05

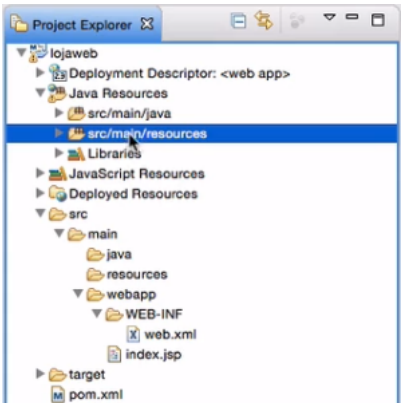
Incrementando nossa loja web

Transcrição

Aprendemos a configurar o projeto web para utilizar uma versão mais recente de Servlet, e como criá-lo no Eclipse. Temos o arquivo `index.jsp`, como qualquer projeto tradicional. No diretório `Java Resources`, podemos criar as classes Java do nosso interesse.

Ela será armazenada em uma nova pasta, a ser criada em `main`, chamada `java`. O Eclipse automaticamente perceberá que há um *source folder* `src/main/java`, e reagrupará os diretórios fazendo com que este último fique acima de `src/main/resources` na área "Project Explorer".

Em `src/main/resources` podemos incluir arquivos *properties*, TXT, e assim por diante. Em `src/main/java` ficarão armazenadas as classes Java.



Clicaremos com o botão direito sobre o diretório `src/main/java` e selecionaremos as opções "New > Class". Na nova caixa de diálogo, configuremos para que o "Package" da classe seja `br.com.alura.maven.lojaweb`, e o seu nome, `ContatoServlet`.

Uma Servlet deve ser estendida para `HttpServlet`, e isso só é possível porque a adicionamos no nosso *classpath*, como dependência no arquivo `pom.xml`.

```
package br.com.alura.maven.lojaweb

public class ContatoServlet extends HttpServlet{

}
```

Em seguida, implementaremos o método `doGet()` da Servlet na classe. Escreveremos o método e pressionaremos o "Ctrl + Barra de espaço".

```
package br.com.alura.maven.lojaweb;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
```

```
import javax.servlet.http.HttpServletResponse;

public class ContatoServlet extends HttpServlet{

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletExcept
        //TODO Auto-generated method stub
        super.doGet(req, resp);
    }
}
```

Retiraremos o conteúdo do método e faremos uma implementação simples da Servlet, apenas para demonstrar como se dá um projeto web tradicional. Acionaremos o `resp` e utilizaremos o método `getWriter()`. Em seguida, pressionaremos "Ctrl + 1" e adicionaremos a variável local `writer`, que por sua vez receberá uma mensagem simples em HTML, que será destacada pro meio de `<h2>`. Para fecharmos, usaremos o método `close()`.

```
public class ContatoServlet extends HttpServlet{

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletExcept
        PrintWriter writer = resp.getWriter();
        writer.println("<html><h2>Entre em Contato</h2></html>");
        writer.close();
    }
}
```

Desse modo, imprimimos a mensagem `Entre em Contato`.

De que forma acessamos a Servlet pelo navegador? Primeiramente precisamos de uma URI; na classe `ContatoServlet` anotaremos `WebServlet()`, e seu `urlPatterns` será `/contato`. Isto quer dizer que no momento em que acessarmos `/contato` em nosso browser, em tese, seremos direcionados para a Servlet.

```
@WebServlet(urlPatterns={"/contato"})
public class ContatoServlet extends HttpServlet{

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletExcept
        PrintWriter writer = resp.getWriter();
        writer.println("<html><h2>Entre em Contato</h2></html>");
        writer.close();
    }
}
```

No momento em que digitamos `localhost:8080/contato` em nosso navegador, temos uma mensagem de erro:

```
HTTP ERROR 404

Problem accessing /contato. Reason:

    Not Found
```

Iremos até o terminal e reiniciaremos nosso programa por meio do comando `mvn jetty:run` para tentar resolver o problema. Dessa vez, ao acessarmos `localhost:8080/contato`, conseguiremos ler a mensagem `Entre em Contato`, como gostaríamos.

Para o `.jsp`, não é necessário reiniciar a compilação, mas o mesmo não vale para as classes Java. Seria mais interessante que não precisássemos reiniciar o programa todas as vezes. O Jetty poderia detectar as mudanças e realizar um *hot swap*, isto é, uma mudança na *Virtual Machine*.

Neste momento voltaremos para as configurações do Jetty que ignoramos a princípio:

```
<plugin>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>9.3.7.v2016115</version>
  <configuration>
    <scanIntervalSeconds>10</scanIntervalSeconds>
    <webApp>
      <contextPath>/test</contextPath>
    </webApp>
  </configuration>
</plugin>
```

Na tag `<scanIntervalSeconds>` temos o intervalo de tempo em segundos para que o Jetty escaneie as classes de um projeto, no caso, `10`. Por padrão, esse valor é `0`. Agora, usaremos apenas o trecho de código que contém as configurações `<scanIntervalSeconds>`, e o inseriremos em nosso arquivo `pom.xml`, no ponto que diz respeito ao plugin de Jetty.

```
<***!***>

<plugin>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-manven-plugin</artifactId>
    <version>9.3.7.v2016115</version>
          <configuration>
    <scanIntervalSeconds>10</scanIntervalSeconds>
        </configuration>
</plugin>
```

Mudaremos a mensagem da nossa Servlet para `Fale Conosco`:

```
@WebServlet(urlPatterns={"/contato"})
public class ContatoServlet extends HttpServlet{

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

Ao acessarmos nossa Servlet no navegador, esperaremos dez segundos e ela será atualizada, exibindo nossa nova mensagem. No terminal, temos a informação do momento exato em que aconteceu a atualização:

```
[INFO] Restart completed at Thu Feb 18 08:23:28 BRST 2016
```

Em tese, o Jetty reiniciou o programa, e as modificações foram atualizadas. Iremos testar para entender se isso foi de fato efetuado ou se, na verdade, houve apenas a alteração do conteúdo na variável na memória, por exemplo. Mudaremos novamente a mensagem da Servlet para `Bata um papo conosco`.

Salvaremos as modificações, e logo em seguida acessaremos o terminal para averiguar se todo o programa foi reiniciado ou apenas um pequeno trecho foi trocado. Analisando o conteúdo exibido no terminal, descobriremos que de fato a aplicação inteira foi reiniciada. Ao acessarmos nossa Servlet no navegador, teremos a mensagem `Bata um papo conosco`.

> Resumindo, para JSPs a aplicação não precisa ser reiniciada quando realizamos modificações, mas o mesmo não vale para as classes Java.

Até agora, aprendemos a configurar o Jetty para fazer uso de *deploys* mais rápidos em nossa aplicação. Há ainda mais configurações que podemos utilizar.

Por enquanto nossa aplicação é executada na **raiz**, um diretório virtual. Podemos adicionar um subdiretório virtual que comporte diversas aplicações. Por exemplo, o blog ocuparia o espaço específico `localhost:8080/blog/contato`, ou a loja do site, `localhost:8080/loja/contato`. Com essa organização, criamos **contextos** na aplicação web.

É exatamente esta a função da configuração `<contextPath>`, presente no modelo da documentação do Jetty. Inseriremos este trecho do código no arquivo `pom.xml`, alterando o nome do contexto de `/text` para `/loja`:

```
<***!***>

<plugin>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-manven-plugin</artifactId>
    <version>9.3.7.v2016115</version>
  <configuration>
    <scanIntervalSeconds>10</scanIntervalSeconds>
        <webApp>
            <contextPath>/loja</contextPath>
        </webApp>
    </configuration>
</plugin>
```

Reiniciaremos a aplicação no terminal por meio do comando `mvn jetty:run`. No navegador, a versão antiga `localhost:8080/contato` não está mais funcional, porém `localhost:8080/loja/contato` e `localhost:8080/loja` estão perfeitamente operantes com suas respectivas mensagens impressas. **Dessa forma, temos a aplicação sendo executada em um contexto.**

A decisão de executarmos a aplicação desta forma, ou não, varia de acordo com o interesse do desenvolvedor. Neste caso, como estamos executando uma aplicação local, iremos remover essa configuração, e trabalharemos com um único servidor. Executaremos a aplicação mais uma vez pelo terminal utilizando o `mvn jetty:run`, e nos atentaremos para uma mensagem específica:

```
[WARNING] Quiet Time is too low for non-native WatchService [sun.nio.fs.PollingWatchService]: 1(
```

Perceba que a unidade referida é de `5000ms`. Na configuração do plugin havíamos colocado `10` segundos. Iremos investigar a documentação (https://www.eclipse.org/jetty/documentation/9.4.x/jetty-maven-plugin.html#jetty-run-goal) para entendermos o que está acontecendo com estes valores.

No tópico `scanIntervalSeconds`, é indicado que a unidade de tempo utilizada para medir a pausa entre as varreduras é segundos. De fato, a verificação está ocorrendo de dez em dez segundos, e para essa verificação acontecer no tempo que nós estipulamos, o Jetty verifica essa passagem de tempo em 1000ms, isto é, os dez segundos são monitorados a cada 1000ms.

O Jetty recomenda que, para serviços não nativos, o número mínimo deva ser de `5000ms`. Quanto menor esse número, menor será a margem de erro no momento do *reload* da aplicação. Esse valor varia de acordo com a implementação, e não há muito o que fazer. Mesmo assim, é interessante entendermos sobre o que se trata essa mensagem de *Warning* para que não haja dúvidas.

Nesta aula, aprendemos sobre a necessidade do *reload* para Servlets, e que para realizá-lo precisamos modificar as configurações do plugin em `<scanIntervalSeconds>`.