTP é considerado stateless.

Neste vídeo, você irá conferir as sessões HTTP para armazenar dados e estados entre requisições, e entenderá que é possível verificar se o usuário realizou o login ou recuperar atributos de usuário e de outros objetos armazenados em sessão. Utilizaremos o objeto "session" e você verá como ele pode persistir, inclusive, entre execuções do navegador.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Podemos controlar sessões de forma muito simples, com o uso da classe `HttpSession`, e um exemplo típico de utilização é no controle de login. Normalmente temos um `Servlet` para a verificação do login, e a sessão deve ser obtida a partir do objeto de requisição, invocando `getSession`.



Atenção

Nas páginas JSP, o controle de sessões é feito com o uso do objeto implícito `session`, da classe `HttpSession`.

Para demonstrar o processo de login, vamos criar uma página JSP, que receberá o nome "Segura.jsp" e representará um recurso com acesso autenticado. Vamos lá!

plain-text

```
<%@page contentType="text/html" pageEncoding="UTF-8" %>
<% if(session.getAttribute("usuario")==null) response.sendRedirect("Login.jsp"); else { %>
```

Esta é uma página protegida!

O usuário <%=session.getAttribute("usuario")%>
está logado.

```
<% } %>
```

Enquanto os parâmetros da requisição HTTP assumem apenas valores do tipo texto, os atributos da sessão permitem guardar qualquer tipo de objeto. Nesse caso, utilizaremos apenas um elemento de texto.

Na primeira parte do arquivo JSP, temos o teste para a existência do atributo "usuário" na sessão, e se ele não existir, isso significa que não há um usuário autenticado, devendo ocorrer o redirecionamento para a página de **login** através de **`sendRedirect`**.

Veja, a seguir, os dois métodos de redirecionamento do Java para web.

Método sendRedirect

Possui a interface `HttpServletResponse`. Envia um sinal de redirecionamento para o navegador, gerando uma nova requisição.

Método forward

Possui a interface `RequestDispatcher`. Envia um sinal de redirecionamento ao nível do servidor, que ocorre de forma interna, com a manutenção da requisição original e suas informações.

Ao analisar o código da página JSP, podemos observar que a instrução "else" é aberta antes do início do código HTML e é fechada apenas no final, logo após a tag de fechamento. Isso indica que a página de resposta será processada somente se o atributo "usuario" estiver presente na sessão. Durante a montagem da página, temos um retorno bastante simples. O login atual é exibido usando o método "getAttribute", e um botão é apresentado para realizar o logout, o qual é implementado através de uma chamada à `ServletLogin`.

Agora vamos criar a página de login com o nome "Login.jsp" e o Servlet responsável por controlar as ações relacionadas aos processos de conexão e desconexão, que será chamado de "ServletLogin". Vamos lá!

plain-text

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
```

Acesso ao Sistema

Login:
Senha:

```
<%  
    if(request.getAttribute("erro")!=null) {  
%>
```

```
Ocorreu um erro: <%=request.getAttribute("erro")%>  
<%  
    }  
%>
```

Na primeira parte de Login.jsp, temos um formulário HTML bastante simples, contendo as informações que deverão ser enviadas para `ServletLogin`, enquanto na segunda parte apresentamos mensagens de erro. Note que a segunda parte será apresentada apenas se o atributo de erro estiver presente na chamada ao JSP, e que não permite o envio por HTTP, apenas por meio do código do Servlet.


```

java
@WebServlet(name = "ServletLogin",
            urlPatterns = {"/ServletLogin"})
public class ServletLogin extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
        throws ServletException, IOException {
        String acao = request.getParameter("acao");
        if(acao==null)
            throw new ServletException("Parâmetro Requerido");
        HttpSession session = request.getSession();
        switch(acao){
            case "conectar":
                if(request.getParameter("login").equals("admin")&&
                   request.getParameter("senha").equals("123")){
                    session.setAttribute("usuario", "Administrador");
                    response.sendRedirect("Segura.jsp");
                } else {
                    request.setAttribute("erro", "Dados inválidos.");
                    RequestDispatcher rd =
                        request.getRequestDispatcher("Login.jsp");
                    rd.forward(request, response);
                }
                break;
            case "desconectar":
                session.invalidate();
                response.sendRedirect("index.html");
                break;
            default:
                throw new ServletException("Parâmetro incorreto");
        }
    }
}

```

Esse é um processo de login muito simples, mas que, por se tratar de uma autenticação de usuário, deverá adotar apenas o método doPost. O parâmetro “acao”, indicando a solicitação de conexão ou desconexão, é obrigatório, sendo gerada uma exceção caso ele não seja fornecido.

A partir de uma instrução switch, com base no parâmetro “acao”, temos a implementação da autenticação e da desconexão do sistema. Observe que o Java permite uso de texto para os desvios de fluxo do switch, mas apenas nas versões atuais da plataforma.

Para responder à ação "conectar" é feito um teste, em que apenas o login "admin" e a senha com valor "123" permitirão a autenticação. Fornecidos os valores corretos, temos o acréscimo do atributo “usuário” à sessão, contendo o valor "Administrador", e ocorre o redirecionamento para a página **Segura.jsp**.



Atenção

Caso sejam fornecidas credenciais diferentes das estipuladas, será definido o atributo erro para a requisição, com a mensagem "Dados Inválidos", e ocorrerá o retorno para Login.jsp, por meio de um redirecionamento interno.

Com a utilização do método invalidate, a sessão atual é fechada, e os atributos associados à conexão do usuário são removidos do servidor. Após a invalidação ocorre o redirecionamento para a página index.html.

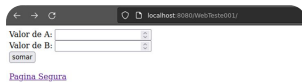
Agora basta adicionar uma chamada para a página Segura.jsp em index.html.

plain-text

[Pagina Segura](#)

As imagens seguintes apresentam o fluxo de execução do aplicativo, que normalmente ocorreria em uma tentativa de acesso à página segura. Confira!

Aplicativo contendo o link para a página segura



Resposta da requisição realizada ao selecionar o link "Página Segura"



A seguir, vemos o login bem-sucedido na página protegida.



Embora seja um processo de autenticação muito simples, com valores prefixados, você pode alterá-lo facilmente para utilizar uma base de dados e senhas criptografadas.

Atividade 4

É bastante comum a manipulação e redirecionamento de respostas e até requisições no servidor. Podemos encaminhar requisições de um cliente, dependendo da regra de negócio. Nesse sentido, qual o recurso utilizado para redirecionar internamente a requisição de um cliente?

A

getSession

B

sendRequest

C

RequestDispatcher

D

HttpSession

E

HttpObject



A alternativa C está correta.

Utilizamos o RequestDispatcher para redirecionar a requisição internamente, enquanto o método sendRedirect determina que o cliente realize uma nova requisição a outra página.

Prática

Neste ponto, temos algumas das principais tecnologias empregadas no desenvolvimento de Servlets: Containeres, JSP, Servlets e sessões. Assim, podemos criar um projeto de maneira a exercitar os conhecimentos obtidos até aqui.

Neste vídeo, você verá a criação de um aplicativo capaz de receber dados do cliente e armazená-los em sessão. Além disso, também faremos o redirecionamento de acordo com o login, aplicando lógica à distribuição de requisições internamente.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

Crie um aplicativo capaz de receber dados de um formulário, armazená-los em um ArrayList e exibir na mesma tela do formulário. Utilize sessions para armazenar a lista de itens. Inclua ainda o botão "limpar", para remover os dados da sessão. Para isso, execute os seguintes passos:

- Crie uma visualização para exibir a lista e o formulário.
- Implemente o Servlet para controlar as ações.

Como resolução dessa prática, teremos:

Arquivo "index.jsp"

```
<%@page import="java.util.ArrayList"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<title>Lista Items</title>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,
initial-scale=1.0">
</head>
<body>
<%
if (session.getAttribute("lista") != null) {
ArrayList<String> lista = (ArrayList<String>)
session.getAttribute("lista");
out.println("<ul>");
for (String item : lista) {
out.println("<li>");
out.println(item);
out.println("</li>");
}
out.println("</ul>");
}
else {
session.setAttribute("lista", new ArrayList());
}
%>
```

Arquivo "servlet.ServLista"

```
@WebServlet(name = "ServLista", urlPatterns = {"/ServLista"})
public class ServLista extends HttpServlet {
protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
request.setCharacterEncoding("UTF-8");

String accao = request.getParameter("acao");
ArrayList<String> lista;

switch (acao) {
case "adicionar":
lista = (ArrayList<String>)
request.getSession().getAttribute("lista");
String item = request.getParameter("item");
lista.add(item);
break;
case "limpar":
request.getSession().invalidate();
break;
default:
throw new AssertionError();
}

RequestDispatcher rd = request.getRequestDispatcher("index.jsp");
rd.forward(request,response);
}
// Código omitido
}
```

Atividade 5

Nem sempre o usuário possuirá o mesmo encoding da aplicação. Em clientes windows, por exemplo, é comum utilizar o encoding "WINDOWS-1252". Qual o mecanismo utilizado para garantir que o servidor "entenda" a requisição do cliente?

A

Request.getSession()

B

Request.getRequestDispatcher()

C

Response.setContentType()

D

Request.setCharacterEncoding()

E

Response.setContentType()



A alternativa E está correta.

Para que o servidor transforme a requisição para determinado "encoding", utilizamos o método `request.setCharacterEncoding`, passando como parâmetro o código do encoding desejado.

Middleware

Para definirmos middleware, devemos entender os conceitos de front-end e back-end. Mas antes, acompanhe o vídeo!

Neste vídeo, você irá conferir a utilização do JDBC como middleware para conexão com o banco de dados. Para isso, utilizaremos o banco de dados Derby como exemplo, criando tabelas, colunas e registros em um banco de dados com o NetBeans.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Vamos lá, então, aos conceitos necessários para definirmos middleware!

Front-end

É a camada de interfaces do sistema, com o uso de uma linguagem de programação. Aqui temos os aplicativos Java web como opção de front-end.

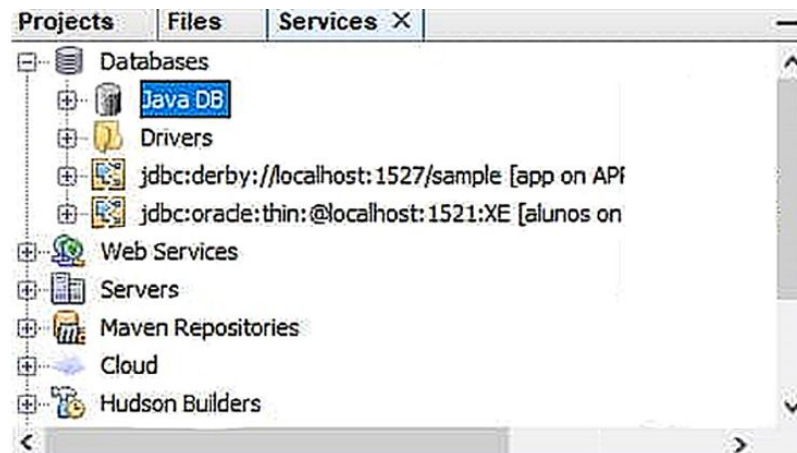
Back-end

É o conjunto de tecnologias que pode ser acessado a partir de nosso front-end, mas que não pertence ao mesmo ambiente, como bancos de dados e mensagerias. Adotaremos o **Derby** como back-end, com a consulta e manipulação de dados através de comandos SQL a partir do front-end.

Com diferentes componentes para acesso e modelos de programação heterogêneos, são muito comuns erros no desenvolvimento ou na integração desses elementos. Por isso, é necessária uma camada de software intermediária que fique responsável por promover a comunicação entre o front-end e o back-end. Chamada de **middleware**, essa camada possibilita uma integração transparente e permite fazer modificações nos componentes com pouca alteração de código.

O JDBC (Java Database Connectivity) é o middleware do ambiente Java para acesso a bancos de dados. Ele permite que utilizemos produtos de diversos fornecedores, sem modificações no código do aplicativo, desde que os bancos de dados aceitem o uso de SQL ANSI.

Entre as diversas opções de repositórios existentes, temos o **Derby**, ou **Java DB**, um banco de dados relacional construído totalmente com tecnologia Java, que não depende de um servidor e faz parte da distribuição padrão do JDK. Apache Derby é um subprojeto do Apache DB, disponível sob licença Apache, e que pode ser embutido em programas Java, bem como utilizado para transações on-line. Veja!



Aba "Services" do NetBeans.

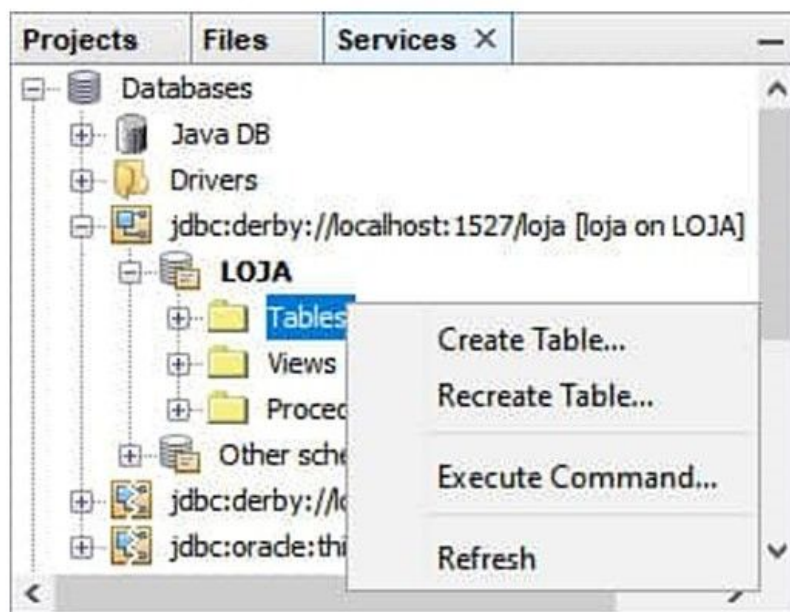
Podemos gerenciar nossos bancos de dados Derby de forma muito simples, por meio da aba Services do NetBeans, na divisão Databases.

Para criarmos um banco de dados, precisamos clicar com o botão direito sobre o driver "Java DB" contido na divisória "Databases", e escolher "Create Database", no menu de contexto. Na janela que será aberta, preenchemos o nome do novo banco de dados com o valor "loja", bem como usuário e senha, utilizando o valor "loja" para ambos. Confira!

"Wizard" para criação de database.

Sendo assim, ao clicar no botão de confirmação, o banco de dados será criado e ficará disponível para conexão, por meio do driver JDBC. A conexão é identificada por sua Connection String, tendo como base o endereço de rede (**localhost**), a porta padrão (**1527**) e a instância que será utilizada (**loja**).

Com isso, a conexão é aberta com o duplo clique sobre o identificador. Com o banco de dados aberto, podemos criar uma tabela, navegando até a divisão "Tables", no esquema LOJA, e utilizando o clique com o botão direito para acessar a opção "Create Table". Vejamos!



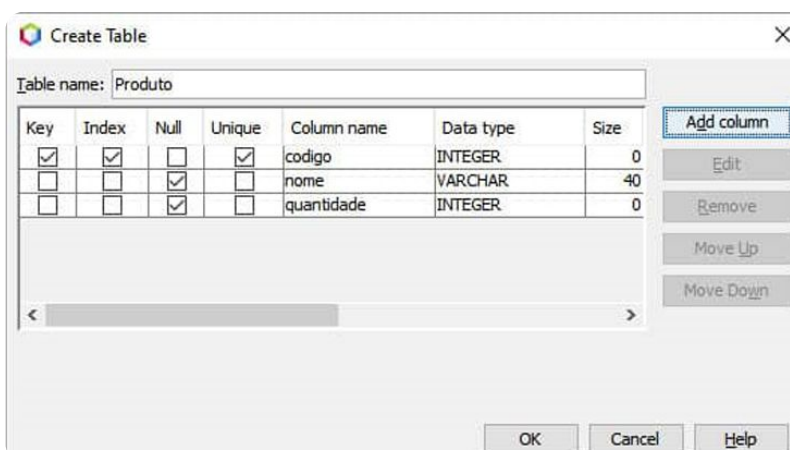
Navegador de serviços do NetBeans.

Após isso, na janela de criação, vamos configurar uma tabela de nome Produto, com os campos definidos de acordo com a tabela seguinte.

Campo	Tipo	Complemento
codigo	INTEGER	Chave primária
nome	VARCHAR	Tamanho: 40
quantidade	INTEGER	Sem atributos complementares

Tabela: Especificação dos campos da tabela “Produto”.
Adaptado por Marcos Alexandre Pinto de Castro Junior.

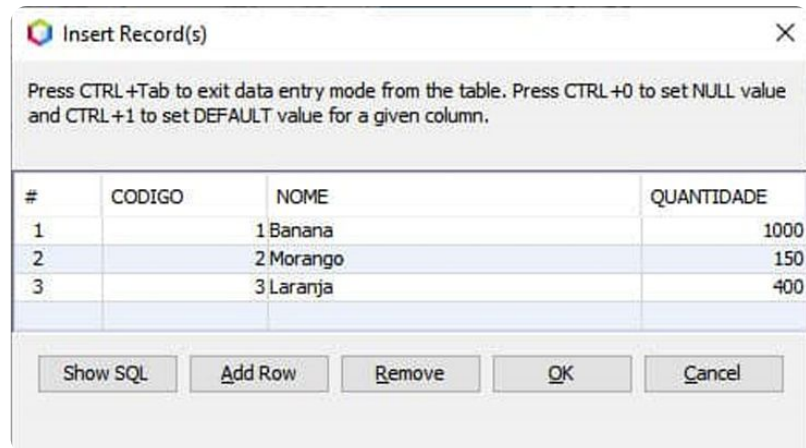
Definindo o nome da tabela e adicionando os campos, teremos a configuração que pode ser observada a seguir. Portanto, clique em “Add column” para criar as colunas.



"Wizard" para criação de tabela.

A tabela criada será acessada através de um novo nó, na árvore de navegação, abaixo de **Tables**, com o nome Produto. Utilizando o clique com o botão direito sobre a tabela e escolhendo a opção View Data, teremos uma área de visualização de dados, clique nesta área e pressione “ALT+I”

Nessa tela de inserção, podemos preencher os valores do novo registro, e se quisermos mais de um registro, basta clicar em "Add Row". Ao final do preenchimento dos dados, clicamos em OK e o NetBeans executará os comandos INSERT necessários.



#	CODIGO	NOME	QUANTIDADE
1		1 Banana	1000
2		2 Morango	150
3		3 Laranja	400

"Wizard" para inserção de dados na tabela.

Atividade 1

A respeito dos middlewares, assinale a alternativa correta.

A

Os middlewares são bancos de dados utilizados para persistir informações específicas.

B

Para se conectar a um banco de dados, utilizamos o middleware Java Derby.

C

Esses recursos permitem a separação e a adequada comunicação entre aplicações.

D

O SQL é o middleware padrão para conexão com o banco de dados Derby.

E

Middlewares são recursos internos da aplicação, não permitindo acesso externo.



A alternativa C está correta.

Para evitar o excessivo acoplamento entre aplicações e facilitar a comunicação, utilizamos middlewares. Dessa forma, podemos abstrair alguns aspectos da comunicação entre softwares.

Acesso ao banco de dados no Java

Com relação à codificação Java, os componentes do JDBC estão presentes no pacote **java.sql**. Sendo assim, conheça os quatro passos simples para o processo de utilização!

- Instanciar a classe do driver de conexão.
- Obter uma conexão (**Connection**) a partir da Connection String, usuário e senha.
- Instanciar um executor de SQL (**Statement**).
- Executar os comandos **DML** (linguagem de manipulação de dados).

Conectando a aplicação ao banco de dados

Neste vídeo, você verá os passos para estabelecer comunicação entre a aplicação e a camada de banco de dados, e entenderá a execução de instruções SQL, alterando e recuperando os registros.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Para o comando de seleção existe a recepção da consulta em um **ResultSet**, o que pode ser observado neste código:

```
java
public class NewClass {

    public static void main(String[] args) throws ClassNotFoundException, SQLException {
        ArrayList listaDescricoes = new ArrayList<>();

        // passo 1
        Class.forName("org.apache.derby.jdbc.ClientDriver");

        // passo 2
        Connection c1 = DriverManager.getConnection(
            "jdbc:derby://localhost:1527/loja",
            "loja", "loja");

        // passo 3
        Statement st = c1.createStatement();

        // passo 4 e recepção no ResultSet
        ResultSet r1 = st.executeQuery("SELECT * FROM PRODUTO");

        while (r1.next()) {
            listaDescricoes.add("Produto " + r1.getInt("codigo")
                + ": " + r1.getString("nome"));
        }

        System.out.println(listaDescricoes);
        r1.close();
        st.close();
        c1.close();
    }
}
```

No início do código, instanciamos o driver Derby a partir de uma chamada para o método **forName**. O driver instanciado fica disponível para o aplicativo, permitindo abrir conexões com o banco de dados através de JDBC.

Em seguida, é instanciada a conexão **c1**, por meio da chamada ao método **getConnection**, da classe **DriverManager**, e os parâmetros fornecidos são a Connection String, o usuário e a senha. A Connection String pode variar, sendo iniciada pelo driver utilizado, seguido dos parâmetros específicos para aquele driver, como servidor, porta e esquema.

Invocando **createStatement**, é gerado um executor de SQL de nome **st**. Com o executor instanciado, realizamos uma consulta ao banco, através da invocação do método **executeQuery**, recebendo um **ResultSet**.



Atenção

As consultas ao banco são feitas com a utilização de **executeQuery**, mas os comandos para manipulação de dados são executados por meio de **executeUpdate**.

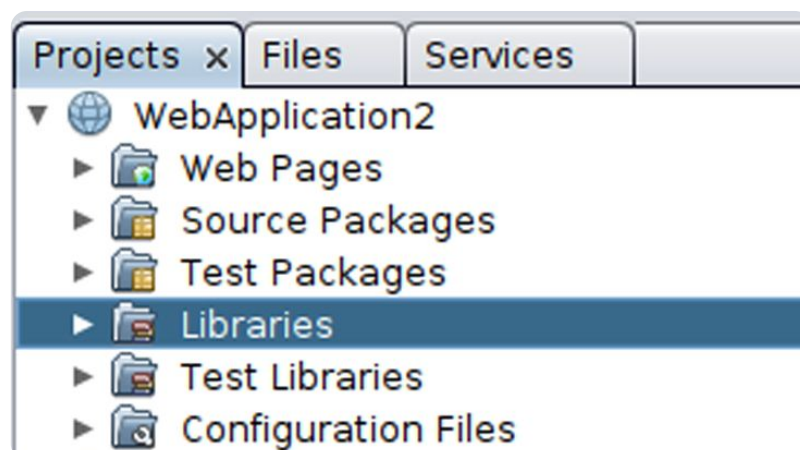
Após a recepção da consulta no objeto de nome **r1**, podemos nos movimentar pelos registros, com o uso de **next**, e acessar cada campo pelo nome para a obtenção do valor, sempre lembrando de utilizar o método correto para o tipo do campo, como **getString** para texto e **getInt** para valores numéricos inteiros.

Ao efetuar a consulta, o **ResultSet** fica posicionado antes do primeiro registro, na posição **BOF** (Beginning of File), e com o uso do comando **next** podemos mover para as posições seguintes, até atingir o final da consulta, na posição **EOF** (End of File). A cada registro visitado, adicionamos a descrição do produto ao **ArrayList**.

Na parte final, devemos fechar os componentes **JDBC** na ordem inversa daquela em que foram criados, já que existe dependência sucessiva entre eles.

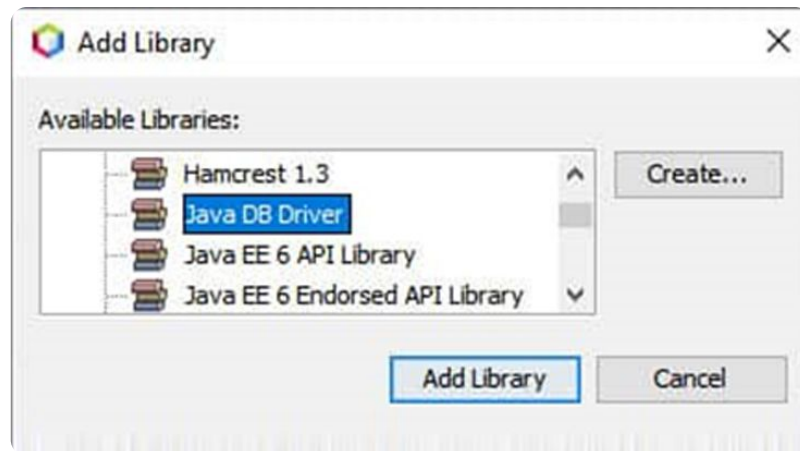
A biblioteca **JDBC** para o tipo de banco de dados escolhido, caso não esteja presente no servidor, deveremos adicioná-la ao projeto, ou ocorrerá erro durante a execução.

Sendo assim, podemos adicionar a biblioteca **Java DB Driver** clicando com o botão direito na divisão “**Libraries**” e usando a opção “**Add Library**”. Veja!



Navegador do projeto com a divisão **Libraries**.

Agora, acompanhe como é realizada a adição de bibliotecas a algum projeto.



"Wizard" para adição de bibliotecas ao projeto.

Atividade 2

Grande parte das aplicações Java – sejam para a web ou não – possui necessidade de persistir dados entre execuções. Normalmente, esses aplicativos utilizam algum recurso de banco de dados. Marque a alternativa que indica a interface utilizada para conectar a bancos de dados utilizando linguagem Java:

A

IDE

B

JDBC

C

JNDI

D

IRCC

E

SQLJ



A alternativa B está correta.

O JDBC (Java Database Connectivity) é a interface padrão para conexão com aplicações de bancos de dados utilizando linguagem Java.

Data Access Object (DAO)

Agora que sabemos efetuar as operações sobre o banco de dados, precisamos organizar a forma de programar, evitando a mistura de código Java com comandos SQL, o que dificulta a manutenção de sistemas.

Neste vídeo, você verá a importância da camada DAO na construção de aplicações que se comunicam com o banco de dados, tendo em vista a necessidade da reutilização de código e da concentração de comandos SQL em apenas um lugar.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

A linguagem Java é orientada a objetos, o que facilita a representação de tabelas como classes e registros como instâncias dessas classes. Vamos conferi-las!

```
java

public class Produto {
    public int codigo;
    public String nome;
    public int quantidade;

    public Produto(){ }

    public Produto(int codigo, String nome, int quantidade) {
        this.codigo = codigo;
        this.nome = nome;
        this.quantidade = quantidade;
    }
}
```

Para evitar o SQL espalhado ao longo do código, utilizamos o padrão DAO (Data Access Object), com o objetivo de concentrar as instruções SQL em um único tipo de classe, permitindo o agrupamento e a reutilização dos diversos comandos relacionados ao banco de dados. Normalmente, temos uma classe DAO para cada classe de entidade relevante para o sistema. Acompanhe no código a seguir:

```

java

public class ProdutoDAO {
    private Connection getConnection() throws Exception{
        Class.forName("org.apache.derby.jdbc.ClientDriver");
        return DriverManager.getConnection(
            "jdbc:derby://localhost:1527/loja",
            "loja", "loja");
    }
    private Statement getStatement() throws Exception{
        return getConnection().createStatement();
    }
    private void closeStatement(Statement st) throws Exception{
        st.getConnection().close();
    }

    public List obterTodos(){
        ArrayList lista = new ArrayList<>();
        try {
            ResultSet r1 = getStatement().executeQuery(
                "SELECT * FROM PRODUTO");
            while(r1.next())
                lista.add(new Produto(r1.getInt("codigo"),
                    r1.getString("nome"),r1.getInt("quantidade")));
            closeStatement(r1.getStatement());
        }catch(Exception e){
        }
        return lista;
    }
}

```

Inicialmente, criamos os métodos **getStatement** e **closeStatement** com o objetivo de gerar executores de SQL e eliminá-los, efetuando também as conexões e desconexões nos momentos necessários. Outro método utilitário é o **getConnection** para encapsular o processo de conexão com o banco.



Atenção

O método **obterTodos** irá retornar todos os registros da tabela **Produto** no formato de um **ArrayList** de entidades do tipo **Produto**. Primeiramente, é executado o SQL necessário para a consulta e, para cada registro obtido no cursor, é gerado um novo objeto da classe **Produto** e adicionado à lista de retorno.

Podemos alterar o código de **ListaCores.jsp** para exibir o nome de cada produto. Confira!

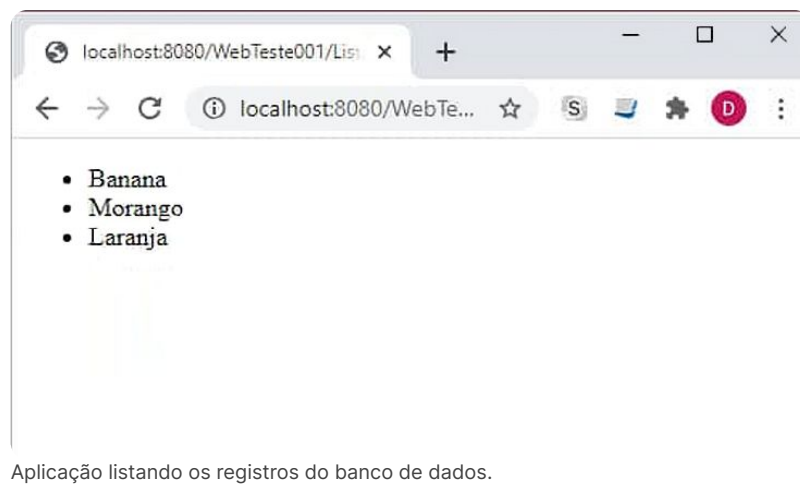
javascript

```
<%@page import="br.com.minhaloja.classes.Produto"%>
<%@page import="br.com.minhaloja.dao.ProdutoDAO"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
```

```

    <%
        ProdutoDAO dao = new ProdutoDAO();
        for (Produto p : dao.obterTodos()) {
            out.println("
    • " + p.nome + "
    ");
        }
    %>
```

Após, importaremos as classes Produto e ProdutoDAO, com a utilização da diretiva page, viabilizando seu uso. A partir daí, instanciamos o DAO, chamamos obterTodos e percorremos a coleção, acrescentando os produtos à lista. Veja!



Agora que testamos o acesso ao banco de dados no ambiente web, vamos acrescentar os métodos para efetuar a **inclusão** e a **exclusão** de produtos na classe **ProdutoDAO**.

java

```
public void excluir(int codigo){
    try {
        Statement st = getStatement();
        st.executeUpdate(
            "DELETE FROM PRODUTO WHERE CODIGO = "+
            codigo);
        closeStatement(st);
    }catch(Exception e){
    }
}
```

Com isso, podemos observar que a exclusão utiliza um **Statement**, apenas com a observação de que em vez de utilizar **executeQuery**, como nas seleções de dados, adotamos o método **executeUpdate**, por se tratar de

um comando DML. Com base no código fornecido, um comando SQL é montado, visando à exclusão do produto identificado pelo código.

```
java

public void incluir(Produto p){
    try {
        PreparedStatement ps = getConnection().prepareStatement(INSERT INTO PRODUTO
VALUES(?,?,?));
        ps.setInt    (1, p.codigo);
        ps.setString(2, p.nome);
        ps.setInt    (3, p.quantidade);
        ps.executeUpdate();
        closeStatement(ps);
    }catch(Exception e){
    }
}
```

O elemento do tipo **PreparedStatement** permite a construção de comandos SQL parametrizados. O uso de parâmetros facilita a escrita do comando SQL, sem a preocupação com o uso de apóstrofe ou outro delimitador, sendo particularmente útil quando tivermos de trabalhar com datas.



Dica

Ao definirmos os parâmetros, utilizamos pontos de interrogação que assumem valores posicionais, começando a partir de um. Isso difere um pouco da indexação dos vetores, que começa em zero.

Sendo assim, os parâmetros são preenchidos com métodos para cada tipo, como **setInt** para inteiro e **setString** para texto. Após o preenchimento, devemos executar o comando SQL com `executeUpdate` no caso das instruções INSERT, UPDATE e DELETE, ou `executeQuery` para a instrução SELECT.

Com as classes Produto e ProdutoDAO completas, podemos implementar a interface web com o uso de **Servlets** e **JSPs**.

Atividade 3

Quando criamos uma classe DAO, é comum a necessidade de gerar os comandos SQL de forma dinâmica, levando em consideração os atributos da classe. Para evitar conversões e concatenações no SQL, podemos utilizar uma classe específica:

A

PreparedStatement

B

ResultSet

C

Statement

D

Connection

E

PreparedStatement



A alternativa A está correta.

A classe PreparedStatement permite definir comandos SQL parametrizados, sendo os parâmetros definidos de forma posicional, por meio de interrogações, e os valores preenchidos com base em métodos como setInt e setString, evitando conversões manuais ou concatenações.

Aplicativos web com uso de DAO

A construção de nosso aplicativo irá se concentrar na melhor utilização de Servlets e JSPs, ou seja, teremos o Servlet como um conversor de formatos e redirecionador de fluxo, enquanto o JSP será responsável pela construção da interface com o usuário.

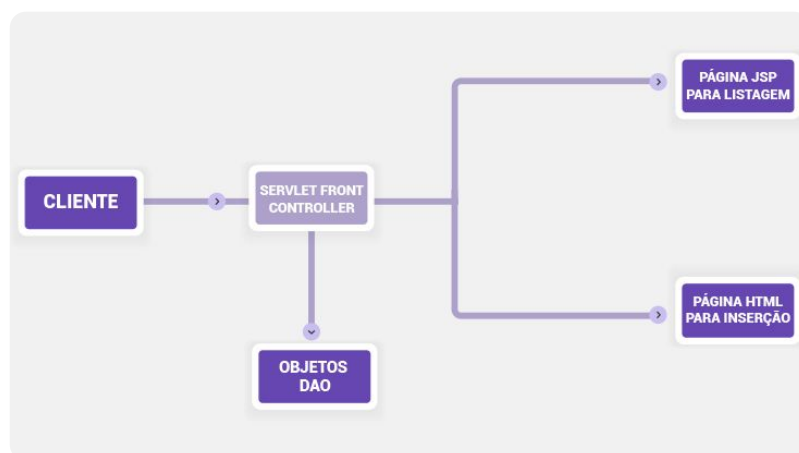
Neste vídeo, você verá uma demonstração de como criar uma aplicação simples para inserção e listagem de registros. Apesar de simples, essa aplicação fará você entender e empregar a separação de responsabilidades entre diferentes componentes de aplicação. É uma prática bem utilizada no mercado devido às vantagens no desenvolvimento e na manutenção.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

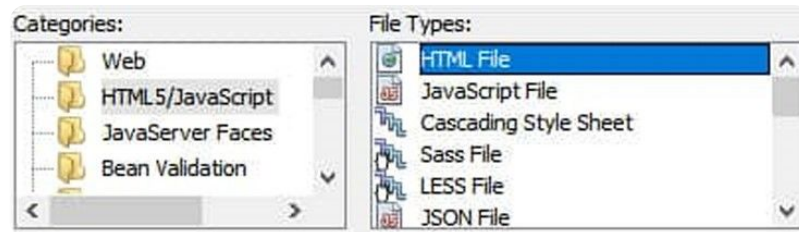
Nosso sistema seguirá o padrão de desenvolvimento **front controller**. Vamos conferi-lo!



Implementação do front controller.

Seguindo o padrão front controller, todas as requisições serão recebidas em apenas um Servlet, onde ocorrerão as conversões e os processamentos necessários, com o redirecionamento para a visualização correta ao final.

Tendo como base o comportamento descrito, fica clara a necessidade de um parâmetro que permita diferenciar qual fluxo deverá ser executado, ao qual daremos o nome de “acao”. Vamos começar pela parte mais simples, acrescentando um arquivo do tipo HTML File, na categoria HTML5/JavaScript, com o nome **ProdutoDados**. Veja!



"Wizard" de criação de arquivos do NetBeans.

Nossa página HTML será constituída apenas de um formulário, com campos de entrada para os atributos de Produto, além de um campo escondido (hidden) para definir o valor de “acao” como "inserirX". Após o usuário preencher os dados e clicar em “Incluir”, ocorrerá o envio pelo método **post** para o Servlet.

plain-text

Codigo:

Nome:

Quantidade:

Em seguida, vamos adicionar um arquivo do tipo JSP, na categoria web, com o nome ProdutoLista. A nova página será responsável pela exibição dos dados emitidos a partir do Servlet, além de apresentar links para inclusão e exclusão de produtos.

plain-text

```
<%@page import="br.com.minhaloja.classes.Produto" %>
<%@page import="java.util.ArrayList" %>
<%@page contentType="text/html" pageEncoding="UTF-8" %>
```

Incluir Produto

```
<% ArrayList lista = (ArrayList) request.getAttribute("listaProduto");
    for (Produto p : lista) {
%>

<% }%>
```

codigo	nome	quantidade	Opções
<%=p.codigo%>	<%=p.nome%>	<%=p.quantidade%>	<div>—</div> <div>excluir</div>

Na parte inicial da página, temos um link para a inclusão de produto, com a chamada para o Servlet, tendo o parâmetro "acao" com valor "incluir". No Servlet, o acesso corresponderá a um redirecionamento direto para a página HTML de cadastro.

Em seguida, temos a criação de uma tabela para a exibição dos dados de cada produto, a partir de uma lista fornecida em um atributo da requisição. A tabela apresenta uma linha inicial com os nomes dos campos, e as demais linhas criadas dinamicamente pelo **Scriptlet**.



Atenção

Como os atributos podem ser de qualquer tipo de objeto, é necessário efetuar o type cast ao recebê-los, via método `getAttribute`.

Após resgatar a lista, executamos um loop do tipo **for each** para visitar os produtos, executando blocos de repetição. Note que o bloco, delimitado com o uso de chaves, contém elementos HTML e Scriptlets.

Dentro do bloco, construímos uma linha da tabela HTML, com colunas preenchidas por valores dos campos. A etiqueta de substituição "`<%= ... %>`" equivale a um **Scriptlet** com o comando **out.print**.

plain-text

```
<!-- Formas Equivalentes --%>
<%= p.codigo %>
<% out.print(p.codigo); %>
```

Observamos a combinação fluida entre HTML e etiquetas de substituição, principalmente na montagem do link, em que cada linha terá um código diferente, fazendo referência ao produto corrente. O link de exclusão utiliza o parâmetro “acao” com valor “excluirX”, enquanto o parâmetro “codigo” recebe o valor dinamicamente.

plain-text

```
<!-- Formas Equivalentes --%>
<%= p.codigo %>
<% out.print(p.codigo); %>
```

Agora chegou o momento de criar o **Servlet**, orquestrando a funcionalidade de nossas páginas e executando os processos necessários para cada chamada. Vamos adicionar um Servlet com o nome **ServletProduto**. Veja!

```

java

@WebServlet(name = "ServletProduto", urlPatterns = {"/ServletProduto"})
public class ServletProduto extends HttpServlet {

    protected void processRequest(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException{
        String acao = request.getParameter("acao");
        if (acao == null) {
            acao = "listar";
        }
        String paginaDestino = (acao.equals("incluir"))
            ? "ProdutoDados.html" : "ProdutoLista.jsp";
        ProdutoDAO dao = new ProdutoDAO();
        switch (acao) {
            case "inserirX":
                System.out.println("Até aqui foi!!!");
                int codigo = Integer.parseInt(request.getParameter("codigo"));
                String nome = request.getParameter("nome");
                int quantidade = Integer.parseInt(
                    request.getParameter("quantidade"));
                {
                    try {
                        dao.incluir(new Produto(codigo, nome, quantidade));
                    } catch (Exception ex) {
                        Logger.getLogger(ServletProduto.class.getName()).log(Level.SEVERE,
null, ex);
                    }
                }
                break;

            case "excluirX":
                dao.excluir(Integer.parseInt(request.getParameter("codigo")));
                break;
        }
        if (!acao.equals("incluir")) {
            request.setAttribute("listaProduto", dao.obterTodos());
        }
        request.getRequestDispatcher(paginaDestino).
            forward(request, response);
    }
}

```

O primeiro passo, na execução do Servlet, é a captura do parâmetro “acao”, e se ele não é fornecido, assumimos o valor “listar”. Com base nesse parâmetro, selecionamos a página de destino, que será ProdutoDados.html para inclusão e ProdutoLista.jsp para as demais ações. Em seguida, é instanciado um objeto do tipo ProdutoDAO para suportar as ações.

Para “acao” com o valor “inserirX”, capturamos os valores fornecidos via HTTP, efetuando a conversão para inteiro, quando necessário, e invocamos o método incluir, passando uma instância de Produto, inicializada com os valores obtidos. Para o valor “excluirX”, resgatamos o valor do parâmetro “codigo”, convertendo para inteiro, e efetuamos uma chamada para o método excluir.

Atividade 4

Um padrão de desenvolvimento muito utilizado na construção de sistemas web é a concentração das chamadas em um ponto único, onde são feitas as conversões necessárias, a execução de métodos de negócios e o redirecionamento de fluxo. Qual o nome desse padrão de desenvolvimento?

A

Session Facade

B

DAO

C

Flyweight

D

Front controller

E

MVC



A alternativa D está correta.

Utilizamos o padrão front controller para centralizar chamadas efetuadas para o sistema, sendo frequente a utilização de Servlets na implementação.

Adicionando uma nova entidade de negócio

O fluxo de trabalho com um framework bem sedimentado é bastante natural. Dessa forma, ao utilizarmos o padrão front controller em conjunto com os recursos de DAO, ganhamos eficiência no trabalho. É importantíssimo que o desenvolvedor esteja familiarizado com a sequência de etapas necessárias à criação de funcionalidades do tipo CRUD.

A criação dos chamados crud é uma das tarefas mais comuns no mundo do desenvolvimento web. Portanto, implementaremos uma nova aplicação para cadastro de animais utilizando os principais componentes encontrados em aplicações web das mais diversas.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Roteiro de prática

Agora que já conhecemos os principais componentes de uma aplicação, que tal criarmos um aplicativo para armazenar um cadastro de pessoas? Crie uma aplicação capaz de incluir, listar, excluir e exibir dados cadastrais de pessoas, contendo código, nome, endereço e e-mail para cada contato. Para tal, siga o passo a passo a seguir.

- Crie um projeto chamado "cadastro".
- Crie os pacotes para classes DAO, modelos e Servlet.
- Insira os dados iniciais em uma tabela em um banco Derby.

- Defina as classes de modelo.
- Implemente o DAO.
- Desenvolva o Servlet.
- Adicione a camada de visualização.

O resultado dessa prática será:

Comando para criar a tabela com dados iniciais

```
CREATE TABLE PESSOA (
  CODIGO INTEGER NOT NULL, primary key,
  NOME VARCHAR(50) NOT NULL,
  ENDEREÇO VARCHAR(100) NOT NULL,
  EMAIL VARCHAR(50) NOT NULL
);
INSERT INTO PESSOA (CODIGO, NOME, ENDEREÇO, EMAIL)
VALUES (1, 'Johnny B. Goode', 'Deep down in Louisiana, close to Orleans',
'johnny.goode@playguitar.com');
```

Classe "br.com.cadastro.modelo.Pessoa"

```
package br.com.cadastro.modelo;

public class Pessoa {

    public int codigo;
    public String nome;
    public String endereco;
    public String email;

    public Pessoa() {
    }

    public Pessoa(int codigo, String nome, String endereco, String email) {
        this.codigo = codigo;
        this.nome = nome;
        this.endereco = endereco;
        this.email = email;
    }
}
```

Classe "br.com.cadastro.dao.PessoaDAO"

```
package br.com.cadastro.dao;

import br.com.cadastro.modelo.Pessoa;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

public class PessoaDAO {

    private static Connection getConnection() throws Exception {
        Class.forName("org.apache.derby.jdbc.ClientDriver");
        return DriverManager.getConnection(
            "jdbc:derby://localhost:1527/cadastro",
            "cadastro", "cadastro");
    }

    private Statement getStatement() throws Exception {
        return getConnection().createStatement();
    }

    private void closeStatement(Statement st) throws Exception {
        st.getConnection().close();
    }

    public List<Pessoa> obterTodos() {
        ArrayList<Pessoa> lista = new ArrayList<>();
        try {
            ResultSet r1 = getStatement().executeQuery(
                "SELECT * FROM PESSOA");
            while (r1.next()) {
                lista.add(new Pessoa(r1.getInt("codigo"),
                    r1.getString("nome"),
                    r1.getString("endereco"),
                    r1.getString("email")
                ));
            }
            closeStatement(r1.getStatement());
        } catch (Exception e) {
        }
        return lista;
    }

    public void excluir(int codigo) {
        try {
            Statement st = getStatement();
            st.executeUpdate(
                "DELETE FROM PESSOA WHERE CODIGO = "
                + codigo);
            closeStatement(st);
        } catch (Exception e) {
        }
    }

    public void incluir(Pessoa p) throws Exception {
        PreparedStatement ps = getConnection().prepareStatement(
            "INSERT INTO PESSOA VALUES(?, ?, ?, ?)");

        ps.setInt(1, p.codigo);
        ps.setString(2, p.nome);
        ps.setString(3, p.endereco);
        ps.setString(4, p.email);
        ps.executeUpdate();
        closeStatement(ps);
    }
}
```

Classe "br.com.cadastro.servlets"

```
package br.com.cadastro.servlets;

import br.com.cadastro.modelo.Pessoa;
import br.com.cadastro.dao.PessoaDAO;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 *
 * @author imale
 */
@WebServlet(name = "ServletPessoa", urlPatterns = {"/ServletPessoa"})
public class ServletPessoa extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        String acao = request.getParameter("acao");
        if (acao == null) {
            acao = "listar";
        }
        String paginaDestino = (acao.equals("incluir"))
            ? "PessoaDados.html" : "PessoaLista.jsp";
        PessoaDAO dao = new PessoaDAO();
        switch (acao) {
            case "incluirX":
                int codigo = Integer.parseInt(request.getParameter("codigo"));
                String nome = request.getParameter("nome");
                String endereco = request.getParameter("endereco");
                String email = request.getParameter("email");
                {
                    try {
                        dao.incluir(new Pessoa(codigo, nome, endereco, email));
                    } catch (Exception ex) {
                        Logger.getLogger(ServletPessoa.class.getName()).log(Level.SEVERE, null, ex);
                    }
                }
                break;
            case "excluirX":
                dao.excluir(Integer.parseInt(request.getParameter("codigo")));
                break;
        }
        if (acao.equals("incluir")) {
            request.setAttribute("listaPessoa", dao.obterTodos());
        }
        request.getRequestDispatcher(paginaDestino).forward(request, response);
    }

    // O restante do código foi omitido
}
```

Página "PessoaLista.jsp"

```
<%@page import="br.com.cadastro.modelo.Pessoa" %>
<%@page import="java.util.ArrayList" %>
<%@page contentType="text/html" pageEncoding="UTF-8" %>
<DOCTYPE html>
<html>

<body>
    <a href="ServletPessoa?acao=incluir" >Incluir Pessoa</a>
    <table border="1" width="80%">
        <tr>
            <td>
                <td-codigo>/td>
                <td-nome>/td>
                <td-endereco>/td>
                <td-email>/td>
            </td>
        </tr>
        <tr>
            <td colspan="4">
                <% ArrayList<Pessoa> lista = (ArrayList<Pessoa>)
                request.getAttribute("listaPessoa");
                for (Pessoa p : lista) {
                    %>
                    <tr>
                        <td>
                            <%=p.codigo%>
                        </td>
                        <td>
                            <%=p.nome%>
                        </td>
                        <td>
                            <%=p.endereco%>
                        </td>
                        <td>
                            <%=p.email%>
                        </td>
                        <td>
                            <a href="ServletPessoa?acao=excluirX&codigo=%<%=p.codigo%>%>"
                                >excluir</a>
                        </td>
                    </tr>
                </td>
            </table>
</body>
```

Página "PessoaDados.html"

```
<html>
<body>
    <form action="ServletPessoa" method="post">
        Código: <input type="number" name="codigo" /> <br />
        Nome: <input type="text" name="nome" /> <br />
        Endereço: <input type="text" name="endereco" /> <br />
        E-mail: <input type="text" name="email" /> <br />
        <input type="hidden" name="acao" value="incluirX" />
        <input type="submit" value="Incluir" />
    </form>
</body>
</html>
```

Atividade 5

Qual recurso do código é responsável por redirecionar requisições para diferentes páginas com base em parâmetros recebidos?

A

O método `processRequest(HttpServletRequest request, HttpServletResponse response)`.

B

A declaração `protected void processRequest(HttpServletRequest request, HttpServletResponse response)` throws `ServletException`, `IOException`.

C

O switch-case que verifica o valor da variável "acao".

D

A inclusão do arquivo "PessoaDados.html" ou "PessoaLista.jsp" como página de destino.

E

A chamada do método `request.getRequestDispatcher(paginaDestino).forward(request, response)`.



A alternativa E está correta.

O código realiza o redirecionamento das requisições para diferentes páginas com base na variável `paginaDestino`. A chamada do método `request.getRequestDispatcher(paginaDestino).forward(request, response)` é responsável por encaminhar a requisição para a página de destino, que pode ser "PessoaDados.html" ou "PessoaLista.jsp", dependendo do valor da variável "acao".

Considerações finais

- Utilização de web Servers.
- Emprego de Application Servers.
- Diferença entre Application Servers e Web Servers.
- Tecnologias Servlet e JSP.
- Manipulação de sessões para persistir dados entre requisições.
- Criação de uma aplicação web capaz de persistir dados em banco.

Explore +

Confira as indicações de leitura que separamos especialmente para você!

- **Introdução ao Desenvolvimento de Aplicações Web**, disponível na página Apache NetBeans.
- **JDBC e Java – Programação para Banco de Dados**, de George Reese, publicado pela editora O'Reilly, 2001.

Referências

CASSATI, J. P. **Programação servidor em sistemas web**. Rio de Janeiro: Estácio, 2016.

CORNELL, G.; HORSTMANN, C. **Core Java**. 8. ed. São Paulo: Pearson, 2010.

DEITEL, P.; DEITEL, H. **AJAX, Rich Internet Applications e Desenvolvimento Web para Programadores**. São Paulo: Pearson Education, 2009.

DEITEL, P.; DEITEL, H. **JAVA - Como Programar**. 8. ed. São Paulo: Pearson, 2010.

FONSECA, E. **Desenvolvimento de Software**. Rio de Janeiro: Estácio, 2015.

MONSON-HAEFEL, R.; BURKE, B. **Enterprise Java Beans 3.0**. 5. ed. USA: O'Reilly, 2006.

MUNIZ, A. *et al.* **Jornada Java**. Rio de Janeiro: Brasport, 2021.

SANTOS, F. **Programação I**. Rio de Janeiro: Estácio, 2017.