01

Começando nossa loja web

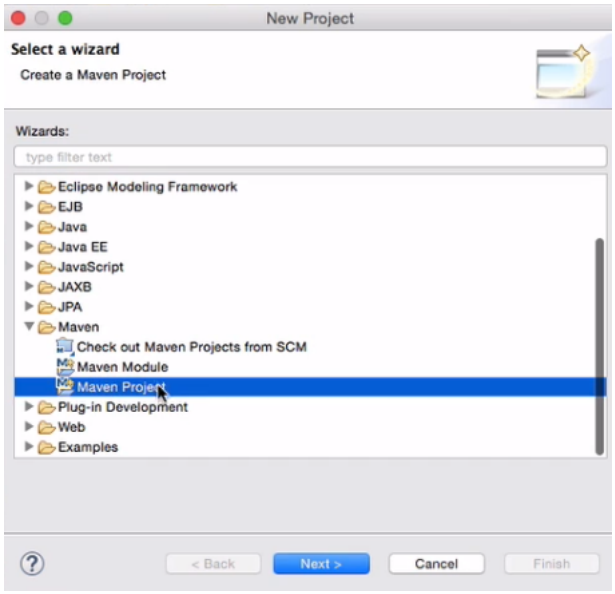
Transcrição

Podemos incluir quantas classes quisermos em nosso projeto. No caso, temos `App` com o método `main`, e a classe de modelo `Produto`. É comum o projeto não ser de um arquivo `.jar`, a ser gerado como biblioteca. Outro caso comum no uso do Maven e do Java é a criação de uma aplicação web, e veremos como isso pode ser feito.

Faremos um projeto novo baseando em um exemplo básico do Maven, usando os seguintes comandos:

```
mvn archetype:generate -DartifactId=blog -DgroupId=br.com.alura.maven -DinterativeMode=false -Da
```

Já aprendemos a utilizar o Maven na linha de comando, e a próxima etapa é utilizar os recursos dessa ferramenta no Eclipse. Na área "Project Explorer" clicaremos com o botão direito e selecionaremos as opções "New > Project". Na caixa de diálogo que se abre selecionaremos a opção "Maven > Maven Project".

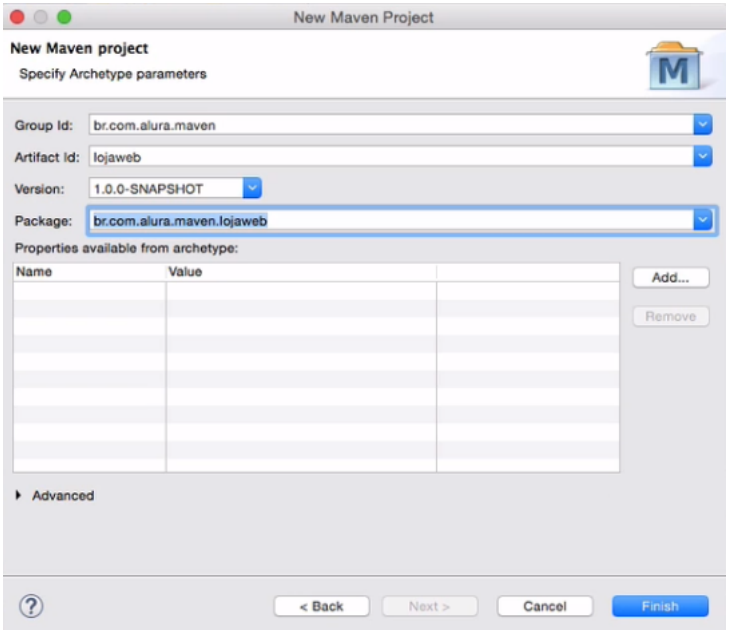


Manteremos a localização do projeto como padrão. Em seguida, é solicitado que escolhemos o arquétipo a ser utilizado na construção do projeto. Na caixa de diálogo encontraremos o item "Catalog", que fornece diversas opções de catálogos, e nos escolheremos "All Catalogs".

Escolheremos o arquétipo `org.apache.maven.archetypes maven-archetype-webapp` na versão `1.0`. Uma Web App é uma aplicação web baseada na API Servlet do Java, que podemos "deployar" em um servidor que tenha suporte para ela, como Tomcat, Jetty ou JBoss. Assim feito, pressionaremos o botão "Next".

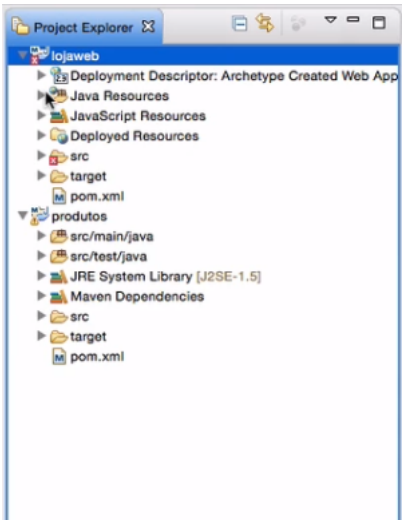
Na próxima etapa teremos campos a serem preenchidos: "Group Id", em que escreveremos a informação `br.com.alura.maven`, e "Artifact Id", que chamaremos de `lojaweb`, isto é, o nome do nosso site. Em seguida, na opção "Package" escolheremos a versão `1.0.0-SNAPSHOT`.

O *package* que usaremos neste projeto será `br.com.alura.maven.lojaweb`.



O pacote está um pouco diferente do que criamos na linha de comando, mas está de acordo com os nossos interesses, afinal temos o nome da empresa, `alura`, e o nome do projeto, `lojaweb`.

Nosso projeto está criado diretamente no Eclipse, e com isto podemos começar a explorar quais elementos ele contêm. Na área "Project Explorer" clicaremos sobre a pasta "Java Resources > Libraries", e verificaremos que a única dependência é o JUnit, por padrão. Não temos nenhuma classe Java.



Na pasta `src` temos o diretório `webapp`, local em que iremos armazenar os arquivos `.jsp`. Caso usemos outra tecnologia para desenvolver web com Java, esse diretório comportará arquivos HTML, por exemplo. Estamos fazendo uso de uma estrutura tradicional de um projeto web Java.

E para conseguirmos executar o projeto, precisamos baixar um servidor. Em vez de entrarmos nos sites do Jetty, Tomcat e JBoss para realizar as configurações em cada um deles, usaremos os plugins do Maven.

Dos três servidores mencionados, o Jetty é o mais leve, e por este motivo o usaremos em nosso projeto. Sua documentação (https://www.eclipse.org/jetty/documentation/9.4.x/jetty-maven-plugin.html) possui alguns *goals* em destaque, sendo um deles `jetty:run`.

Assim, encontraremos o código a ser usado para implementar o plugin em nosso projeto:

```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>9.3.7.v20160115</version>
  <configuration>
    <scanIntervalSeconds>10</scanIntervalSeconds>
    <webApp>
      <contextPath>/test</contextPath>
    </webApp>
  </configuration>
</plugin>
```

Utilizaremos apenas as três primeiras linhas e descartaremos as configurações, afinal queremos apenas os recursos padrão. É importante destacar que não há necessidade alguma de decorarmos a versão dos plugins, pois a informação de que precisamos consta na documentação, e ela é bem simples de ser encontrada.

Dentro de `<build>` do arquivo `pom.xml`, inseriremos os nossos `<plugins>`:

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
    </dependecy>
<dependencies>
<build>
    <finalName>lojaweb</finalName>
        <plugins>
            <plugin>
                <groupId>org.eclipse.jetty</groupId>
                <artifactId>jetty-maven-plugin</artifactId>
                <version>9.3.7.v20160115</version>
            </plugin>
        </plugins>
</build>
```

Na linha de comando, acessaremos o diretório `lojaweb`. Em seguida compilaremos o projeto utilizando o comando `mvn compile`.

```
pwd
cd ..
pwd
cd lojaweb/
mvn compile
```

Por enquanto não temos nenhuma classe, portanto não há nada para ser compilado. Executaremos o comando `mvn jetty:run` para instalar o plugin em nosso projeto. Ao final do procedimento de instalação, seremos avisados de que o Jetty pode ser executado na porta `8080`. No browser, acessaremos o endereço `localhost:8080`, e veremos a mensagem "Hello World!".

As próximas etapas envolvem a construção tradicional de um projeto web. No Eclipse podemos criar as classes, os JSPs, HTMLs, e realizar configurações nas dependências.

> Para aprender mais sobre como construir projetos web, na Alura temos os cursos de Servlet (https://www.alura.com.br/curso-online-servlets-fundamentos-programacao-web-java), JSP (https://cursos.alura.com.br/course/jstl), e de frameworks diversos, como JSF (https://cursos.alura.com.br/course/jsf) e VRaptor (https://cursos.alura.com.br/course/projeto-controle-horas-java).

Continuando, em "src > main > resources > webapp > WEB-INF > index.jsp" encontraremos o código de suporte da mensagem "Hello World!", que vimos no browser. Mudaremos para `Minha loja web!` :

```
<html>
<body>
<h2>Minha loja web!</h2>
</body>
</html>
```

O código será recompilado, e no navegador será exibida a nova mensagem. Ainda existem alguns detalhes que podem ser explorados, como as configurações do plugin que ignoramos em um primeiro momento, e a mensagem de erro que surge no arquivo `index.jsp` ao alterarmos a mensagem.