03

Inserindo Produto como dependência

Transcrição

O arquivo `.war`, gerado a partir do projeto `lojaweb`, possui as dependências que incluímos ao longo da construção da aplicação, que são Caelum Stella e Javax Servlet. Contudo, o escopo de teste do JUnit não está presente. Observe o trecho do arquivo `pom.xml`:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
</dependency>
```

O escopo de teste não entra no arquivo `.war`, pois ao inserirmos o `<scope>` de teste na dependência não usamos o padrão de compilação, que é `<scope>compile</scope>`, que faz com que o projeto esteja disponível para compilação para quem depender dele. Por exemplo, se iremos realizar um *deploy* no servidor de aplicação, precisaremos das dependências do projeto, e graças ao `compile` elas estarão disponíveis para compilação em tempo de execução.

Adicionaremos mais um tipo de dependência, a do projeto `produtos`. Em `pom.xml` de `lojaweb` incluiremos o `<artifactId>` e a `<version>` na nova tag `<dependency>`.

```
<dependency>
    <groupId>br.com.alura.maven</groupId>
    <artifactId>produtos</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

O nosso arquivo será atualizado, mas não ocorrerá nenhum download, pois o projeto `produtos` não está na internet. O que ocorre é uma busca no diretório local pela biblioteca. Dessa forma, o projeto `lojaweb` passa a depender de `produtos`.

Verificaremos se tamos acesso às classes de produtos: em `ContatoServlet.java`, tentaremos acessar `Produto`, lembrando que ele necessita de um construtor com `nome` e `preco`, e essa informação é exibida para nós.

```
@Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletExcept
        new Produto();
        PrintWriter writer = resp.getWriter();
        writer.println("<html><h2>Bata um papo conosco</h2></html>");
        writer.close();
    }
}
```

No momento em que configuramos o `pom.xml`, incluindo a dependência de `produtos`, dessa forma, `lojaweb` e `produtos` são **interdependentes**, e quando um for "buildado" será necessário o arquivo `.jar` de outro.

Geraremos novamente o *package* de `lojaweb` e checaremos as informações. Notaremos a mensagem de erro `BUILD FALIURE` na linha de comando.

```
[WARNING] The POM for br.com.alura.maven:produtos:jar:1.0-SNAPSHOT is missing, no dependency in
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
```

Aparentemente `produto:jar:1.0-SNAPSHOT` não é encontrado, embora `Produtos` tenha sido compilado em `ContatoServlet`, como verificamos anteriormente. Faremos uma investigação nas configurações do Maven no momento em que inserimos a nova dependência.

Ao analisarmos o diretório "Libraries > Maven Dependencies", não encontraremos o `.jar`, mas sim o projeto `produtos` no *classpath*. O Maven, ao atuar com o Eclipse, possui uma desvantagem: caso o arquivo `.jar` fosse inserido no diretório `Maven Dependencies`, teríamos de gerar um `.jar` toda vez que fizéssemos alguma mudança para testes, e então atualizá-lo.

Se na classe `Produto` queremos inserir uma `String categoria`. Quando formos declarar `Produto` em `ContatoServlet`, a modificação será automática, e a classe passará a precisar de `nome`, `preco` e `categoria`. O Eclipse percebe a interdependência de projetos que estão no IDE, portanto ele cria referenciações entre os projetos, e não ao `.jar`.

O lado negativo disso é que podemos criar tudo que está no *classpath* do projeto, inclusive `ProdutoTest()` e todo o código fonte, o que não faz sentido nenhum. Precisamos tomar um cuidado importante: **não devemos utilizar as classes que estão na parte de teste.** Faremos um pequeno exemplo — na classe `ContatoServlet` inseriremos `Produto p = new Produto("bala", 15.0)`.

Ao executarmos o Jetty, clicaremos com o botão direito do mouse sobre 'lojaweb `na área de "Project Explorer", e escolheremos a opção "Run As > Maven build...", nomearemos como` lojaweb jetty `, e na opção "Goals" escreveremos o comando` jetty:run`.

Teremos a mesma mensagem de erro de antes na linha de comando. O Maven não possui o mesmo mecanismo de funcionamento do Eclipse, portanto ele realmente não encontra o arquivo `.jar`. É necessário instalar o projeto no repositório local, por meio dos comandos:

```
cd
pwd
cd produtos/
mvn install
```

Dessa forma teremos o arquivo `.jar` no repositório local. Não realizar essa instalação é um erro muito comum, por isso devemos ficar atentos. Apesar da sucesso do funcionamento no Eclipse, que realiza o reconhecimento de seus projetos internos, sem a instalação do local o arquivo não será achado.

Dessa forma nós conseguimos executar novamente o Jetty e acessar a página de contato (`localhost:8080/contato`) no navegador. Conseguimos acessar `produto`, mas ainda temos alguns conteúdos para aprender!