

# Aula 1

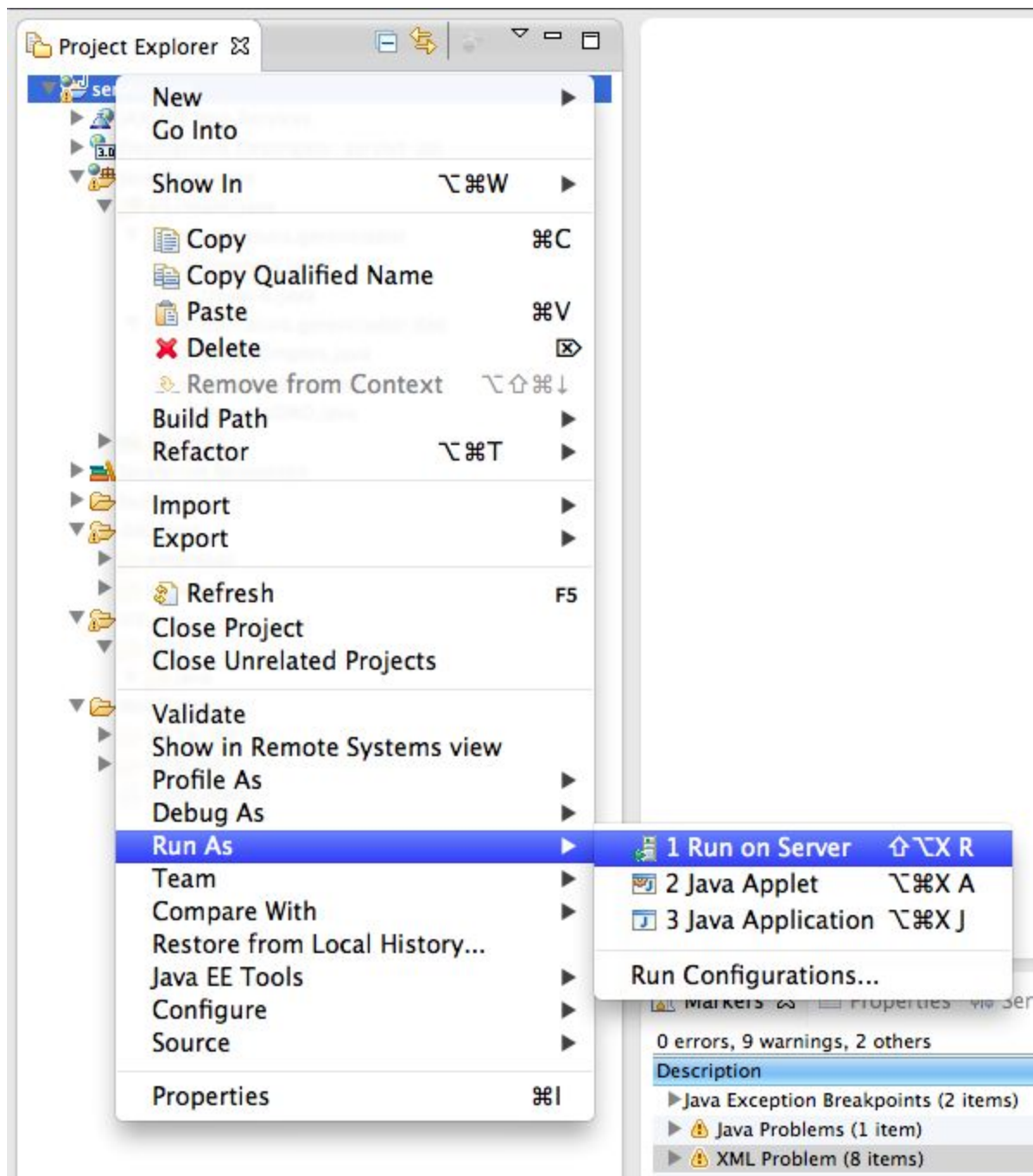
## Servlet API

Nesse curso veremos como criar um sistema de cadastro e busca de empresas, além de um sistema de autenticação em Java. Para entendermos como funcionam os servidores Java e suas bibliotecas usaremos a base da maior parte delas: a Servlet API. Falaremos sobre o protocolo HTTP, algumas de suas características e com o avanço do curso passaremos a utilizar boas práticas, por exemplo padrões de projeto como o Model View Controller (MVC).

A web que conhecemos é composta por requisições que devolvem respostas. Agora chegou a hora de criarmos nosso primeiro pequeno servidor, um servidorzinho, em inglês, uma Servlet. Após baixar o projeto de exemplo contido no exercício você pode descompactá-lo e importá-lo no Eclipse com Import Existing Project into Workspace.

Para rodar o projeto precisamos de um servidor web que implementa a especificação de Servlets e utilizaremos o Tomcat. Entramos em <http://tomcat.apache.org> e baixamos a última versão do mesmo. Descompactamos.

No Eclipse clicamos da direita em nosso projeto, Run As, Run On Server.



Escolha a versão do Tomcat que está utilizando. Note que é importante baixar uma versão do Tomcat que o Eclipse suporta. Selecione também "Always use this server when running this project..."

Run On Server

Select which server to use

How do you want to select the server?

☐ Choose an existing server

☒ Manually define a new server

[Download additional server adapters](#)

Select the server type:

type filter text

- Tomcat v4.0 Server
- Tomcat v4.1 Server
- Tomcat v5.0 Server
- Tomcat v5.5 Server
- Tomcat v6.0 Server
- Tomcat v7.0 Server
- Basic
  - J2EE Preview
- JBoss

Publishes and runs J2EE and Java EE Web projects and server configurations to a local Tomcat server.

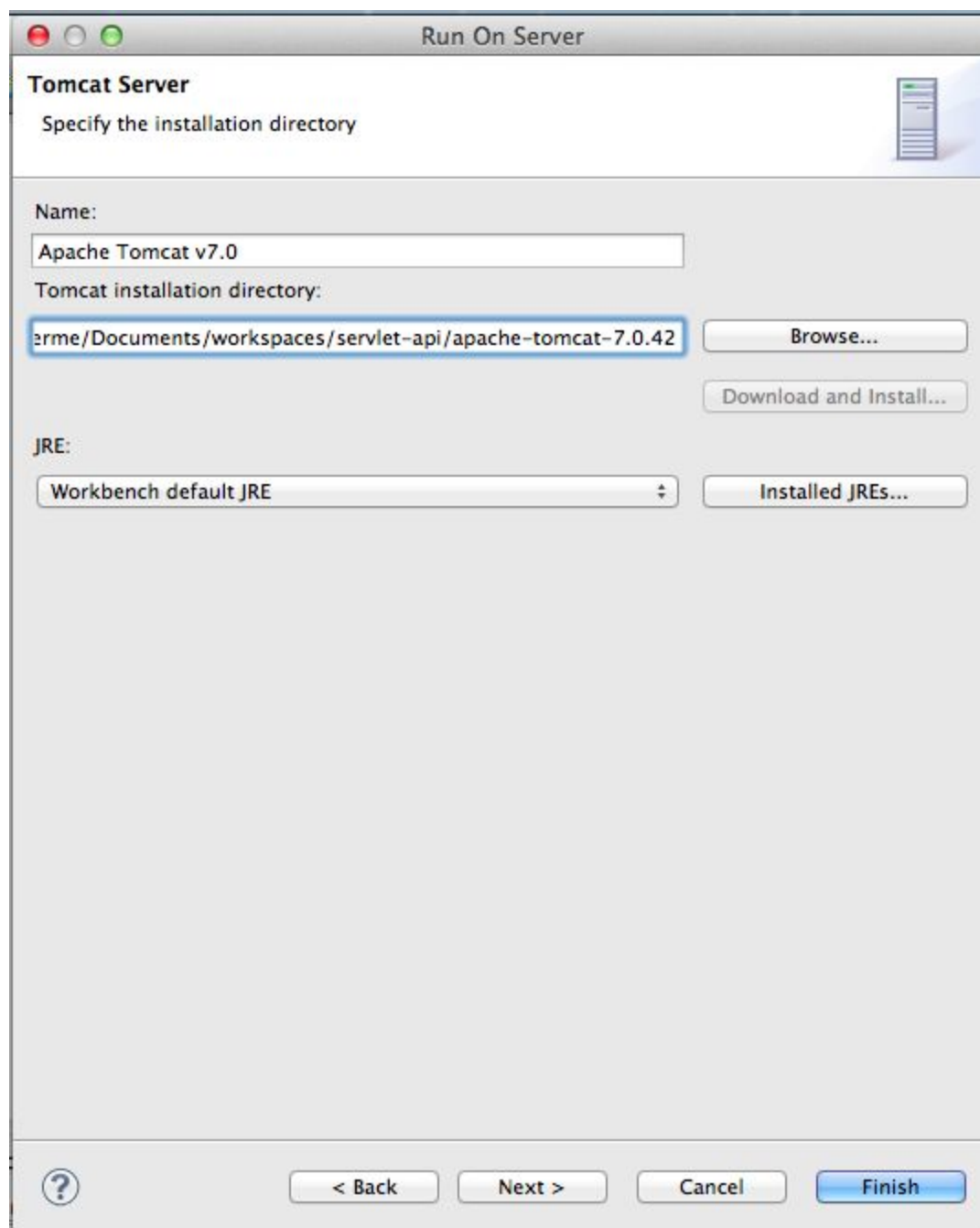
Server's host name: localhost

Server name: Tomcat v7.0 Server at localhost

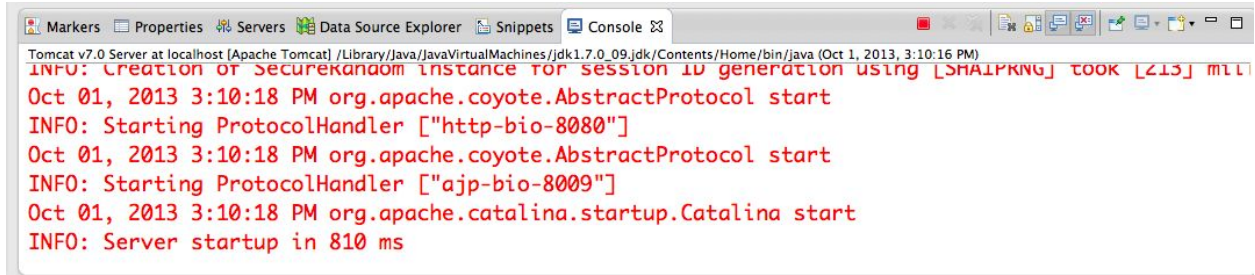
☒ Always use this server when running this project

? < Back Next > Cancel Finish

Selecione Next e escolha agora o diretório onde você descompactou seu Tomcat:



Selecione Finish. O Eclipse levanta o servidor e abre uma janela de browser com a uri de boas vindas: "Bem vindo ao nosso gerenciador de empresas". Na aba Console aparece a mensagem que o servidor está rodando:

The image shows a screenshot of the Eclipse IDE's console window. The title bar indicates it's the 'Console' tab. The text in the console is red and shows the following logs: 'Tomcat v7.0 Server at localhost [Apache Tomcat] /Library/Java/JavaVirtualMachines/jdk1.7.0\_09.jdk/Contents/Home/bin/java (Oct 1, 2013, 3:10:16 PM)', 'INFO: Creation of SecureRandom instance for session ID generation using [SHA1PRNG] took [213] ms', 'Oct 01, 2013 3:10:18 PM org.apache.coyote.AbstractProtocol start', 'INFO: Starting ProtocolHandler ["http-bio-8080"]', 'Oct 01, 2013 3:10:18 PM org.apache.coyote.AbstractProtocol start', 'INFO: Starting ProtocolHandler ["ajp-bio-8009"]', 'Oct 01, 2013 3:10:18 PM org.apache.catalina.startup.Catalina start', and 'INFO: Server startup in 810 ms'.

```
Tomcat v7.0 Server at localhost [Apache Tomcat] /Library/Java/JavaVirtualMachines/jdk1.7.0_09.jdk/Contents/Home/bin/java (Oct 1, 2013, 3:10:16 PM)
INFO: Creation of SecureRandom instance for session ID generation using [SHA1PRNG] took [213] ms
Oct 01, 2013 3:10:18 PM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-bio-8080"]
Oct 01, 2013 3:10:18 PM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["ajp-bio-8009"]
Oct 01, 2013 3:10:18 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 810 ms
```

Vamos testar em um navegador tradicional? Nosso servidor está rodando em nossa máquina (localhost) na porta 8080, por isso acessaremos <http://localhost:8080>. Quando o servidor roda na porta padrão do http (80) não é necessário citá-la, como no exemplo <http://www.google.com> que é equivalente a <http://www.google.com:80>.

Um servidor que suporta a Servlet API é chamado de Servlet Contêiner e ele pode conter diversos projetos ao mesmo tempo. No nosso caso, o projeto chamado *gerenciador* estará disponibilizado em <http://localhost:8080/gerenciador/index.html>.

Acessamos no navegador a URI <http://localhost:8080/gerenciador/index.html>. Perfeito! Nosso arquivo index.html está no diretório WebContent. Tudo que está dentro deste diretório pode ser acessado através da web em nossa aplicação, exceto dois diretórios: o META-INF e WEB-INF. No diretório WEB-INF/classes estarão as classes de nossa aplicação, enquanto que no WEB-INF/lib estarão as bibliotecas jar. Note que já existe uma biblioteca de exemplo nesse diretório, um serializador de/para xml e json, o XStream.

No nosso projeto, sempre que acessamos um diretório real (como /) o servidor procurará um arquivo chamado *index.jsp* ou *index.html*. Sendo assim ao invés de acessarmos a URI anterior podemos utilizar simplesmente: <http://localhost:8080/gerenciador/> ou ainda <http://localhost:8080/gerenciador>.

Isso acontece pois configuramos nossa aplicação web no arquivo **web.xml**, que nesse instante diz que devemos procurar esses dois tipos de arquivos:

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  id="WebApp_ID" version="3.0">
  <display-name>gerenciador</display-name>

  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>

```

O mesmo **web.xml** é usado para diversas outras configurações de um servlet contêiner.

Vamos criar nossa primeira Servlet. Para isso CTRL+N, New class, e criamos uma classe chamada *BuscaEmpresa* em *br.com.gerenciador.web*. Pedimos para que ela herde de *HttpServlet*:

```

public class BuscaEmpresa extends HttpServlet{

}

```

E qual será o método chamado quando acessarmos ela? Quando acessamos uma URI diretamente no navegador, estamos efetuando um GET de uma URI, por isso vamos sobrescrever o método *doGet*:

```

@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

}

```

Repare que o método recebe dois argumentos: a requisição - o request - e a resposta - o response. Queremos mostrar uma mensagem de sucesso na resposta. Sempre que desejamos escrever algo para o cliente desejamos usar a resposta. Para escrever o resultado que será recebido pela requisição http utilizamos o método *getWriter*:

```

PrintWriter writer = response.getWriter();
writer.println("<html>");
writer.println("<body>");
writer.println("Resultado da busca:<br/>");
writer.println("</body>");
writer.println("</html>");

```

Mas qual a URI que será acessada? Precisamos falar em algum lugar que ao acessar a URI `/busca`, desejamos executar esta Servlet. Para isso anotamos a Servlet:

```

@WebServlet(urlPatterns = "/busca")
public class BuscaEmpresa extends HttpServlet {
    // ...
}

```

Vamos testar? Paramos o servidor e iniciamos ele novamente. Acessamos agora:

<http://localhost:8080/gerenciador/busca>.

E o resultado é:



Mas no nosso caso desejamos listar as empresas similares a "doce", como "Doceria Extra" e "Ferragens Docente". Nosso projeto já possui uma classe de acesso aos dados de empresas, o *EmpresaDAO*:

```
Collection<Empresa> empresas = new EmpresaDAO().buscaPorSimilaridade("doce");
```

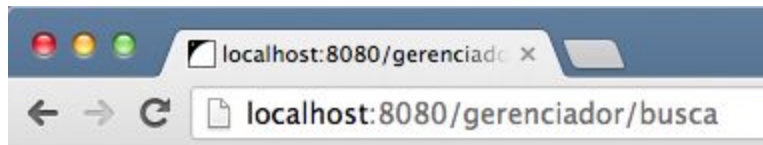
Gostaríamos de devolver para nosso cliente o nome e o id dela:

```

writer.println("<ul>");
for(Empresa empresa : empresas) {
    writer.println("<li>" + empresa.getId() + ": " + empresa.getNome() + "</li>");
}
writer.println("</ul>");

```

Restartamos nosso servidor e o código funciona! Note que não é necessário saber como funciona o acesso aos dados: esse é o papel do DAO. No nosso caso específico estamos simulando um banco de dados com variáveis estáticas. O resultado é a lista de empresas com "doce":



Resultado da busca:

- 1: Doceria Bela Vista
- 2: Ferragens Docel

Como último passo de introdução vamos alterar nossa Servlet para que como clientes sejamos capazes de dizer quais empresas queremos buscar, empresas com "doce" ou empresas com "dextra", por exemplo. Note que essa é uma informação que o cliente passará para o servidor, se é uma informação que o cliente *envia*, então ela vem no *request*. Portanto usamos o *request.getParameter*:

```
String filtro = req.getParameter("filtro");
```

```
Collection<Empresa> empresas = new EmpresaDAO().buscaPorSimilaridade(filtro);
```

Reinicializamos nosso servidor e acessamos agora a URI. Se não passamos nenhum parametro ela traz todas as empresas.

Se acessarmos com o parâmetro *filtro*, o mesmo é utilizado para mostrar somente as empresas que estamos interessados. Sucesso.



Resultado da busca:

- 1: Doceria Bela Vista
- 2: Ferragens Docel

Com o request somos capazes de ler informações de nossos clientes enquanto que com o response podemos escrever e devolver informações para ele. Escrever html dentro de uma servlet parece ser feio e é, veremos como melhorar e utilizar boas práticas de desenvolvimento web para remover esse código durante este curso.



# Aula 2

## Post e os métodos HTTP

Já vimos como tratar uma requisição que recebe parâmetros e executa uma busca, vamos agora adicionar uma nova empresa. O *EmpresaDAO* já possui o método *adiciona*, portanto basta instanciarmos uma empresa baseada nos dados que o usuário nos enviar, algo como:

```
String nome = req.getParameter("nome");
```

Agora que temos o valor para instanciar uma empresa, podemos executar o *new* e adicioná-la ao *dao*. Podemos também mostrar uma página de resultado que indica o sucesso:

```
Empresa empresa = new Empresa(nome);
new EmpresaDAO().adiciona(empresa);
PrintWriter writer = resp.getWriter();
writer.println("<html><body>Empresa " + nome + " adicionada!</body></html>");
```

Vamos criar a servlet *NovaEmpresa* e em seu método *doGet* implementamos o código mencionado:

```
String nome = req.getParameter("nome");
Empresa empresa = new Empresa(nome);
new EmpresaDAO().adiciona(empresa);
PrintWriter writer = resp.getWriter();
writer.println("<html><body>Empresa " + nome + " adicionada!</body></html>");
```

Agora podemos mapeá-la para a URI **/novaEmpresa**:

```
@WebServlet(urlPatterns = "/novaEmpresa")
public class NovaEmpresa extends HttpServlet {
// resto do código
}
```

Poderíamos acessar diretamente a URI

<http://localhost:8080/gerenciador/novaEmpresa?nome=NOME> mas faremos antes um formulário para facilitar nosso serviço.

O formulário HTML será adicionado na página **index.html** que está no diretório **WebContent**:

```
<html>
<body>Bem vindo ao nosso gerenciador de empresas!</body>
</html>
Começamos adicionando o formulário:
<body>Bem vindo ao nosso gerenciador de empresas!
<form action="novaEmpresa">
</form>
</body>
```

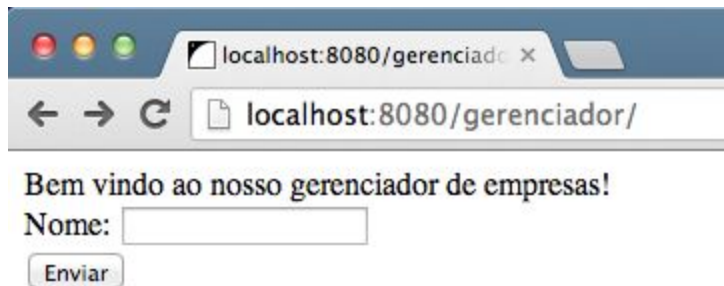
Colocamos agora um campo do tipo *text* chamado *nome*:

Nome: `<input type="text" name="nome" /><br/>`

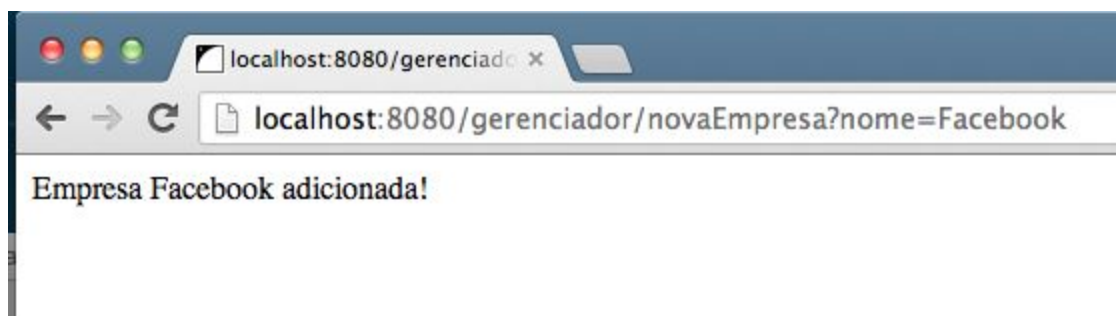
E um campo do tipo *submit* para o usuário enviar o formulário:

`<input type="submit" value="Enviar" />`

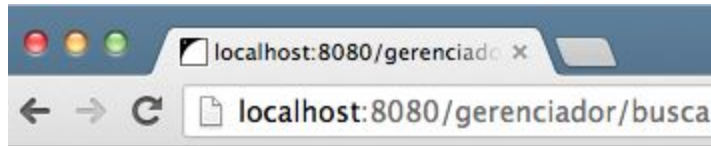
Vamos reinicializar o servidor e acessar a URI <http://localhost:8080/gerenciador/>:



Preenchemos os campos com o valor **Facebook**. Enviamos o formulário e temos sucesso, mas note a URI final:



O resultado pode ser conferido na página de busca:



Resultado da busca:

- 1: Doceria Bela Vista
- 2: Ferragens Docel
- 3: Alura
- 4: Google
- 5: Caelum
- 6: Casa do Código
- 7: Facebook

Ao utilizarmos um formulário HTML padrão, o navegador executa um GET para a URI que indicamos. Os parâmetros, como já vimos anteriormente, são enviados na própria URI. Um primeiro problema dessa abordagem envolve o limite de caracteres: proxies ou servidores podem limitar o número de bytes que uma URI utiliza e se o usuário preencher um formulário muito grande, os dados não chegarão do outro lado ou - pior ainda - chegarão pela metade. Outro problema está ligado com cache: todo navegador pode, por padrão, cachear o resultado de uma requisição GET. No nosso caso o que aconteceria se desejássemos cadastrar **outra** empresa chamada **Facebook**?

**Se** o navegador efetuar um cache de nossa página, ele não enviará a requisição ao servidor, mostrando para nós o resultado de sucesso - de nova empresa cadastrada - mas na verdade ele nem foi até o servidor. Mas isso só acontece **se** ele cachear. E na prática você não controla isso. Por isso não podemos depender do navegador para permitir que o usuário cadastre duas empresas com o mesmo nome.

Quando fazemos uma requisição HTTP para o servidor executamos um comando como:  
GET /gerenciador/novaEmpresa?nome=Facebook HTTP/1.1  
Host: localhost:8080

Ao invés de usarmos uma requisição do tipo GET, o protocolo HTTP disponibiliza um outro método, um que **por padrão** não permite cache do cliente, e permite um número ilimitado de dados enviados ao servidor. O método se chama POST e os parâmetros deixam de ser enviados via URI, e passam a ser enviados no **corpo** da requisição:

POST /gerenciador/novaEmpresa HTTP/1.1  
Host: localhost:8080

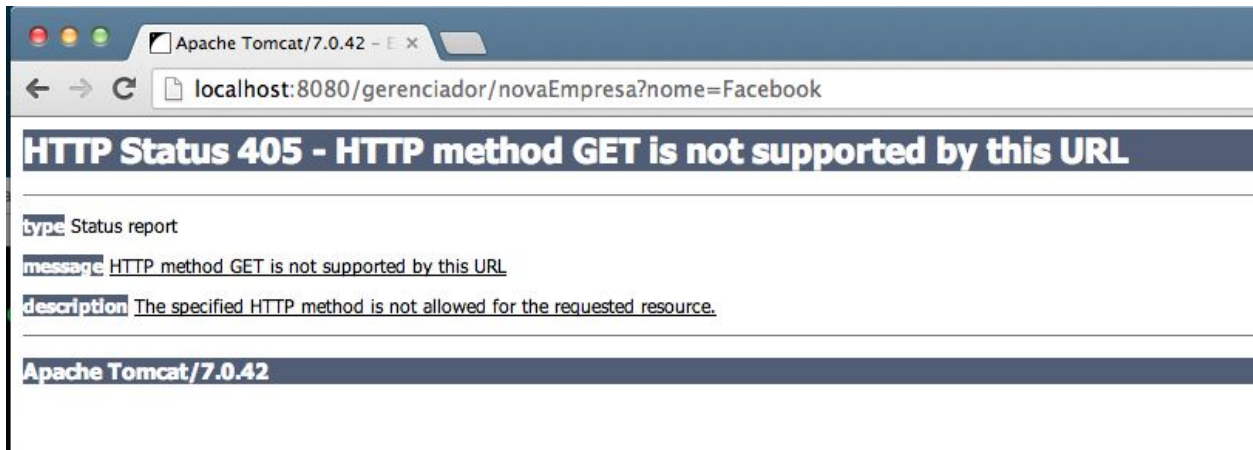
nome=Facebook

Como fazer para dar suporte ao *POST* no nosso servidor? Basta mudarmos o método *doGet* para *doPost*:

@Override

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    String nome = req.getParameter("nome");
    Empresa empresa = new Empresa(nome);
    new EmpresaDAO().adiciona(empresa);
    PrintWriter writer = resp.getWriter();
    writer.println("<html><body>Empresa " + nome + " adicionada!</body></html>");
}
```

Reinicializamos o servidor e testamos novamente, mas temos um erro:



Acontece que deixamos de suportar o *GET* quando paramos de sobrescrever tal método.


Faltou alterarmos o formulário para utilizar o método *POST*:

```
<form action="novaEmpresa" method="post">
  Nome: <input type="text" name="nome" /><br/>
  <input type="submit" value="Enviar" />
</form>
```

Não podemos esquecer de atualizar a tela e testamos novamente:



Podemos abrir a aba de desenvolvimento do navegador e refazer a requisição, notando que na aba de *Network* visualizamos o *POST*:

× Elements Resources Network Sources Timeline Profiles Audits Console							
Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency	
 novaEmpresa /gerenciador	POST	200 OK	text/h...	Other	158 B 55 B	3 ms 2 ms	

Claro que quando criamos um formulário de login ou que envolva algum dado sensível desejamos utilizar o *POST* e não o *GET* para que a informação não apareça na barra do navegador. Mas utilizar *POST* **não garante segurança!!!!** Qualquer programa no meio do caminho consegue visualizar os cabeçalhos enviados e entender o valor da requisição. Se o ponto é criptografar os dados enviados, é importante utilizar SSL e não só utilizar *POST*. Por trás do panos, o que a Servlet faz é dar suporte a diversos métodos como *doGet*, *doPost*, *doDelete*, *doTrace* mas na prática é importante saber que os **navegadores suportam apenas** os métodos *GET*, *POST* e *HEAD*. Se desejar utilizar outros métodos, deverá fazer isso via javascript ou através de outra aplicação cliente.

A Servlet suporta todos os métodos de uma vez só através de seu método *service*. Se deseja suportar todos, pode sobrescrevê-lo, mas tome cuidado quando fizer isso pois estará respondendo para todos da mesma maneira.

# Aula 3

## Características da web

Começamos então acessando uma página qualquer em nosso navegador, como o site <http://localhost:8080/ExemploServlets/>. Note que quando acessamos uma página estamos pedindo algo para o servidor. No caso o servidor pode ser encontrado em <http://localhost:8080/ExemploServlets> e a página é a /. O servidor retorna o resultado - uma página html - para nós.

Clicamos da direita em nosso navegador e indicamos que desejamos ver o código fonte de nossa página: o html é texto puro. Se o resultado de nossa requisição é um texto html, de onde vieram as imagens?

Dentro do html temos tags como *img*, que indica que deve ser colocada uma imagem aqui. Mas ele não tem o conteúdo da imagem, somente uma URI, um link para a imagem. Então da mesma maneira que o navegador fez uma requisição para <http://localhost:8080/ExemploServlets/> agora ele faz uma requisição para <http://localhost:8080/ExemploServlets/nome-da-imagem.jpg>.

Nessa segunda requisição, o servidor retorna uma imagem. No nosso navegador podemos ver isso acontecendo na barra de desenvolvedores: no menu, procuramos a opção de barra de desenvolvedores, ferramentas de desenvolvimento (Developer Tools). Deixamos aberta a parte de Network e executamos um refresh de nossa tela.

Repare a quantidade de idas e vindas que o navegador fez! Cada vez que acessamos uma página html, diversos outros recursos (imagens, javascripts etc) são carregados e, segundo o protocolo http que utilizamos, para cada um deles o cliente - nosso navegador - faz uma requisição e recebe uma resposta.

Vimos também que o HTTP suporta diversos tipos métodos em cada requisição. Algumas vantagens do GET estão ligadas com a possibilidade de cachear o resultado da requisição mesmo que o servidor não indique nada sobre cache para seus clientes. Isso faz com que proxies no meio do caminho armazenem o resultado e sirvam a mesma informação posteriormente.

Mas ele também tem um limite: a URI de requisição pode ter um limite de tamanho implementada tanto pelo cliente, quanto pelo servidor, como por proxies no meio do caminho -

e o cliente não sabe esse limite. Por isso não podemos arriscar e enviar possíveis formulários longos através de GET.

Além disso, o cache poderia fazer com que uma requisição não fosse executada de verdade, somente parecendo ter sido executada, o que pode ser um risco ao enviar um formulário. Por fim, o cabeçalho, onde a URI aparece, não suporta conteúdo binário por padrão - se você deseja fazer o upload de um conteúdo como uma imagem JPEG, terá que fazê-lo com outro método.

O método GET deve ser usado para buscar informações, para requisições que não trazem efeitos colaterais indesejáveis, para requisições que podem ser requisitadas diversas vezes sem causar nenhum dano.

O método POST é considerado um método que efetua uma alteração de estado no servidor - por isso o navegador sempre pergunta se você tem certeza do que faz ao atualizar uma página que fez POST. Ele tem um número ilimitado de bytes que podem ser enviados em seu corpo - incluindo conteúdo binário como imagens - e por padrão não é cacheado (mas pode ser caso o servidor deseje).

Por fim, existem alguns exemplos que demonstram casos interessantes do mal uso do GET. Uma biblioteca que implementasse a opção de imprimir um livro ou documento através de um link, ou um sistema que permita remover dados através de um link. Links são métodos GET, portanto um plugin do navegador pode desejar buscar o resultado desses métodos de antemão, cacheando localmente o resultado. Quando o usuário clicar no link, ele terá o resultado bem rápido.

O que acontece é que o plugin assume que método GET não causa efeito indesejado, é idempotente, pode ser feito quantas vezes quiser sem trazer nenhum problema - mas a implementação dessas empresas não foi feita dessa maneira.

# Aula 4

## Trabalhando com Filtros

Já sabemos criar nossas servlets, tratar parâmetros da requisição de nosso cliente e devolver respostas com código html.

Mas algumas tarefas queremos executar em mais de uma servlet, como por exemplo logar quais URIs estão sendo acessadas (auditoria), ou então um processo de autorização que verifica se o usuário atual pode acessar um recurso.

Quando existem preocupações ou trechos de código que queremos executar em diversas URIs ou quando diversas Servlets são executadas, podemos criar um filtro, algo que fica antes de um grupo (ou uma única) servlet ou página html etc.

Vamos criar um primeiro filtro de auditoria que loga todos os acessos ao sistema. Para isso criaremos uma nova classe chamada *FiltroDeAuditoria*, no pacote *web*. Ela deve implementar a interface *Filter*:

```
public class FiltroDeAuditoria implements Filter{  
}
```

O Eclipse passa a reclamar que a classe não implementou os métodos definidos na interface que implementamos. Com o *Ctrl+1* escolhemos para implementá-los. Temos um método de inicialização (*init*), um de destruição (*destroy*) e um que é executado toda vez que uma requisição passa por este filtro:

```
public class FiltroDeAuditoria implements Filter{  
  
    @Override  
    public void destroy() {  
    }  
  
    @Override  
    public void init(FilterConfig config) throws ServletException {  
    }  
}
```



```

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
    }
}

```

Vamos implementar então o método *doFilter* para que ele chame a servlet:

```

public void doFilter(ServletRequest request, ServletResponse response,
    FilterChain chain) throws IOException, ServletException {
    chain.doFilter(request, response);
}

```

Mas o que acontece se temos dois filtros que devem ser acessados ao utilizar uma URI? Bom, o método *doFilter* chamará o próximo filtro ou a servlet, ou renderizará a página. O filtro atual não sabe isso e não tem problema, ele simplesmente pede para o servlet contêiner: por favor execute o próximo passo.

Se existir um outro filtro nesta URI, ele será executado. Se não existirem mais filtros, a página será devolvida ou a servlet será executada.

Sabendo disso, antes de executarmos o *doFilter* e ir para o próximo filtro ou servlet/página, queremos logar a URI. Primeiro fazemos o cast para **HttpServletRequest** e depois executamos o método **getRequestURI**:

```

    HttpServletRequest req = (HttpServletRequest) request;
    System.out.println("Usuario acessando a URI " + req.getRequestURI());
    chain.doFilter(request, response);

```

Vamos agora mapear nosso filtro para ser utilizado para qualquer URI de nosso sistema. Assim como na classe que representa a Servlet mapeamos com a anotação *WebServlet*, no nosso filtro fazemos:

```

@WebFilter(urlPatterns=/*")
public class FiltroDeAuditoria implements Filter {
    // resto do código
}

```

Agora reinicializamos nosso servidor, acessamos a URI base */gerenciador* e vemos o log:

INFO: Reloading Context with name [/gerenciador] is completed

Usuario acessando a URI */gerenciador/*

Testamos a busca e adicionar uma nova empresa:

Usuario acessando a URI */gerenciador/*

Usuario acessando a URI */gerenciador/busca*

Usuario acessando a URI */gerenciador/busca*

Usuario acessando a URI */gerenciador/novaEmpresa*

O conceito de filtro é amplamente utilizado em aplicações web como uma maneira de adicionar características a parte da aplicação. Usando a API de servlets basta criar uma classe que implementa *Filter* e anotá-la com *WebFilter*. Em outras APIs é comum encontrar o nome de *Interceptor* ao invés de *Filter* para o mesmo conceito.

# Aula 5

## Trabalhando com Cookies

Vimos anteriormente como criar um filtro usando a api de Servlets para auditar todas as URIs que estão sendo acessadas. Desejamos adicionar agora um controle de qual usuário está acessando qual parte do sistema.

Para isso vamos criar uma nova servlet de autenticação. Autenticar é o processo no qual recebemos os dados do cliente e verificamos se ele é um usuário registrado, efetuando seu login.

Começamos lendo os parâmetros *email* e *senha* como já conhecemos:

```
String email = req.getParameter("email");  
String senha = req.getParameter("senha");
```

Usaremos o *UsuarioDAO* para *buscar por email e senha* passando esses dois campos:

```
Usuario usuario = new UsuarioDAO().buscaPorEmailESenha(email, senha);
```

E verificamos se o usuário não foi encontrado, mostrando uma mensagem de erro:

```
if (usuario == null) {  
    writer.println("<html><body>Usuário ou senha inválida</body></html>");  
}
```

Caso o usuário tenha sido encontrado com sucesso, mostraremos o email do usuário logado:

```
writer.println("<html><body>Usuário logado: " + email  
+ "</body></html>");
```

Ele entra com os dados dele através de um formulário de login que adicionaremos ao nosso

**index.html:**

```
<form action="login" method="post">  
    Email: <input type="text" name="email" /><br/>  
    Senha: <input type="password" name="senha" /><br/>  
    <input type="submit" value="Login" />  
</form>
```

Mas e agora? Como faremos para marcar este cliente, este navegador, dizendo que ele é o usuário **X**? Ele enviou os dados, mas lembre-se que na web via *http*, toda requisição nova é, por padrão, uma requisição solta - não existe nada que identifique a próxima requisição sendo do mesmo usuário/navegador.

Para resolver este problema uma solução é a utilização de cookies. Um cookie é um par de strings que é enviado do servidor para o cliente e fica lá por um tempo determinado. **Toda vez** que o cliente fizer uma requisição para o mesmo servidor, esse cookie será enviado de volta - e o servidor terá a opção de fazer algo com ele.

Para criar um cookie com o email do usuário que estamos querendo logar, podemos fazer:

```
Cookie cookie = new Cookie("usuario.logado", email);
```

E adicionamos o cookie na **resposta** uma vez que somos nós, servidor, enviando o cookie para o cliente:

```
Cookie cookie = new Cookie("usuario.logado", email);  
resp.addCookie(cookie);
```

Por fim, nossa servlet fica assim, mapeando a URI */login* e verificando se o usuário é inválido ou não:

```
@WebServlet(urlPatterns = "/login")  
public class Login extends HttpServlet {  
  
    @Override  
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)  
        throws ServletException, IOException {  
        String email = req.getParameter("email");  
        String senha = req.getParameter("senha");  
        Usuario usuario = new UsuarioDAO().buscaPorEmailESenha(email, senha);  
  
        PrintWriter writer = resp.getWriter();  
        if (usuario == null) {  
            writer.println("<html><body>Usuário ou senha inválida</body></html>");  
        } else {  
            Cookie cookie = new Cookie("usuario.logado", email);  
            resp.addCookie(cookie);  
            writer.println("<html><body>Usuário logado: " + email  
                + "</body></html>");  
        }  
    }  
}
```

```
}
```

Reinicializamos o servidor e testamos a URI da raiz de nossa aplicação, <http://localhost:8080/gerenciador/> :

Toda requisição terá seu cookie enviado de volta para o servidor **de origem**. Sabendo disso, vamos alterar nosso filtro para pegar o usuário atual, **se** ele existir. No nosso filtro, logo após fazer o cast de *HttpServletRequest*, pegamos todos os cookies **deste domínio**:

```
HttpServletRequest req = (HttpServletRequest) request;  
Cookie[] cookies = req.getCookies();
```

Agora fazemos um for para procurar o cookie que desejamos, aquele que tem o nome **usuario.logado** e extraímos seu valor, o email do usuário logado:

```
for (Cookie cookie : cookies) {  
    if(cookie.getName().equals("usuario.logado")) {  
        String usuario = cookie.getValue();  
    }  
}
```

Para melhorar nosso código, extraímos um método para buscar esse usuário. Caso ele encontre, devolve o cookie, caso contrário, retorna null:

```
private Cookie getUsuario(HttpServletRequest req) {  
    Cookie[] cookies = req.getCookies();  
    for (Cookie cookie : cookies) {  
        if (cookie.getName().equals("usuario.logado")) {  
            return cookie;  
        }  
    }  
    return null;  
}
```

No nosso método *doFilter*, basta chamar o método *getUsuario* e usar o resultado para nossa mensagem de log, conferindo se o usuário voltou ou não:

**@Override**

```
public void doFilter(ServletRequest request, ServletResponse response,  
    FilterChain chain) throws IOException, ServletException {  
    HttpServletRequest req = (HttpServletRequest) request;  
  
    Cookie cookie = getUsuario(req);  
    String usuario = "<deslogado>";  
    if (cookie != null)  
        usuario = cookie.getValue();  
    System.out.println("Usuario " + usuario + " acessando a URI "  
        + req.getRequestURI());  
    chain.doFilter(request, response);  
}
```

Reinicializamos o servidor e acessamos uma URI qualquer. Como ainda temos o cookie, nossa mensagem no log é:

**Usuario <fulano@email.com> acessando a URI /gerenciador/**

Servidores que implementam a especificação de Servlet API, o cookie vive enquanto o navegador está aberto. Feche seu navegador **completamente** e tente novamente. Temos um erro 500, uma exception que aconteceu. Na linha *req.getCookies* ocorreu um *NullPointerException*:

Para corrigir nosso código devemos verificar se existe algum cookie:

```
private Cookie getUsuario(HttpServletRequest req) {  
    Cookie[] cookies = req.getCookies();  
    if (cookies == null)  
        return null;  
    for (Cookie cookie : cookies) {  
        if (cookie.getName().equals("usuario.logado")) {  
            return cookie;  
        }  
    }  
    return null;  
}
```

Reinicializamos o servidor e tentamos novamente, agora o log indica:  
Usuario <deslogado> acessando a URI /gerenciador/

# Aula 6

## Manipulando cookies existentes

Como falamos anteriormente, o cookie adicionado por um servlet contêiner do Java é expirado por padrão quando o navegador (e não só uma aba ou janela dele) é fechado. Mas e se desejamos manter o cookie por mais tempo no cliente?

Nesse caso podemos configurar o tempo máximo de **segundos** através do método **setMaxAge**:

```
cookie.setMaxAge(60 * 10); // 10 * 60 segundos, são dez minutos
```

Mesmo que o navegador seja fechado, se aberto durante os próximos dez minutos, o cookie ainda estará lá. Note que nesse caso, cada novo acesso posterga o timeout do cookie em mais dez minutos. Portanto se ele acessa uma página as 3:00 e as 3:09 outra página, o cookie dele só expirará as 3:19.

Se desejamos remover um cookie nessa requisição, basta alterar o tempo dele para 0:

```
cookie.setMaxAge(0);
```

Em ambos os casos devemos escrever o cookie de volta no response:

```
cookie.setMaxAge(0);  
resp.addCookie(cookie);
```

Portanto podemos escrever nossa servlet de logout, que verifica a existência do cookie para removê-lo:

```
Cookie cookie = getUsuario(req);  
if(cookie!=null) {  
    cookie.setMaxAge(0);  
    resp.addCookie(cookie);  
}  
  
PrintWriter writer = resp.getWriter();  
writer.println("<html><body>Logout  
efetuado</body></html>");
```

E mapeá-la para **/logout**:

```
@WebServlet(urlPatterns = "/logout")  
public class Logout extends HttpServlet {
```



```
// código aqui  
}
```

Mas o método *getUsuario* não está disponível nesta servlet, e sim no **FiltroDeAuditoria**. Vamos simplificar nossa implementação criando uma classe chamada **Cookies**, que recebe os cookies:

```
public class Cookies {  
  
    private final Cookie[] cookies;  
  
    public Cookies(Cookie[] cookies) {  
        this.cookies = cookies;  
    }  
  
}
```

E possuí o método para extrair o usuário logado:

```
public Cookie getUsuarioLogado() {  
    if (cookies == null)  
        return null;  
    for (Cookie cookie : cookies) {  
        if (cookie.getName().equals("usuario.logado")) {  
            return cookie;  
        }  
    }  
}
```

```
return null;
```

```
}
```

Basta alterarmos agora nosso **FiltroDeAuditoria** e nosso **Logout** para usar tal método:

```
Cookie cookie = new
```

```
Cookies(req.getCookies()).getUsuarioLogado();
```

Vamos criar o botão de logout em nossa página **index.html**:

```
<form action="logout" method="post">
```

```
  <input type="submit" value="Logout" />
```

```
</form>
```

Reiniciamos o servidor, efetuamos nosso login como anteriormente e o cookie é mantido. Clicamos em logout, repare que agora o servidor respondeu com o cookie e o tempo máximo de vida **0**.

Como podemos ver, toda vez que, como servidores, desejamos alterar o estado de um cookie, devemos readicioná-lo ao cliente através de **response.addCookie**.

Mas todo esse tempo estamos vendo o conteúdo do cookie no nosso navegador. O que acontece se eu alterar ele? No console da aba de

desenvolvedores do Chrome ou de outro navegador podemos entrar com o seguinte código:

```
document.cookie="usuario.logado=zedasilva@teste.com.br";
```

Ao acessarmos a raiz de nossa aplicação vemos no log:

```
Usuario zedasilva acessando a URI /gerenciador/
```

O que aconteceu? Nós como desenvolvedores não temos controle sobre o conteúdo do cookie que o cliente nos enviará. Por esse motivo não podemos deixar exposto como deixamos um valor tão importante e fácil de ser hackeado como o identificador do usuário. O cookie é genial para marcar um cliente e acompanhá-lo por diversas páginas, mas para usá-lo como autenticação teremos que utilizar algo mais seguro.

Um outro cuidado importante é que se o cookie for muito grande, toda requisição terá que enviar e receber esses dados novamente: quanto maior seu cookie, mais lenta será sua requisição. Como nos proteger do ataque de segurança e ao mesmo tempo garantir que o tamanho dos cookies não cresça indefinidamente?

# Aula 7

## Mantendo informações no lado do servidor com session

Como vimos anteriormente, os cookies permitem que marquemos nossos clientes com determinadas informações. Mas tudo o que adicionamos em um cookie pode ser alterado pelo usuário final - um possível problema de segurança, e todos os dados são enviados pelo cliente ao servidor a cada nova requisição - o que pode ficar lento, portanto existem limites nos tamanhos dos cookies.

Antes de começar os próximos passos, efetuarei o logout para garantir que não tenho nenhum cookie de minha aplicação.

Para resolver esses problemas ao invés de deixarmos o identificador do usuário (no nosso caso, o email) no lado do cliente, podemos deixar ele do lado do servidor, e enviar um código aparentemente randômico para ele. Isto é, no login geramos um código e marcamos quem o usuário é:

```
static Map<String,String> USUARIOS_LOGADOS = new HashMap<>();
```

```
public String login(String usuario) {
```

```
    String codigo = System.currentTimeMillis() + ""; // exemplo  
    de geração de código fraco
```

```
    USUARIOS_LOGADOS.put(codigo, usuario);  
    return codigo;  
}
```

Enviamos para nosso cliente este **código**. Agora quando ele fizer uma requisição, saberemos baseado no código quem é que está logado:

```
Cookie cookie = getUsuarioLogado();  
String codigo = cookie.getValue();  
String usuario = USUARIOS_LOGADOS.get(codigo);
```

Com essa abordagem, todo tipo de informação pode ficar armazenada em nosso *HashMap* no servidor, evitando transferir muitos dados a cada nova requisição. Para um usuário se fazer passar por outro, ele tem que acertar o código gerado para o outro usuário, algo bem difícil de acontecer. Além disso, um *HashMap* permite que utilizemos qualquer tipo de objeto como valor: não estamos mais restritos a usar somente strings, uma vez que os objetos ficam o tempo todo no servidor: eles não vão de um lado para o outro!

Note que a chave do *HashMap* continua sendo uma string, mas o valor poderia ser o próprio *Usuario*!

Vamos adaptar nosso código para deixar de usar cookies e passar a usar esse *HashMap*. Na verdade, o Java já implementa essa alternativa para nós com o nome de *Session* e suporta até mesmo um pouco mais!

Para acessar a *Session* de um cliente, pegamos:

```
HttpSession session = req.getSession();
```

Caso ela ainda não existisse (nenhum cookie com código gerado exista no cliente), uma session será criada agora e conectada a este cliente. Agora para logar nosso usuário na sessão devemos usar o método *setAttribute*. Como uma única sessão de um usuário pode armazenar diversos valores, passamos uma chave e um valor:

```
session.setAttribute("usuario.logado", usuario);
```

Portanto nosso código do *else* da servlet de **Login**:

```
HttpSession session = req.getSession();  
session.setAttribute("usuario.logado", usuario);  
writer.println("<html><body>Usuário logado: " +  
email  
+ "</body></html>");
```

O próximo passo é efetuar o **Logout**. Podemos remover o atributo com **removeAttribute** ou simplesmente invalidar a sessão atual:

```
session.removeAttribute("usuario.logado");  
session.invalidate();
```

No nosso caso usaremos o **removeAttribute** para que nenhum outro valor dessa mesma sessão seja anulado caso o usuário se deslogue. Nosso código de **Logout** que verificava o cookie, passa a ser mais simples:

```
HttpSession session = req.getSession();  
session.removeAttribute("usuario.logado");  
PrintWriter writer = resp.getWriter();  
writer.println("<html><body>Logout  
efetuado</body></html>");
```

Por fim, em nosso **FiltroDeAuditoria** queremos pegar o valor do atributo *usuario.logado*. Como a *Session* armazena qualquer tipo de objeto, teremos que fazer o cast para *Usuario*:

```
HttpSession session = req.getSession();  
Usuario usuarioLogado = (Usuario)  
session.getAttribute("usuario.logado");
```

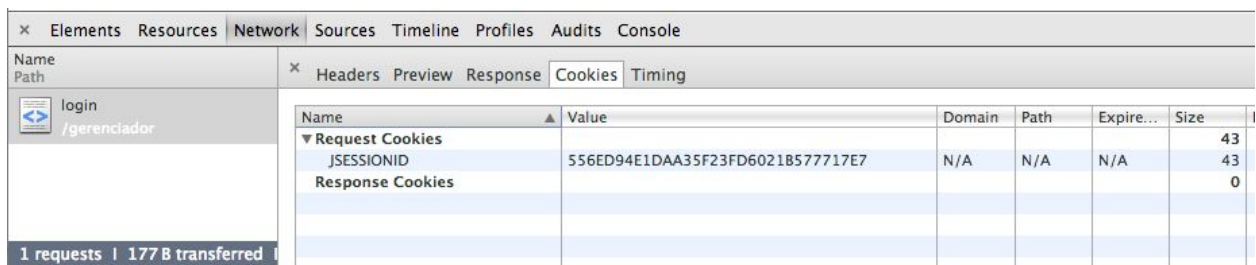
Mas é importante lembrar que a variável pode referenciar *null* caso o usuário não esteja logado e não seja encontrada essa chave (*usuario.logado*) dentro de sua session:

```
String usuario = "<deslogado>";  
  
if (usuarioLogado != null)  
    usuario = usuarioLogado.getEmail();
```

Vamos reiniciar o servidor e verificar o processo de login. Acessamos a raiz e no log temos:

Usuario <deslogado> acessando a URI /gerenciador/

Abro a aba *Network* do navegador e faço o login, repare o cookie que indica nossa sessão:



The screenshot shows the Network tab of a web browser. The left pane shows a request to 'login /gerenciador'. The right pane shows the 'Cookies' tab for this request. It lists 'Request Cookies' and 'Response Cookies'. The 'Request Cookies' section shows a single cookie: 'JSESSIONID' with value '556ED94E1DAA35F23FD6021B577717E7'. The 'Response Cookies' section is empty.

Name	Value	Domain	Path	Expire...	Size
<b>Request Cookies</b>					
JSESSIONID	556ED94E1DAA35F23FD6021B577717E7	N/A	N/A	N/A	43
<b>Response Cookies</b>					
					0

O servlet contêiner usa um cookie chamado **JSESSIONID** para indicar o código do usuário atual. No servidor ele armazena em um tipo de *HashMap*



qual o código e quais os dados atrelados ao usuário. Executo agora uma busca:

```
Usuario zedasilva@teste.com.br acessando a URI  
/gerenciador/busca
```

Como vimos, é muito mais simples usar a *Session* do que cookies e em muitos casos é ela a solução de nossos problemas de identificação de usuário. Mas de qualquer maneira devemos sempre tentar minimizar as informações que estão na *Session*: quanto mais informação nela, mais dados em memória que não podem ser jogados fora até o logout do usuário. Tome muito cuidado com isso. Coloque o mínimo possível de informações na *Session*.

# Aula 8

## Mais sobre servlets

Sabemos que a Servlet é uma classe Java, mas como e quando será que ela é instanciada? Vamos fazer alguns testes para entender melhor o funcionamento de uma servlet.

Podemos adicionar uma mensagem do tipo *System.out* na construção da Servlet, fazendo:

```
public BuscaEmpresa() {  
    System.out.println("Instanciando uma Servlet do tipo  
BuscaEmpresa " + this);  
}
```

Repare que temos dois métodos na classe *HttpServlet* que indicam a inicialização e destruição da mesma. Adicionaremos *System.out* também nos dois métodos, *init* e *destroy* respectivamente:

```
@Override  
public void init() throws ServletException {  
    System.out.println("Inicializando a Servlet " + this);  
}
```

```
}
```

```
@Override
```

```
public void destroy() {
```

```
    System.out.println("Destruindo a Servlet " + this);
```

```
}
```

Mas sempre que sobrescrevemos um método é importante lembrarmos se desejamos chamar o método na classe pai. Como o método de inicialização e destruição de uma servlet podem fazer algo, devemos chamar o *super*:

```
@Override
```

```
public void init() throws ServletException {
```

```
    super.init();
```

```
    System.out.println("Inicializando a Servlet " + this);
```

```
}
```

```
@Override
```

```
public void destroy() {
```

```
    super.destroy();
```

```
    System.out.println("Destruindo a Servlet " + this);
```

```
}
```

Reinicializamos o servidor e acessamos a URI de busca diversas vezes. O resultado no console do Eclipse é:

```
Oct 02, 2013 3:15:05 PM org.apache.catalina.startup.Catalina
start
INFO: Server startup in 1473 ms
Instanciando uma Servlet do tipo BuscaEmpresa
br.com.javaweb.gerenciador.web.BuscaEmpresa@552d4a00
Inicializando a Servlet
br.com.javaweb.gerenciador.web.BuscaEmpresa@552d4a00
```

Note que a servlet foi instanciada uma única vez, e o mesmo objeto foi inicializado também logo após a criação, nunca mais. A destruição ainda não foi executada, uma vez que não desligamos nosso servidor.

Para testar a destruição e o método *destroy*, apertamos o botão *stop* da aba de Servidor. O resultado é a destruição da única servlet existente:

```
INFO: Server startup in 1473 ms
Instanciando uma Servlet do tipo BuscaEmpresa
br.com.javaweb.gerenciador.web.BuscaEmpresa@552d4a00
Inicializando a Servlet
br.com.javaweb.gerenciador.web.BuscaEmpresa@552d4a00
Oct 02, 2013 3:15:29 PM org.apache.catalina.core.StandardServer
await
```

```
INFO: A valid shutdown command was received via the shutdown  
port. Stopping the Server instance.
```

```
Oct 02, 2013 3:15:29 PM org.apache.coyote.AbstractProtocol  
pause
```

```
INFO: Pausing ProtocolHandler ["http-bio-8080"]
```

```
Oct 02, 2013 3:15:29 PM org.apache.coyote.AbstractProtocol  
pause
```

```
INFO: Pausing ProtocolHandler ["ajp-bio-8009"]
```

```
Oct 02, 2013 3:15:29 PM
```

```
org.apache.catalina.core.StandardService stopInternal
```

```
INFO: Stopping service Catalina
```

```
Destruindo a Servlet
```

```
br.com.javaweb.gerenciador.web.BuscaEmpresa@552d4a00
```

Qual a importância de não criar uma nova servlet a cada requisição do cliente? Na época de seu lançamento, a capacidade da servlet de reutilizar uma única instância de objeto, tendo seu método chamado diversas vezes - inclusive em paralelo - se revelava como uma vantagem de desempenho.

E qual a desvantagem dessa abordagem? O ponto mais importante a tomar cuidado envolve múltiplas requisições. Imagine que definimos uma variável membro, como a *String filtro*, que armazena o valor do filtro atual.

```
String filtro;
```

```

@Override
protected void doGet(HttpServletRequest req,
HttpServletRequest resp)
    throws ServletException, IOException {
    PrintWriter writer = resp.getWriter();
    writer.println("<html>");
    writer.println("<body>");
    writer.println("Resultado da busca:<br/>");

    filtro = req.getParameter("filtro");
    Collection<Empresa> empresas = new
EmpresaDAO().buscaPorSimilaridade(filtro);
    // resto do código
}

```

O que aconteceria se dois usuários acessarem esse mesmo objeto ao mesmo tempo? Existe a possibilidade de duas threads rodarem esse código simultaneamente, como só existe uma variável *filtro* (variável membro), então corremos o risco das duas requisições se misturarem: não existem duas variáveis, somente uma!

Como podemos simular isso? Basta adicionar um código que deixe nossa busca lenta o suficiente para que as duas requisições sejam executadas ao mesmo tempo. Para isso adicionamos um *Thread.sleep* que dormirá 10 segundos:

```
filtro = req.getParameter("filtro");  
  
try {  
    Thread.sleep(10000);  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}  
  
Collection<Empresa> empresas = new  
EmpresaDAO().buscaPorSimilaridade(filtro);
```

Agora abrimos duas abas, uma buscando "Dextra" e a outra buscando "Doce". Como somente uma variável membro existe e o programa ficou esperando tempo suficiente para a segunda requisição alterar o valor da variável, as duas buscas mostram o mesmo resultado, o resultado do "Doce"!

Por isso é extremamente importante só utilizar variáveis membro em uma servlet se você for cuidar do acesso paralelo. Em geral não usamos e evitamos ao máximo tais variáveis em servlets.

Removemos então a variável membro filtro e voltamos a utilizá-la como variável local.

## Aula 9

### Redirecionamento no client e server side

Note que até agora para cada uma das lógicas de nosso sistema, tivemos que criar o código que executa a lógica de negócios e a parte que renderiza uma página html. Mas isso significa que misturamos lógica (Java) com visualização (html). Não seria possível separá-las?

Por exemplo, no caso do **Logout**, ao invés de removermos o atributo da **HttpSession** e escrevermos na saída do usuário, poderíamos somente executar a lógica de remoção:



```
HttpSession session = req.getSession();  
session.removeAttribute("usuario.logado");
```

E depois dizer para o **response** que ele deve notificar o cliente para que ele seja **redirecionado**. Isto é, enviamos uma resposta que manda ele redirecionar para outro lugar, para nossa página **logout.html**:

```
HttpSession session = req.getSession();  
session.removeAttribute("usuario.logado");  
resp.sendRedirect("logout.html");
```

Precisamos criar a página **logout.html** no diretório **Web-Content**. Lembrando que este diretório é a pasta raiz de nossa aplicação web:

```
<html><body>Logout efetuado</body></html>
```

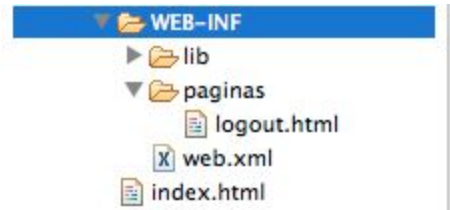
Reinicializamos o servidor, nos logamos e testamos o logout, perfeito, fomos deslogados, mas repare a URI, o cliente redirecionou para **/logout.html**:



Se clicarmos no botão de stop da aba Network e repetir **todo** o processo do zero, veremos que a resposta da página **/logout** é um 302 para a página **/logout.html**. Por isso o cliente redireciona para essa página. O código 302 significa que o recurso foi encontrado, mas o usuário deve encontrá-lo em outro lugar, por isso o navegador segue essa indicação.

Um resultado indesejado seja que ao efetuarmos um reload da página, uma atualização, um F5, um refresh, o usuário acessará a página **/logout.html**, sem passar pela lógica de logout. Isso não faz sentido! Queremos que o cliente continue na página **/logout** pois se ele fizer refresh é para executar a lógica novamente. Outro ponto é que não queremos que o usuário possa acessar a página **/logout.html** a qualquer instante. Imagina que um usuário qualquer digita <http://localhost:8080/gerenciador/logout.html>, isso não deveria funcionar.

Primeiro vamos esconder nossa página. Sabemos que os arquivos dentro de **WEB-INF** ficam escondidos, portanto criamos um diretório chamado **WEB-INF/paginas**. Movemos nosso **logout.html** para lá:



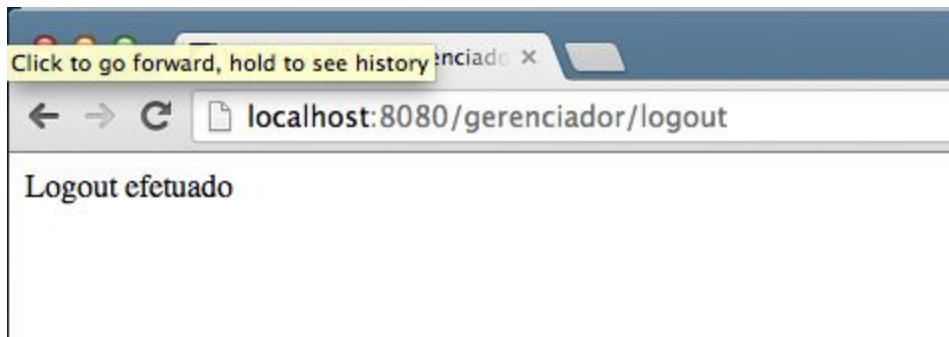
Agora alteramos nosso código de lógica para ao invés de notificar o cliente, fazer um forward no lado do servidor. Pegamos um rapaz responsável por redirecionar a requisição e a resposta, um *request dispatcher*:

```
RequestDispatcher dispatcher =  
req.getRequestDispatcher("/WEB-INF/paginas/logout.html");
```

E agora efetuamos o forward:

```
RequestDispatcher dispatcher =  
req.getRequestDispatcher("/WEB-INF/paginas/logout.html");  
dispatcher.forward(req, resp);
```

Reinicializamos nosso servidor e efetuamos todo o processo novamente. Repare como o cliente não ficou sabendo do redirecionamento:



Existem portanto dois tipos de redirecionamento. O primeiro é feito no lado do cliente, através de um retorno de código diferente de 200 (ok), como o 302 (found) - que notifica o usuário final onde deve procurar o resultado de sua requisição. Outra maneira de redirecionar é fazer um *server side redirect*, onde o cliente nem fica sabendo o que aconteceu. Se ele fizer um refresh da página, passará pela lógica de negócios novamente.

Aula10

## Páginas dinâmicas

Agora que já sabemos isolar o conteúdo fixo de nossas lógicas de negócio, vamos tentar isolar o conteúdo dinâmico, que envolve tanto tags html quanto variáveis como o nome da empresa que acaba de ser cadastrada:

```
PrintWriter writer = resp.getWriter();  
writer.println("<html><body>Empresa " + nome + "  
adicionada!</body></html>");
```

Nesse caso poderíamos redirecionar para uma página como */WEB-INF/paginas/novaEmpresa.html*, mas como mostraríamos o nome da nova empresa?

```
<html><body>Empresa QUERO_O_NOME_DA_EMPRESA_AQUI  
adicionada!</body></html>
```

Para isso vamos usar uma tecnologia que suporta receber variáveis e utilizá-las no meio de nossa página. A tecnologia padrão do Java é o JSP - Java Server Pages - e usaremos uma expressão bem simples para mostrar o nome:

```
<html><body>Empresa ${nome} adicionada!</body></html>
```

Basta utilizar o cifrão e as chaves para iniciar nossa expressão. Colocando o valor da variável, teremos ela impressa. Mas como faremos o

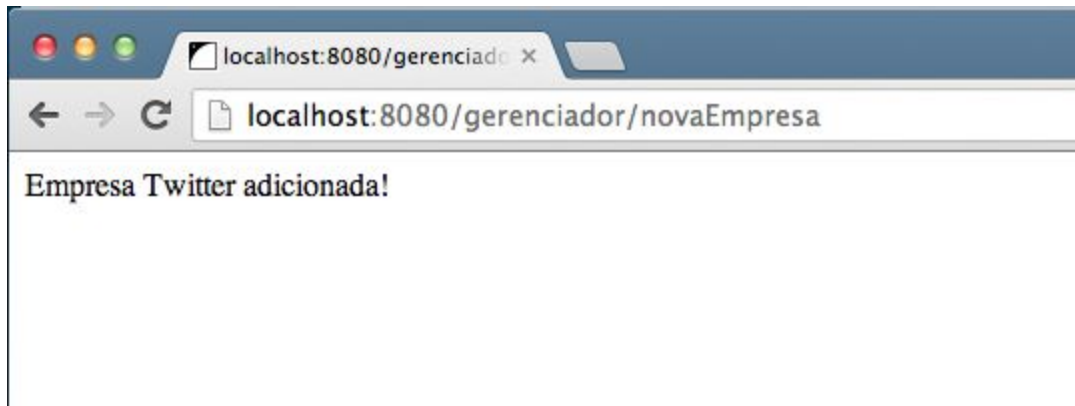
redirecionamento em nossa servlet? Chamaremos nossa página de **novaEmpresa.html**, mas como passar a variável?

```
String nome = req.getParameter("nome");  
Empresa empresa = new Empresa(nome);  
new EmpresaDAO().adiciona(empresa);  
  
req.getRequestDispatcher("/WEB-INF/paginas/NovaEmpresa.jsp").forward(req, resp);
```

Da mesma maneira que a *HttpSession* possui métodos para armazenar valores que duram durante toda a vida útil de uma sessão, a *HttpServletRequest* permite o armazenamento de valores que duram somente o ciclo de uma requisição - uma ida ao servidor e volta ao cliente, incluindo redirecionamentos do tipo server side, que estamos usando. Assim como no *HttpSession*, o método é o *setAttribute*:

```
req.setAttribute("nome", nome);
```

Reiniciamos nosso servidor e testamos o processo de adicionar uma nova empresa. Repare que como usamos o redirecionamento no lado do servidor, vemos somente a URI que acessamos:



## Trabalhando com Taglibs

Da mesma maneira que fizemos com a nova empresa faremos com a busca de empresa, passando as empresas:

```
String filtro = req.getParameter("filtro");  
Collection<Empresa> empresas = new  
EmpresaDAO().buscaPorSimilaridade(filtro);  
req.setAttribute("empresas", empresas);  
  
req.getRequestDispatcher("/WEB-INF/paginas/buscaEmpresa.jsp").  
forward(req, resp);
```

Mas e na view, em nosso jsp, como podemos fazer um laço **for** por todas as empresas?

```
<html><body>  
Resultado da busca:<br/>  
<ul>  
<!-- EU QUERO UM FOR AQUI -->  
${empresas }  
<!-- ATE AQUI -->  
</ul>  
</body></html>
```

Vamos pedir ajuda para uma biblioteca de tags (uma *taglibrary* ou *taglib*) que permite a execução de laços junto com as nossas expressões



(*expression language*). Para isso, executaremos um *forEach* em todas as *empresas*, chamando a variável de *empresa*:

```
<c:forEach var="empresa" items="${empresas }">  
    AQUI EU TENHO A EMPRESA  
</c:forEach>
```

Para cada empresa, imprimimos uma tag *li* com seu id e nome:

```
<c:forEach var="empresa" items="${empresas }">  
    <li>${empresa.getId() }: ${empresa.getNome() }</li>  
</c:forEach>
```

Podemos simplificar nosso código ainda mais. A *expression language* do JSP permite que utilizemos somente *id* e *nome* quando desejamos chamar getters:

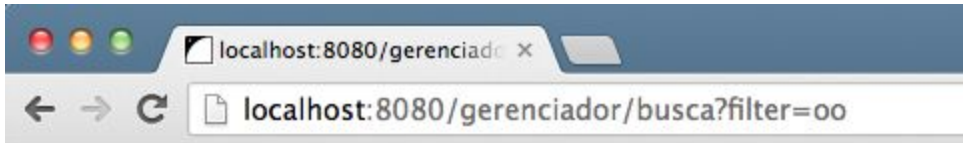
```
<c:forEach var="empresa" items="${empresas }">  
    <li>${empresa.id }: ${empresa.nome }</li>  
</c:forEach>
```

Assim como em código Java, devemos importar as bibliotecas que utilizamos. A linha de importar uma taglib costuma ficar no começo do

arquivo jsp, e no caso da taglib que disponibiliza a tag *forEach*, seu nome é *core*:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"
%>
<html><body>
Resultado da busca:<br/>
<ul>
<c:forEach var="empresa" items="{empresas }">
  <li>${empresa.id }: ${empresa.nome }</li>
</c:forEach>
</ul>
</body></html>
```

Reiniciamos o servidor e testamos a busca com sucesso:



Resultado da busca:

- 1: Doceria Bela Vista
- 2: Ferragens Docel
- 3: Alura
- 4: Google
- 5: Caelum
- 6: Casa do Código

Sempre que adicionamos uma taglib é importante ter os jars equivalentes em seu diretório **WEB-INF/lib**. No nosso caso já entregamos o projeto com a implementação da taglib core: o **jstl-1.2.jar**.

Assim como a expression language é capaz de buscar variáveis no escopo de *request*, ela pode buscar também no escopo de sessão. Caso ela não encontre a variável no *request*, ela procura na *sessão*. Sabendo disso podemos alterar nosso **index.html** para **index.jsp** e mostrar o email do usuário logado, caso ele exista:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"
%>
<html>
<body>
```

```
Bem vindo ao nosso gerenciador de empresas!<br/>
<br/>
<c:if test="${usuario.logado}">
    Você está logado como ${usuario.logado}<br/>
</c:if>
<!-- formularios aqui-->
</body>
</html>
```

Mas note que `${usuario.logado}` significa invocar o método equivalente a *getLogado* ou *isLogado* em uma variável chamada usuário. Não é isso que queremos, mas o nome que escolhemos para nossa chave foi ruim. Na prática devemos escolher nomes que evitam caracteres especiais como pontos. Vamos então renomear para **usuarioLogado**:

```
<c:if test="${usuarioLogado!=null}">
    Você está logado como ${usuarioLogado.email}<br/>
</c:if>
```

Alteramos nossas classes **Login** e **Logout** de acordo:

```
session.setAttribute("usuarioLogado", usuario);
```

E

```
session.removeAttribute("usuarioLogado");
```

Por fim, alteramos nosso filtro para usar essa variável:

```
Usuario usuario = (Usuario)  
session.getAttribute("usuarioLogado");
```

Reiniciamos o servidor, nos logamos e visualizamos a página inicial:



