Servlets

Neste curso aprenderemos o que o Tomcat realiza com Servlet, e como fazer o mapeamento do protocolo HTTP para as chamadas, requisições e como devolver respostas.

Discutiremos sobre a organização de uma aplicação: como mudar a chamada para um modelo, e como criar visualizações HTML no navegador. Falaremos ainda sobre JSP, Expression Language, Tag Libs, e o modelo por trás dos despachadores de requisições e redirecionamento.

Tomcat

Apache Tomcat ou apenas Tomcat é um servidor web em Java

De longe o servidor mais famoso no mundo Java e que foi criado pela Apache, que é uma organização *open source*.

programando em Java, queremos trabalhar na web e, no que diz respeito ao Tomcat, o mais importante é que ele utiliza o protocolo HTTP e HTML.

O Tomcat é puramente Java enquanto Apache HTTP é escrito em C. Além disso, o Apache HTTP é um servidor estático (por padrão pelo menos) e precisa de alguma integração com outra linguagem para se gerar HTML dinamicamente. O Tomcat já é dinâmico de cara, usando Java e JSP, claro!

Tomcat é um servidor web sabe trabalhar com o protocolo HTTP, por padão roda na porta 8080, ele sabe gerar uma página html

Dynamic Web Project

Quando não estamos lidando apenas com um simples projeto Java, e precisamos trabalhar com arquivos web, HTML e talvez alguma imagem. Exatamente por essa complexidade, escolhemos a opção "Dynamic Web Project".

Servlet

**Servlet é um objeto especial armazenado dentro do projeto**, e sua particularidade consiste na possibilidade de o chamarmos via protocolo HTTP.

Um conteúdo dinâmico. Ao receber uma requisição podemos configurar o envio de um e-mail, cadastrar alguma informação no banco de dados ou ler dados do banco e gerar um HTML dinamicamente, por exemplo.

Todas essas possibilidades não são possíveis dentro de uma página HTML, já que se trata apenas de um arquivo de texto. Portanto, precisaremos programar, ou seja, escrever uma classe, criar objetos e executar lógicas no banco dados. Essa é a tarefa do Servlet.

Quando o Tomcat recebe a requisição do navegador com relação aos dados do projeto **gerenciador**, ao abrirmos a página não estamos mais lidando com um arquivo, mas com um Servlet. Isto é, um objeto especial executado para gerar uma resposta HTTP dinâmica.

Anotações:

* Apache Tomcat ou apenas Tomcat é um servidor web em Java
* Tomcat *entende* o protocolo HTTP e roda por padrão no http://localhost:8080
* O projeto Java faz parte da URL, no nosso caso: http://localhost:8080/gerenciador
* Uma aplicação web Java pode ter páginas HTML
* Uma servlet é um objeto Java que podemos chamar a partir de uma requisição HTTP
* Para mapear a URL para uma servlet usamos a anotação @WebServlet
* Uma servlet deve estender a classe HttpServlet e sobrescrever um determinado método (por exemplo service)

Métodos Servlet

service – método que aceita tanto um request quanto um response de POST ou GET

doPost – método que aceita apenas um POST

doGet – método que aceita apenas um GET

Anotações:

Nessa aula aprendemos como:

* escrever uma servlet que atende apenas GET ou POST
* ler parâmetros da requisição dentro da servlet
* criar um formulário HTML usando as tags form e input
* enviando os dados pelo formulário através do POST
* diferença entre GET e POST

JSP

JSP é uma **página dinâmica Java** ou Java Server Page.

.jsp = .html

O JSP é uma tecnologia relacionada ao Servlet que transformará a página HTML em algo dinâmico, que permite algumas ações de programação.

Porém, neste novo formato podemos inserir códigos Java, desde que eles sejam marcados por <% %>. Chamamos essa inserção de código Java um arquivo jsp de scriptlet.

JSP é uma tecnologia que renderiza as páginas no servidor antes de enviá-la. Todo o código (scriptlet) é executado no servidor antes de enviar a resposta para o navegador.

Exemplo sintaxe:

<%

String nome = "Nico";

%>

<p><%= nome %></p>

A sintaxe <%= nome %> é um atalho para não usar out.println(nome);

Conexão entre o Servlet e o JSP

Para executarmos esse processo usaremos a request, que possui um objeto específico para despachar a requisição para o JSP, trata-se do getRequestDispatcher(). Precisamos indicar ao despachador o destino da requisição, isto é, o arquivo /novaEmpresaCriada.jsp. Nos será devolvido um RequestDispatcher, portanto pegaremos a referência rd e realizaremos a importação.

Resta adicionarmos um item que ativa o percurso dessa requisição, o método forward() que receberá os parâmetros de request e response.

No Servlet, agruparemos o nome da empresa em request. Para isso utilizaremos setAttribute("empresa", empresa.getNome()), contudo perceba que assinatura do método setAttribute() recebe um string name e um Object. A string name funciona como apelido, que será empresa e em seguida o objeto empresa.

O apelido empresa será usado no arquivo JSP. Usaremos a request e o método getAttribute() que receberá empresa. O método devolverá um objeto, e sabemos que se trata de uma string, indicando uma referência mais específica.

Anotações:

Nessa aula começamos a separar as responsabilidades entre servlet e JSP e aprendemos:

* JSP significa Java Server Pages
* JSP é uma página automaticamente processada pelo Tomcat
* Para gerar HTML dinamicamente no JSP usamos Scriptlets
* Um scriptlet <% %> é um código Java dentro do HTML
* Um scriptlet só funciona em uma página JSP
* Usamos o RequestDispatcher para chamar um JSP a partir da servlet
* Obtemos o RequestDispatcher a partir do HttpServletRequest
* Usamos a requisição para colocar ou pegar um atributo (setAttribute(.., ..) ou getAttribute(..))

 Expression language ${ }

Essa linguagem possui algumas outras funções, mas seu cerne é interpretar expressões e imprimir o resultado dessa interpretação na tela

Uma forma de imprimir a expressão utilizando menos scriptlets é usar uma outra linguagem bastante simples - mais precisamente de uma sintaxe, marcada por ${ }, e utilizada para definir expressões. Uma expressão escrita entre as chaves { } é interpretada pelo navegador e impressa na tela.

simplificamos nosso código utilizando essa linguagem. Contudo, a *expression language* não realiza laços, portanto precisamos aprender outras ferramentas para substituir de fato todo o scriptlet do código.

<html>

<body>

Empresa ${ empresa } cadastrada com sucesso!

</body>

</html>

JSTL

JSTL significa *Java Standard Tag Library*

É a biblioteca padrão de tags do Java, mas existem outras bibliotecas. Por exemplo, o Spring MVC possui a sua própria.

Precisamos adicionar uma lib específica para que possamos utilizá-la.

Alternativa correta! JSTL não vem embutido com o Tomcat.

Para marcar nosso laço, declarando seu começo e fim por meio de tags.

Para realizamos essa ação, precisaremos de uma biblioteca, afinal, por padrão, o Tomcat não possui essa capacidade

Usaremos a tag <forEach> para realizar o laço. Contudo, o Eclipse ainda não saberá que este <forEach> é da nossa nova biblioteca anexada.

Precisaremos importar a nova biblioteca. Normalmente as importações são agrupadas no começo da página, e seguiremos esse padrão. A declaração de importação é parecida com a de uma classe ou pacote, usamos as marcações <%@ %> e a expressão taglib, que está relacionada a uma biblioteca de tags.

O JSP é, portanto, uma página HTML com JSTL e expression language.

Tags sub-biblioteca JSLT:

**JSTL (Java Standard Tag Library)**

**core** - controle de fluxo

**fmt** - formatação / i18n (internacionalização)

**sql** - executar SQL

**xml** - gerar XML

o *core*, que realiza o controle de fluxo, mas na verdade a biblioteca possui mais tags com outros focos. Um exemplo é o **fmt**, que serve para a formatação de datas, números e i18n, isto é internacionalização. Quando escrevemos uma página JSP, ela pode atender no idioma português, inglês ou francês, varia de acordo com o seu interesse, e é justamente para isso que serve esta biblioteca.

Para utilizarmos o **core** e o **fmt** sempre teremos de ter as seguintes declarações no início da página - lembrando que o prefixo pode ser escolhido livremente:

**core - controle de fluxo** <%@ taglib uri = "http://java.sun.com/jsp/jstl/core" prefix = "c"%>

**fmt - formatação/i18n (internacionalização)** <%@ taglib uri = "http://java.sun.com/jsp/jstl/fmt" prefix = "fmt"%>

Nessa aula falamos sobre Expression Language e JSTL para fugir dos scriptlets. Em detalhe vimos:

* Expression Language (EL) é uma linguagem simples e limitada para imprimir o resultado de uma expressão
* EL usa a sintaxe de ${ .. }
* JSTL é a biblioteca padrão de tags
* JSTL não vem com Tomcat e precisamos copiar um JAR
* JSTL define 4 taglibs, as mais importantes são core e fmt
* a taglib core serve para controle de fluxo, fmt para formatação
* é preciso importar as taglib, core e fmt separadamente:
* <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>COPIAR CÓDIGO

* JSTL e EL devem ser usados em conjunto
* vimos várias tags do core como c:if, c:forEach e c:url
* da fmt vimos a tag fmt:formatDate
* JARs ficam na pasta WEB-INF/lib do projeto

**Redirecionamento Client Side / Redirecionamento Server Side**

O nome desse processo é **Redirecionamento Client Side**, isto é, "do lado do cliente". Quando utilizamos despachadores de um Servlet para outro, esse processe é denominado **Redirecionamento Server Side**, "do lado do servidor".

Vimos que a última requisição feita no lado do servidor pode ser repetida caso o usuário pressione a tecla F5 ou qualquer outro atalho com o mesmo efeito. Isso pode se tornar um grande problema quando a operação é de escrita, pois podemos acabar repetindo informações.

Marque a opção que resolve a questão de repetição da última requisição:

 Após a operação de escrita ser realizada, podemos redirecionar o navegador que realizará uma nova requisição, no caso de leitura. Se o usuário pressionar F5, ele repetirá a operação de leitura.

Nessa aula aprendemos:

* o problema de reenviar uma requisição
* a diferença entre redirecionamento pelo cliente e servidor
* para redirecionar pelo navegador usamos o método response.sendRedirect("endereço")
* o código de resposta para redirecionamento HTTP é 30X (301 ou 302)

Nesta aula, aprendemos:

* Como implementar a remoção de empresas
* Como implementar a edição de empresas
* Qual atributo correto devemos utilizar para identificação dos elementos (id)
* Como definir um input escondido
* O que significa CRUD

XML

O arquivo XML sabe da existência da classe OiMundoServlet, assim como a url pattern /ola, mas não conhece a relação entre os dois elementos. Por isso precisaremos criar um apelido para o Servlet, que deve ser feito com o uso do elemento <servlet-name>, que será OiMundoServlet. Essa declaração deve estar presente tanto no mapeamento da classe quanto no da url pattern.

<display-name>gerenciador</display-name>

<welcome-file-list>

<welcome-file>bem-vindo.html</welcome-file>

</welcome-file-list>

<servlet>

<servlet-name>OiMundoServlet</servlet-name>

<servlet-class>br.com.alura.gerenciador.servlet.OiMundoServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>OiMundoServlet</servlet-name>

<url-pattern>/ola</url-pattern>

</servlet-mapping>

</web-app>

Servlet

**Servlet é um objeto que pode ser acionado por meio de uma requisição do protocolo HTTP**. Essa interação é possibilitada pelo Tomcat, e precisa seguir algumas regras, como estender, sobrescrever os métodos doGet(), doPost() e service(), e fazer o mapeamento para indicar a URL.

Como o Servlet é um objeto, devemos criar uma instância da classe OiMundoServlet.java. Repare que em nenhum local do projeto temos o método main(). Quem participou dos cursos básicos do Java sabe que sempre criávamos esse método e, a partir dele, criávamos a nossa aplicação, instanciando objetos, chamando métodos etc. Veja o exemplo a seguir:

public static void main(String[] args) {

OiMundoServlet servlet = new OiMundoServlet();

servlet.service(req, resp);

} COPIAR CÓDIGO

Não fizemos esse procedimento, mas o objeto da classe OiMundoServlet precisa ser criado... e quem faz isso é o Tomcat, nosso ***servlet container***. Isso ocorre porque é o Tomcat que receberá a requisição HTTP, e depois deve chamar o método necessário.

Ou seja, o Tomcat realiza o papel intermediário entre o navegador e objeto, e por isso também é conhecido como *middleware*.

Para provarmos esse ponto, criaremos um construtor sem argumentos em oiMundoServlet, e usaremos o System.out.println() para imprimir uma mensagem:

public oiMundoServlet() {

System.out.println("Criando Oi Mundo Servlet");

} COPIAR CÓDIGO

Para testar, acessaremos o endereço <http://localhost:8080/gerenciador/ola> no navegador. Será impressa na tela a mensagem "*oi, parabens vc escreveu o primeiro servlets*". Já no console, receberemos a mensagem "*Criando Oi Mundo Servlet*...

Ou seja, o Tomcat criou o Servlet. Porém, reiniciarmos o servidor, notaremos que ele não criará o Servlet de forma automática, isto é, ele não instanciará todos os Servlets de uma vez. No momento em que alguém chama o mapeamento associado com esse Servlet, ele é criado. O Tomcat só criará o objeto quando ele for completamente necessário.

Para demonstrarmos melhor esse comportamento do Tomcat, na página <http://localhost:8080/gerenciador/ola>, repetiremos a requisição diversas vezes pressionando o atalho "F5".

No console apesar das diversas requisições, o Tomcat terá criado apenas uma instância do Servlet, chamando uma única vez o construtor. O objeto sempre fica em memória no Tomcat, e esse objeto é reaproveitado nas próximas requisições.

Criando Oi Mundo Servlet

o servlet OiMundoServlet foi chamado

o servlet OiMundoServlet foi chamado

o servlet OiMundoServlet foi chamado

o servlet OiMundoServlet foi chamado

o servlet OiMundoServlet foi chamado

No Tomcat teremos apenas um Servlet, oiMundoServlet ou ListaEmpresaServlet, e isso se eles forem chamados. Por isso o Servlet é chamado de ***Singleton***, um escopo, que sobrevive no projeto por tempo indeterminado enquanto o Tomcat estiver no ar, sem nunca recriá-lo.

Esse processo poderia ser diferente, isto é, o Tomcat poderia recriar um Servlet a cada nova requisição. Existem outras bibliotecas ou frameworks que recriam objetos a cada nova requisição recebida pelo servidor. O escopo é aquilo que determina quanto tempo vive um objeto, e por padrão ele é *Singleton*.

Esse assunto faz parte de um tópico que chamamos de **inversão de controle**, em inglês **IOC** (***-Inversion Of Control***). Isso significa que o método main() é quem instancia o objeto, mas no caso do nosso projeto em realiza esse processo é o Tomcat, nós apenas criamos as classes.

O Tomcat trabalha com o protocolo HTTP e nos envia os dados de forma que não precisemos nos preocupar com o *parsing* dos cabeçalhos e assim por diante.

Nós poderíamos desenvolver nosso próprio servidor, mas geralmente não é assim que se ganha dinheiro. A atividade lucrativa neste caso é criar modelos de negócio para empresas, com suas devidas diferenças e especificidades.

O Tomcat poderia ser mais complexo e permitir, por exemplo, a criação e a configuração de um Servlet para cada requisição, ou ainda, que ele pudesse gerenciar transações com o banco de dados, verificar regras de segurança antes de fazer a chamada para o Servlet e assim por diante.

Esses são assuntos para *containers* mais sofisticados. Um deles é o ***SpringMVC***, que possui uma infraestrutura muito maior que o Tomcat. Ele realiza configurações de segurança, cash, escopo, entre outras, e nós apenas criamos as classes.

O mais importante dessa explicação é entendermos o conceito da *inversão de controle*, que é realizada pelo servidor.

Vimos durante a aula o conceito de inversão de controle (IOC). No nosso contexto, o que é inversão de controle?

Inversão de controle significa que não é meu método main que instancia objetos, quem se preocupa com a criação das instâncias é o Tomcat.

Correto! Normalmente instanciamos num método main, o que não ocorre nesse caso.

Aprendemos que o Tomcat só instancia a servlet quando realmente ocorre uma requisição. Isso é o comportamento padrão que podemos alterar facilmente! A anotação @WebServlet possui um atributo loadOnStartup que muda esse comportamento:

@WebServlet(urlPatterns="/oi", loadOnStartup=1)

public class OiMundoServlet extends HttpServlet {

public OiMundoServlet() {

System.out.println("Criando Oi Mundo Servlet");

}

@Override

protected void service(HttpServletRequest req, HttpServletResponse resp) throws IOException {

//implementacao do metodo service omitida

}

}COPIAR CÓDIGO

Ao reiniciar o Tomcat será criado o objeto da servlet OiMundoServlet e consequentemente aparece no console a mensagem definida no construtor. Ou seja, o Tomcat não é tão preguiçoso não :)

Deploy

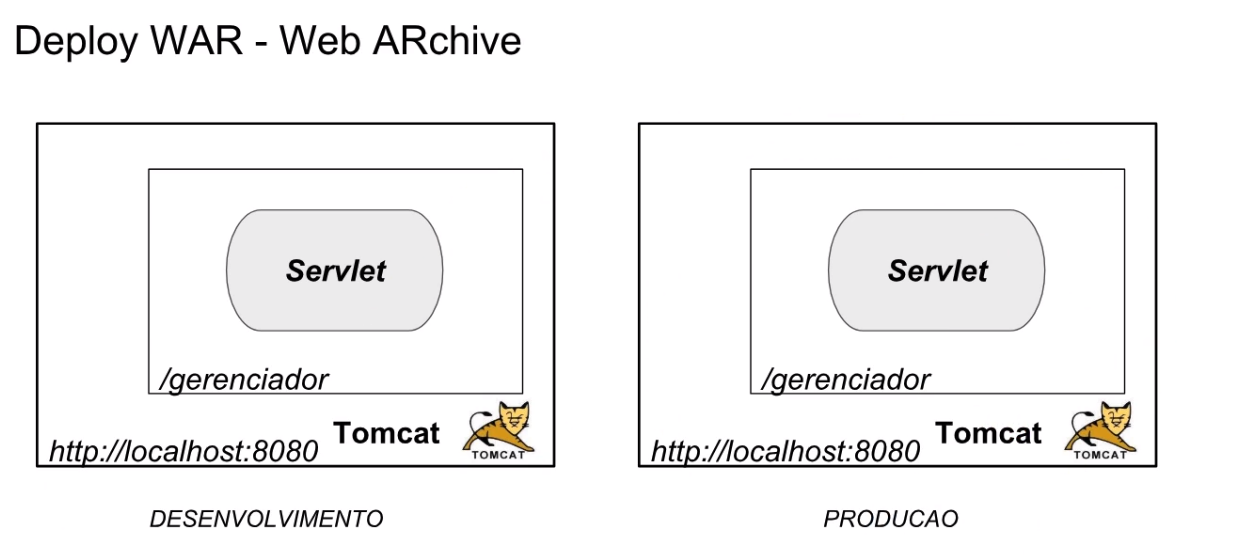
Bem-vindos! Esta aula teremos mais abordagens práticas, já que terminamos de abordar o tópico conceitual de inversão de controle. Para finalizarmos a primeira parte do nosso curso, realizaremos o **deploy**.

Estamos criando uma aplicação em nossa máquina e em algum momento precisaremos e gostaríamos de colocar nosso código em produção, o que significa executá-lo em outra máquina, seja para que o cliente veja o projeto ou para que aplicação seja de fato utilizada, afinal o usuário final não acessará nossa máquina de desenvolvimento.

Para simular o processo de deploy, criaremos mais uma instância no Tomcat. Temos ainda o arquivo zip do download do Tomcat, no início de nosso curso, versão 9.0.11. Copiaremos esse arquivo e colaremos na raiz da pasta D:, feito isso extrairemos o conteúdo do arquivo zip. Renomearemos esse arquivo extraído de apache-tomcat-9.0.11 para apache-tomcat-9.0.11-prod. É muito comum utilizar o sufixo "prod", refere-se ao termo "produção" ou "production". Agora temos um Tomcat cru, não é o mesmo que utilizamos para o desenvolvimento da aplicação.

Passaremos os arquivos do projeto para a nova instância do Tomcat, e temos vários! São classes Java, arquivos de configuração, web.xml, bibliotecas e outros arquivos formato jsp. A ideia é compactarmos os arquivos e depois transferi-los, semelhante ao processo que fazemos com biblioteca, que nada mais são várias classes .jar compactadas em um zip.

Existe uma organização parecida no mundo web, mas o formato dos arquivos não são .jar, e sim .war, isto é, Web Archive. Passaremos os arquivos de **desenvolvimento** para a instância de **produção**, ou seja, realizaremos o processo de deploy.



Este processo pode conter diversas instâncias intermediárias, por exemplo uma instância responsável pela execução de testes, homologação ou produção, constituindo uma pipeline que garantirá a qualidade do projeto.

Neste caso, simularemos nosso deploy com apenas duas instâncias do Tomcat. Criaremos o arquivo .war, e para isso usaremos o Eclipse. Clicaremos com o botão direito sobre gerenciador selecionaremos as opções "Export > WAR File". Caso esta opção não está disponível para você, acessaremos " Export > Export...". Na nova caixa de diálogo, escreveremos "WAR" na opção "Select an export wizard", e em seguida configuraremos o destino desse arquivo para a pasta raiz D:, com o nome de gerenciador.war.

O arquivo .war é semelhante ao .jar, trata-se um zip renomeado. Vamos fazer o teste acessando nossa pasta raiz e renomeando o arquivo gerenciador.war para gerenciador.zip. Em seguida, extrairemos o arquivo zipado. Neste novo arquivo extraído, notaremos que há todo o conteúdo da pasta "WebContent", como "META-INF", "WEB-INF" formAlteraEmpresa.jsp, novaEmpresaCriada.jsp e assim por diante.

Temos, contudo, uma pequena mudança: ao acessarmos a pasta "WEB-INF" veremos o arquivo web.xml, "lib" e "classes", Esta última pasta não é exibida no Eclipse. O padrão da especificação Servlet é que existe na "WEB-INF" uma pasta "classes" que abrigará todas as classes compiladas do projeto. Dentro desse arquivo war teremos todo o código - não o .java, mas o .class - responsável pela execução do programa.

Renomearemos novamente gerenciador.zip para gerenciador.war, dessa forma o Tomcat poderá ser utilizado. Arrastaremos este arquivo para a pasta apache-tomcat-9.0.11-prod. Não é apenas o Tomcat que poderíamos usar, existem diversos servidores Java como Jetty, que também sabem trabalhar com o formato .war e especificação Servlet. Precisaremos inserir gerenciador.war na pasta de deploy, que normalmente possui o mesmo nome. Neste caso, especificamente, o Tomcat atribuiu à pasta de deploy o nome "webapps".

Com os arquivos alocados, resta subirmos a nova instância no Tomcat. Não queremos realizar a configuração via Eclipse, afinal não o instalaremos em uma máquina na nuvem da Amazon, e talvez essa máquina nem tenha uma interface gráfica. Iremos inicializar a máquina na linha de comando.

Abriremos uma Prompt de Comando, isto é, o terminal. Acessaremos o Tomcat de produção:

C:\Users\Alura>d:

D:\>cd apache-tomcat-9.0.11-prodCOPIAR CÓDIGO

Devemos achar a pasta para subir o Tomcat, que normalmente é chamada de bin e contém os arquivos de inicialização:

D:\>cd apache-tomcat-9.0.11-prod\bin>startupCOPIAR CÓDIGO

No sistema operacional Linux, o arquivo de inicialização é o startuo.sh, já no Windows startup.bat. Usaremos este último.

D:\>cd apache-tomcat-9.0.11-prod\bin>startup.batCOPIAR CÓDIGO

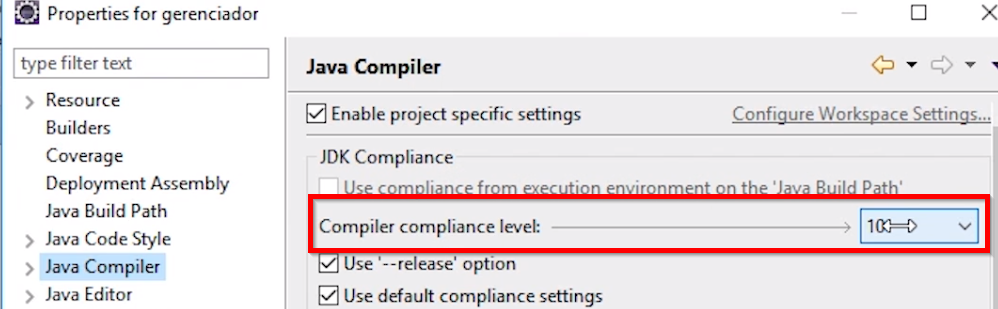
Será aberta uma nova janela do terminal em que será executado o processo Java. Por fim, acessaremos novamente apache-tomcat-9.0.11-prod e veremos que uma nova pasta gerenciador foi criada, essa pasta contém todos os arquivos necessários para executar nossa aplicação.

Testaremos no navegador todas as modificações que realizamos. No navegador, escreveremos localhost:8080 e verificaremos que o Tomcat é executado, estaremos acessando a uma tela padrão do Tomcat, já que o iniciamos fora do Eclipse. Agora abriremos nosso gerenciador, então, no navegador escreveremos localhost:8080/gerenciador/, e teremos a seguinte mensagem exibida na tela:

Bem-vindo no curso de Servlets da Alura!COPIAR CÓDIGO

O html inicial foi exibido, como esperávamos. Tentaremos agora listar as nossas empresas acessandolocalhost:8080/gerenciador/listaEmpresas/. E teremos uma mensagem de erro HTTP Status 500 - Internal Server Error, a root cause é UnsupportedClassVersionError, relacionada a class file version 54.0. Isso ocorreu porque compilamos as classes com uma versão mais recente daquela que usamos para executar o Tomcat.

No Eclipse, podemos consultar qual a versão do Java que estamos utilizando para a compilação: clicaremos com o botão direito sobre a pasta "gerenciador" e selecionaremos a opção "Properties", na caixa de diálogo, procuraremos por "Java Compiler". Verificaremos que a versão utilizada é a mais recente (10) disponível no momento de gravação deste curso.



Ao acessarmos o console do Tomcat veremos que a versão executada é 1.8. Poderíamos realizar a recompilação no Eclipse utilizando uma outra versão, mas ao invés disso instruiremos o Tomcat a utilizar a versão mais recente.

Vamos entender o funcionamento do Tomcat para que possamos entender o que deve ser feito. No Explorer, clicaremos com o botão direito sobre "Este Computador" e selecionaremos a opção "Propriedades". Na nova caixa de diálogo, clicaremos sobre "Configurações avançadas do Sistema", em seguida "Variáveis de Ambiente". Caso você esteja utilizando outro sistema operacional, haverá outros percursos da janela para acessar as variáveis de ambiente, mas o mesmo conceito existe.

Há uma variável de ambiente que o Tomcat utiliza: JAVA\_HOME, localizada em C:\Program Files (x86)\Java\jre1.8.0\_171\bin, que define qual máquina virtual será utilizada. Clicaremos sobre a opção "Editar" e configuraremos o destino C:\Program Files\Java\jre-10.0.2.

Teremos mais uma variável de ambiente que pode nos atrapalhar, Path, configurando em diversas pastas de diferentes e que por sua vez define quais executáveis serão visíveis na linha de comando. Uma das pastas é C:\Program Files\Java\jre-10.0.2\bin, ela está correta, afinal o executável do Java está armazenado dentro de uma pasta bin.

Abriremos novamente o terminal e acessaremos a pasta do Tomcat, acionaremos o arquivo de inicialização startup.bat:

D:\apache-tomcat-9.0.11-prod\bin>startup.batCOPIAR CÓDIGO

Dessa forma, teremos a versão correta do Java sendo executada. Acessaremos no navegador a nossa lista de empresas via localhost:8080/gerenciador/listaEmpresas, e teremos o conteúdo esperado exibido na tela:

Lista de empresas:

. Alura - 03/09/2018 edita remove

. Caelum - 03/09/2018 edita removeCOPIAR CÓDIGO

Podemos, inclusive, clicar em "edita" e realizar alterações nas empresas que constam na lista, bem como removê-las. Dessa forma, conseguimos executar a aplicação em outro Tomcat, com o ambiente de desenvolvimento na mesma versão que o ambiente de produção.

Nossa aplicação está executando, e fizemos esse processo de maneira manual para entendermos melhor seu funcionamento interno. Em um projeto real, todas essas etapas seriam automatizadas.

Veremos um último item: no console do Tomcat, todas as saídas System.out estão visíveis. Se quisermos realizar alguma alteração, precisaremos trabalhar no código fonte, isto é, precisaríamos criar o war novamente. A plataforma de ensino **Alura** opera por meio de um Tomcat instalado na nuvem, mas ela não é executada na porta 8080, como isso se dá?

Acessaremos o Tomcat de produção, afinal a porta é algo definido pelo servidor. Na pasta apache-tomcat-9.0.11-prod teremos a pasta "conf", em que encontraremos o arquivo server.xml, que precisa ser editado. É importante utilizarmos um editor previsível, para que nenhuma alteração fora do nosso controle seja realizada, neste caso usaremos o Notepad++.

Com o arquivo server.xml aberto no Notepad++, utilizaremos o atalho "Ctrl + F" para buscar a porta "8080", e encontremos o trecho de código correspondente:

<Connector port="8080" protocol="HTTP/1.1"

connectionTimeout="20000"

redirectPort-"8443" />COPIAR CÓDIGO

Modificaremos a porta 8080 simplesmente para 80, porque essa é a porta padrão do protocolo http e é esse valor assumido quando não especificamos a porta. Assim feito, iremos reiniciar o Tomcat e acessaremos nosso projeto no navegador digitando simplesmente localhost/gerenciador, sem qualquer problema.

Vimos durante os vídeos que para realizar o deploy da nossa aplicação é necessário exportar um arquivo do tipo WAR. Em outros tipos de aplicações seria necessário exportar um arquivo do tipo JAR.

No final, qual a diferença entre esses dois tipos?

WAR é o ZIP de projetos Java Web, que você roda de dentro de um servidor como o Tomcat.

JAR é o ZIP para projetos comuns, que não são Web.

Correto! É exatamente isso! JAR e WAR são nada mais do que arquivos ZIP, no entanto um WAR possui os arquivos do mundo web como imagens, CSS, JS, JSP e HTML.

Nesta aula, aprendemos:

* A importância do web.xml
* O que é inversão de controle
* Como o Tomcat trabalha por debaixo dos panos
* Como inicializar o Tomcat na linha de comando
* Como gerar o WAR (Web ARchive)
* Como fazemos o deploy da nossa aplicação