



Desenvolvimento de Sistemas

Memória da aplicação *web*

Sistemas *web* são, por natureza, sem estado (*stateless*), ou seja, não guardam uma memória, como uma variável, um vetor ou outras estruturas comuns na programação *desktop*, que mantenham seus valores entre requisições diferentes. Cada requisição é tratada como uma ação única, e todas as informações necessárias devem ser repassadas do *front-end* para o *back-end*.

Há recursos, no entanto, que permitem guardar dados de um usuário e reutilizá-los nas páginas dos sistemas ou até compartilhá-los com outros sistemas *web*. No conteúdo a seguir, serão tratados os dois recursos mais importantes para isso: os **Cookies** e as **Sessões**.

Armazenamento de dados entre requisições

O recurso de armazenamento temporário de informações do usuário de um *website* ou sistema *web* são os famosos **Cookies**. Eles guardam informações no lado do cliente (no navegador) e reenviam essas informações de maneira automática ao servidor.

Hoje, *cookies* são muito usados para identificar ações de um usuário e o comportamento deles em um ou mais *sites*. Veja alguns exemplos de uso de *cookies*.

- ◆ Configurar a preferência de linguagem do usuário
- ◆ Lembrar de itens de um carrinho de compras
- ◆ Autenticar um usuário
- ◆ Mostrar propagandas com base no comportamento do usuário
- ◆ Monitorar como o usuário interage com as propagandas
- ◆ Preencher automaticamente campos em formulários

Há grande discussão quanto ao uso de *cookies* e a privacidade dos usuários de internet, tanto que se tornou padrão incluir um consentimento de uso desse recurso nas páginas (o usuário informa se quer continuar usando *cookies*, se não deseja usar ou se deseja selecionar para quais usos podem ser aplicados).



The screenshot shows the BBC Brasil website. At the top is a red header with the 'BBC NEWS BRASIL' logo. Below it is a navigation bar with links: Notícias, Brasil, Internacional, Economia, Saúde, Ciência, Tecnologia, and Vídeos. The main content area features a large image of a crowd holding a blue sign that reads 'Ministério do Trabalho e Emprego' and 'CARTEIRA DE TRABALHO PREVIDÊNCIA SOCIAL'. To the right of the image is the article title 'CLT faz 80 anos: o que mudou e pode mudar no Brasil da informalidade e aplicativos', followed by a short summary and the date '1 maio 2023'. At the bottom of the page is a dark grey cookie consent banner with the text 'Diga-nos se concorda com o uso de cookies', a brief explanation of cookies, and two buttons: 'Sim, concordo' and 'Não concordo, volte para Configurações'.

Figura 1 – Consentimento de uso de *cookies* no *site* da BBC Brasil

Fonte: <https://www.bbc.com/portuguese>

Tecnicamente, *cookies* são pequenos arquivos de texto contendo informações únicas que identificam o computador ou o usuário de um *site*. Esses arquivos são armazenados pelo navegador e são criados quando se acessa pela primeira vez um *site* que use *cookies*. Nas vezes posteriores em que se acessa esse mesmo *site*, os dados de *cookie* serão transmitidos e o *site* identificará o usuário de alguma maneira, usará esses dados para alguma personalização no *website* ou registrará em banco de dados alguma informação relativa.

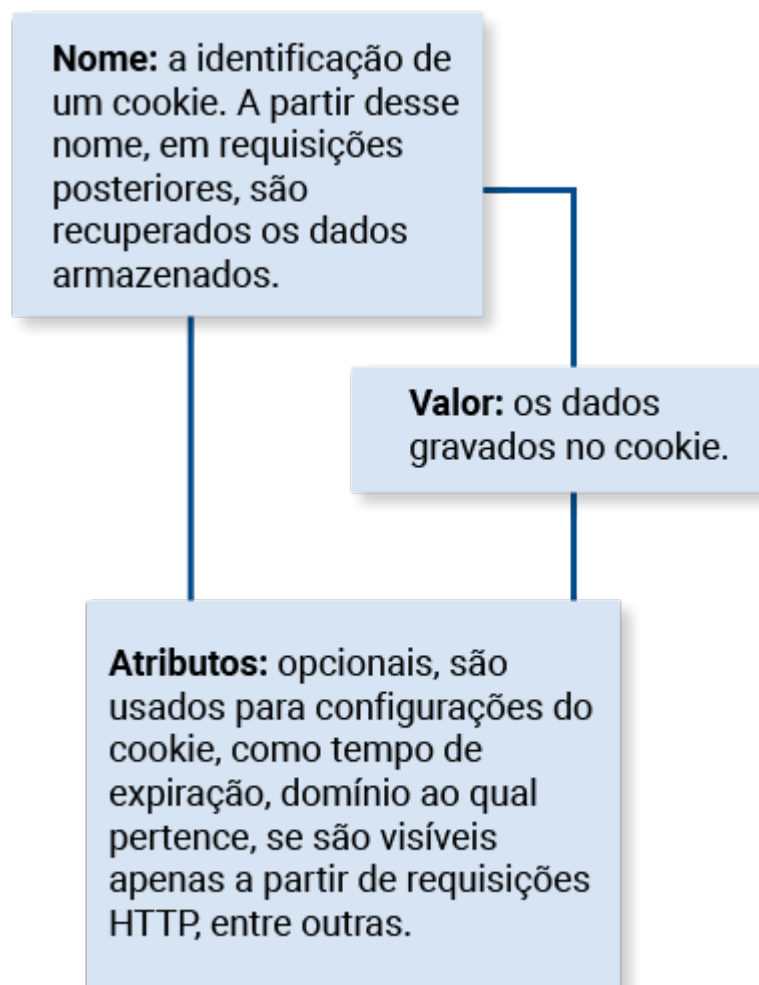
Figura 2 – Funcionamento de *cookies*

Fonte: Rai (2019)

Os *cookies* são criados no lado do servidor e precisam ser incluídos na resposta (*response*) enviada do servidor ao cliente. Quando houver uma requisição para uma página que lê *cookies*, o navegador enviará nos dados da requisição (*request*) as informações de *cookies* presentes.


Você pode verificar os *cookies* escritos por uma página a partir da propriedade *set-cookie* do response e os lidos por uma página a partir de *cookies* do *request*. Eles podem ser vistos em ferramentas de desenvolvimento, como o Chrome Devtools.

A estrutura de um *cookie* consiste de três elementos:



(imgs/ativo5.png)

Ao criar *cookies*, pode-se informar alguns **atributos** que alteram seu funcionamento. Um cookie pode ter limitado o seu tempo de vida, ou seja, pode-se informar quando um *cookie* expira e é excluído do navegador usando o atributo **Expires**. É possível restringir se o *cookie* será entregue apenas em conexões seguras (usando HTTPS) por meio do atributo **Secure**. Também pode-se definir se ele será invisível ao Javascript (atributo **HttpOnly**). Veja um exemplo de *cookie* definindo expiração e segurança.



```
Set-Cookie: id-cookie=meu-id-123; Expires=Thu, 21 Oct 2021 07:28:00 GMT; Secure; HttpOnly
```

Outros atributos normalmente usados são **Domain**, para definir que o *cookie* só será enviado em determinado domínio e subdomínios de internet, e **Path**, que implica que os cookies só existam em um caminho específico no *website* (por exemplo, apenas em páginas que estão em “/docs”).

Por serem recursos de navegador (lado cliente), o controle sobre os *cookies* está nas mãos do usuário, ele poderá decidir se os *sites* devem gravar ou não essas informações e poderá excluí-las quando quiser. Assim, trata-se de um recurso que não deve ser usado com finalidades vitais ao funcionamento de um sistema *web*, pois não há garantia de que a informação estará lá quando uma nova requisição ao sistema acontecer (o usuário pode ter apagado).

Um uso muito comum, no entanto, é para manter a autenticação de um usuário entre várias páginas. Lembra-se de como seria possível fazer isso em um sistema *desktop*? Basicamente ao realizar *login*, se guardava uma variável, o valor da autenticação e os dados do usuário e, ao abrir novas telas, eram consultadas essas variáveis para verificar quem está usando o sistema, suas permissões etc. Em sistemas *web*, como dito anteriormente, não há variáveis que persistam entre requisições. No entanto, pode-se usar o conceito de sessões para manter esses dados, e as **sessões** podem usar *cookies*.

Cookies

A criação e o uso de *cookies* em Java dependem de algumas classes específicas. A primeira é a classe **Cookie** (`jakarta.servlet.http.Cookie`), responsável pela criação de um *cookie*, permitindo configurações, como tempo de expiração e domínio. Outra classe fundamental é **HttpServletResponse** (`jakarta.servlet.http.HttpServletResponse`), que envia o *cookie* como resposta à requisição do usuário para que ele seja gravado no navegador. Veja um exemplo genérico de criação de *cookie*.

```
Cookie novoCookie = new Cookie("nome", "valor");
novoCookie.setMaxAge(3600);
novoCookie.setDomain("localhost");
novoCookie.setPath("/");
novoCookie.setHttpOnly(true);
novoCookie.setSecure(true);
```

```
//HttpServletResponse response  
response.addCookie(novoCookie);
```

Para a leitura de *cookies*, pode ser usada a classe `HttpServletRequest` (`jakarta.servlet.http.HttpServletRequest`) e usar o método `getCookies()`, que traz todos os *cookies* enviados pela requisição. É necessário percorrer uma lista se quiser encontrar um *cookie* específico. Veja a sintaxe a seguir, simulando o método que retorna um *cookie* específico.

```
Cookie[] cookies = request.getCookies();  
for (Cookie cookie : cookies ) {  
    if (cookie.getName().equals("nome")) {  
        return cookie;  
    }  
}
```

Porém, pelo *framework* Spring, há uma maneira mais simples quando é necessário obter valor de um *cookie* específico, que é uma anotação **@CookieValue** (`org.springframework.web.bind.annotation.CookieValue`) aplicada a parâmetros de métodos. Veja um exemplo de sintaxe.

```
@CookieValue(name="nome", defaultValue="", required=true) String valorCookie
```

Propriedades do Cookie

Alguns métodos da classe `Cookie` podem ser usados para configurar as propriedades do cookie na criação dele. São os seguintes:

Clique ou toque para visualizar o conteúdo.

setDomain(String):

Atributo Domain, que recebe por texto a qual domínio de internet o *cookie* ficará restrito.

setHttpOnly(boolean):

Atributo HttpOnly, que define (*true* ou *false*) se o *cookie* deve ser restrito a requisições diretas de HTTP (via navegador).

setMaxAge(int):

Define o atributo Max-Age, um tempo de expiração para o *cookie* expresso em segundos (o *cookie* será excluído após esse tempo).

setPath(String):

Atributo Path, que define em qual caminho de URL o *cookie* deve ser usado.

setSecure(boolean):

Atributo Secure, que restringe o *cookie* a requisições com HTTPS.

Cada propriedade tem seus métodos getter correspondentes (ou “is” em `isHttpOnly()`).

Agora é hora de praticar com exemplos em projeto Spring MVC e, em seguida, um projeto RESTful.

Cookies com Spring MVC

Para o primeiro teste, crie a partir do Spring Initializr (<https://start.spring.io>) um novo projeto Spring MVC:

- ◆ Project: Maven.
- ◆ Language: Java.
- ◆ Spring Boot: maior versão que não esteja com “SNAPSHOT”, “M” ou “RC”. Na data da produção deste conteúdo, a versão é 3.0.6.
- ◆ Group: com.senactds.
- ◆ Artifact e Name: cookiemvc.
- ◆ Packaging: Jar.
- ◆ Java: versão instalada em sua máquina (aqui é usada a 17).
- ◆ Dependencies: incluir “Spring Web” (categoria Web) e “Thymeleaf” (categoria Template Engines).

Clicando no botão “Generate”, baixando o arquivo compactado e descompactando-o em uma pasta, é aberto o projeto com o NetBeans.

Primeiramente, é criado um pacote “com.senactds.cookieMvc.controller” e, nele, uma classe CookiesController:

```
package com.senactds.cookieMvc.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
@RequestMapping("/cookies")
public class CookiesController {
}
```

Dentro dessa classe, criar uma ação responsável por criar um *cookie*:

```
@RequestMapping("/grava")
public String criaCookie(HttpServletResponse response){
    Cookie novoCookie = new Cookie("user-id", "123abc");
    response.addCookie(novoCookie);
    return "criacookie";
}
```


São necessários “import jakarta.servlet.http.Cookie;” e “import jakarta.servlet.http.HttpServletResponse;”.

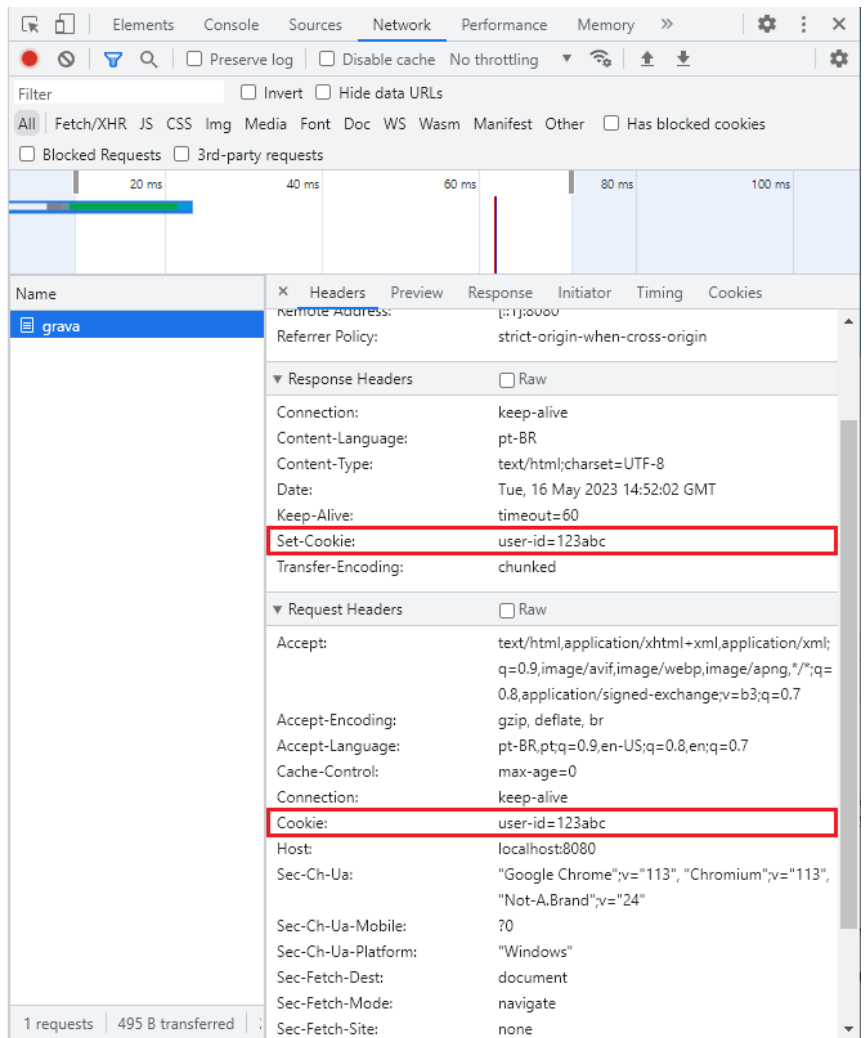
O método é simples e apenas cria um *cookie* com chave “user-id” e valor “123abc”, sem outras configurações. Em seguida, o inclui na resposta à requisição. Será criado o arquivo “criacookie.html” na pasta “src/main/resources/templates” para a visão desta ação.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cookies - criação</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale
=1.0">
  </head>
  <body>
    <p>Gravando cookie</p>
  </body>
</html>
```

Já é possível testar rodando a aplicação e usando a URL “http://localhost:8080/cookies/grava” no navegador.

A resposta deve ser uma simples frase “Gravando cookie” na tela, mas o mais interessante pode ser visto na ferramenta de desenvolvedor do navegador (Chrome DevTools, por exemplo). Use a aba Rede (Network) e recarregue a página. Veja os detalhes da única requisição que aparece e note, na aba Cabeçalho (Headers), os detalhes da resposta (Response Headers) que mostra uma linha “Set-Cookie: user-id=123abc”. Isso indica que o código funcionou e o *cookie* foi enviado ao navegador para que seja gravado. Uma prova disso é observada logo abaixo na seção “Request Headers”: ao recarregar a página, o *cookie* já existe, assim ele é passado pela requisição. Por isso, é possível notar ali uma linha “Cookie” que apresenta o valor “user-id=123abc”.

Gravando cookie

Figura 3 – Testando a criação de *cookie*

Fonte: Google Chrome (2023)

Agora será criada uma ação de leitura do *cookie* criado, a partir de um novo método na classe `CookieController`.

```
@RequestMapping("/le")
public String leCookie(@CookieValue(name="user-id", defaultValue="nenhum
-valor")String userId, Model model){
    model.addAttribute("userid", userId);
    return "lecookie";
}
```

São necessários `“import org.springframework.web.bind.annotation.CookieValue;”` e `“import org.springframework.ui.Model;”`

Atenção à anotação `@CookieValue`, aplicada ao parâmetro `String userId`. De acordo com o argumento `“name”` da anotação, está sendo buscado o *cookie* `“user-id”` e, de acordo com o argumento `“defaultValue”`, caso o *cookie* não exista, retornará o texto `“nenhum-valor”`.

valor". O valor do *cookie* é repassado ao parâmetro "userId" do método.



O segundo parâmetro é do tipo Model, apenas para se comunicar com a visão HTML.

Na pasta "templates", foram incluídos, então, lecookie.html.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cookies - leitura</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
    <p>Valor do cookie 'user-id':<span th:text="${userid}" /> </p>
  </body>
</html>
```

É possível recompilar o código e rodar novamente o projeto para testá-lo. No navegador, usar a URL "http://localhost:8080/cookies/le". Caso o cookie tenha sido gravado anteriormente, aparecerá a mensagem "Valor do cookie 'user-id':123abc". Caso contrário, aparecerá "Valor do cookie 'user-id':nenhum-valor".

Para limpar *cookies*, você pode usar as ferramentas de limpeza de histórico do seu navegador ou usar nova janela anônima.

Para excluir um *cookie*, deve-se simplesmente recriá-lo com a propriedade Max-Age em 0, ou seja, ele deve expirar imediatamente. Veja o teste com um novo método em CookiesController:

```
@RequestMapping("/exclui")
public String excluiCookie(HttpServletResponse response){
    Cookie novoCookie = new Cookie("user-id", null);
    novoCookie.setMaxAge(0);
    response.addCookie(novoCookie);
    return "excluicookie";
}
```

A diferença com relação ao método `criaCookie()` é a passagem do valor “null” ao construtor `Cookie()` e o uso do método `setMaxAge()` para definir expiração imediata.

A visão `excluicookie.html` correspondente deve ser criada na pasta “templates”:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cookies - exclusão</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale
=1.0">
  </head>
  <body>
    <p>Excluindo cookie</p>
  </body>
</html>
```

Agora você pode testar recompilando e rodando novamente o projeto e, no navegador, acessando primeiro `http://localhost:8080/cookies/grava`, depois `http://localhost:8080/cookies/le`, que deve mostrar o valor “123abc” para o *cookie*; em seguida `http://localhost:8080/cookies/exclui` e depois `http://localhost:8080/cookies/le` de novo, que agora deve mostrar “nenhum-valor”.

Caso, na criação, tenha sido especificada a propriedade “domain” ou “path”, é necessário que os mesmos valores dessas propriedades sejam incluídos na criação do *cookie* para exclusão. Caso contrário, o sistema considerará como dois *cookies* diferentes.

Crie um projeto Spring MVC com uma página que registre *cookie* com a data e a hora atuais e outra página que resgate esse *cookie*, mostrando a data e a hora armazenadas. Experimente também calcular quantos segundos se passaram desde a hora armazenada anteriormente no *cookie* (estude sobre a classe `LocalDate` para entender como converter texto para data e operar sobre os valores).

Aplicabilidade: *cookies* em projeto Spring MVC

Agora será aplicado mais funcionalmente o uso de *cookies*. Você pode criar um novo projeto ou seguir no projeto anterior, conforme a seguir. Será criada uma página que permita a configuração do nome do usuário e um esquema de cores preferido e outra que aplique o estilo escolhido e mostre o nome do usuário. Tudo será gravado com *cookies*.

Primeiramente, será criada uma classe `SiteController` no pacote “`com.senactds.cookiemvc.controller`”.

```
package com.senactds.cookiemvc.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class SiteController {
    @RequestMapping("/preferencias")
    public String preferencias(){
        return "preferencias";
    }
}
```

Já está sendo criada uma primeira ação que responde à URL “/preferencias”, executando o método `preferencias()`. Agora será desenvolvida a visão correspondente: em “templates” criar `preferencias.html`.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Site - Preferências</title>
        <meta charset="UTF-8"/>
        <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
    </head>
    <body>
        <h1>Informe suas preferências para o site</h1>
        <form th:action="@{/preferencias}" method="POST">
            <p><label for="txtNome">Seu nome:</label><input type="text" name="nome" id="txtNome"/></p>
            <p><label for="selEstilo">Seu esquema de cores favorito</label>
            <select name="estilo" id="selEstilo">
                <option value="claro">Tema Claro</option>
                <option value="escuro">Tema Escuro</option>
            </select></p>
            <p><input type="submit" value="OK"/></p>
        </form>
    </body>
</html>
```

Com o código acima, está sendo criada uma página com um formulário que tenha um campo para nome e um campo de seleção com os valores “claro” e “escuro” para o estilo da página. Não está sendo vinculado um objeto diretamente; em vez disso, foram usados os atributos “name” dos campos para preencher os dados no ato do POST.

Antes de criar a ação que recebe o POST, crie uma classe de modelo chamada “Preferencia” em um novo pacote “com.senactds.cookiemvc.model”.

```
package com.senactds.cookiemvc.model;
public class Preferencia {
    private String nome;
    private String estilo;
    public String getNome() { return nome; }
    public void setNome(String nome) { this.nome = nome; }
    public String getEstilo() { return estilo; }
    public void setEstilo(String estilo) { this.estilo = estilo; }
}
```

Note que os atributos têm nomes idênticos aos informados nos atributos “name” dos campos do formulário HTML – isso é essencial para que os dados possam ser transmitidos ao *back-end* na postagem do formulário.

A seguir, há o código para o método gravaPreferencias(), que deve ser programado na classe SiteController. Esse método processará os dados do formulário.

```
@PostMapping("/preferencias")
public ModelAndView gravaPreferencias(@ModelAttribute Preferencia pref, HttpServletResponse response){
    Cookie cookiePrefNome = new Cookie("pref-nome", pref.getNome());
    cookiePrefNome.setDomain("localhost"); //disponível apenas no domínio "localhost"
    cookiePrefNome.setHttpOnly(true); //acessível apenas por HTTP, JS não
    cookiePrefNome.setMaxAge(86400); //1 dia
    response.addCookie(cookiePrefNome);
    Cookie cookiePrefEstilo = new Cookie("pref-estilo", pref.getEstilo());
    cookiePrefEstilo.setDomain("localhost"); //disponível apenas no domínio "localhost"
    cookiePrefEstilo.setHttpOnly(true); //acessível apenas por HTTP, JS não
    cookiePrefEstilo.setMaxAge(86400); //1 dia
    response.addCookie(cookiePrefEstilo);
    return new ModelAndView("redirect:/"); //"index";
}
```

São necessários `import org.springframework.web.bind.annotation.PostMapping;`
`import jakarta.servlet.http.HttpServletResponse;` `import`
`com.senactds.cookiemvc.model.Preferencia;` `import`
`org.springframework.web.bind.annotation.ModelAttribute;` `import`
`org.springframework.web.servlet.ModelAndView;` e `import jakarta.servlet.http.Cookie.`

Veja aqui alguns detalhes: o primeiro é uso de `@PostMapping`, indicando que esse método é alcançado apenas com requisições do tipo POST. O segundo detalhe é o retorno do tipo `ModelAndView`, que permitirá que, ao concluir o processamento do método, seja redirecionado para outra ação (no caso, ainda será programada essa ação logo mais). Isso porque é desejável que, depois de registrar as preferências, o usuário volte à página inicial. O terceiro ponto é o uso de `@ModelAttribute` para o parâmetro do tipo “Preferencia”. Ele fará com que os dados do formulário sejam preenchidos nos campos do objeto “pref”.

Por fim, a criação de *Cookies*, que não difere muito dos exemplos já praticados aqui. Nesse caso, no entanto, estão sendo configuradas algumas propriedades: `setDomain()` para que o *cookie* só seja visto no domínio “localhost”, `setHttpOnly()` para que o *cookie* não seja manipulado com JavaScript e `setMaxAge()` para que o *cookie* expire em 24 horas. Os *cookies* criados têm nome “pref-estilo” e “pref-nome” e usam como valores os campos “estilo” e “nome” do objeto “pref” preenchido no parâmetro.

Siga programando `SiteController`, agora incluindo o método `index()`.

```
@RequestMapping("/")
public String index(@CookieValue(name="pref-nome", defaultValue="")String nome,
    @CookieValue(name="pref-estilo", defaultValue="claro")String tema,
    Model model){
    model.addAttribute("nome", nome);
    model.addAttribute("css", tema);
    return "index";
}
```

São necessários `import org.springframework.web.bind.annotation.CookieValue;` e
`import org.springframework.ui.Model.`

A ação `index()` deve executar para a página inicial do *site*. Por isso, é usado `@RequestMapping("/")`, ou seja, será executado a partir de `"localhost:8080/"`. É usada a anotação `@CookieValue` para incluir o valor dos *cookies* `"pref-nome"` e `"pref-estilo"` nos parâmetros `"String nome"` e `"String tema"`, definindo também valores-padrão. No corpo do método, passe como atributo de `Model` os valores capturados dos *cookies* e, por fim, requisitar a visão `"index"`.

Veja que há um processamento com relação aos *cookies* necessários antes de remeter à visão `index.html`. É por isso que, no método `gravaPreferencia()`, se faz um redirecionamento a esse método em vez de simplesmente mostrar a visão `"index"` direto. Ou seja, a partir do `"return new ModelAndView("redirect:/");"` de `gravaPreferencia()`, `index()` é executado.

Agora crie a página inicial `index.html` em `"templates"`.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Site - página inicial</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link th:href="@{/css/}+${css}+'.css'" rel="stylesheet" type="text/css"/>
  </head>
  <body>
    <h1>Olá <span th:text="${nome}"/> !</h1>
    <p>Seja muito bem-vindo(a) ao seu site! </p>
    <p>O estilo atual da página é <strong th:text="${css}" /></p>
    <p>Você pode configurar suas preferências de exibição <a href="/preferencias">ne
ste link</a></p>
  </body>
</html>
```

A página é muito simples, mas fique atento às intervenções do Thymeleaf:

- ◆ `th:href="@{/css/}+${css}+'.css'"` está apenas indicando que a página deve procurar na pasta `css` o arquivo definido pelo nome atribuído ao `Model` pela chave `"css"` (a linha `"model.addAttribute("css", tema);"` do método `index()`). Assim, a escolha do CSS é dinâmica: se o usuário escolheu tema claro, usará `"claro.css"`; se escolheu escuro, usará `"escuro.css"`.
- ◆ `th:text="${nome}"` e `th:text="${css}"` estão capturando o valor do atributo `"nome"` passado ao `Model` pelo método `index()`.

Por fim, crie os arquivos CSS. Antes, navegue no projeto por “Other Sources” > “src/main/resources” e clique com o botão direito em “static”, selecionando “New”>”Folder” para criar uma subpasta apenas para os arquivos .css. Dentro dessa nova pasta, crie o arquivo claro.css:

```
body{
  background-color: white;
  font-family: Verdana;
  color: #505050
}
a, a:visited{
  color: #505050;
}
```

Também crie na pasta o arquivo escuro.css:

```
body{
  background-color: #505050;
  font-family: Verdana;
  color: white;
}
a, a:visited{
  color: white;
}
```

E, por fim, é possível testar, rodando o projeto e acessando localhost:8080.

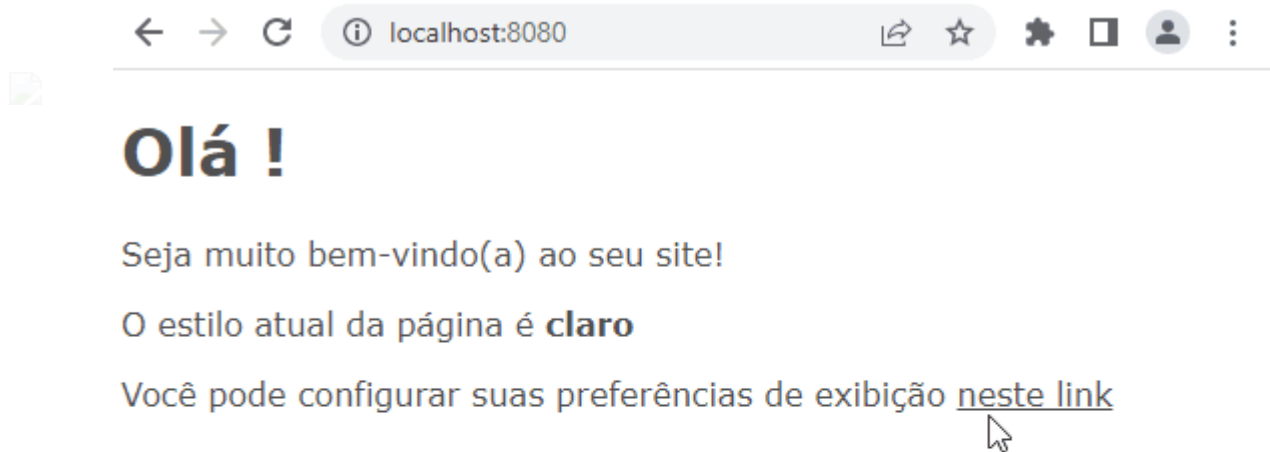


Figura 4 – Retorno página

Fonte: Senac EAD (2023)

Incremente o projeto incluindo uma terceira preferência em que o usuário escolhe se prefere fonte Times New Roman ou Verdana.

Aplicabilidade: *cookies* em projetos RESTful

Apesar de *webservices* RESTful, por seu conceito, prezarem pela ausência de estado, é possível criar *Cookies* e recuperá-los por meio desses serviços. O procedimento não se diferencia em nada do aplicado para Spring MVC. É hora de exercitar com um exemplo, criando um projeto com o Spring Initializr.

- ◆ No *site* do Spring Initializr, configure “Project: Maven”, “Spring Boot” na maior versão que não seja Snapshot, RC ou M2, “Group: com.senactds”, “Artifact” e “Name” com valor “cookierestful”, “Packaging: Jar” e “Java: 17”, ou versão correspondente à instalada em sua máquina.
- ◆ Em “Dependencies”, inclua “SpringWeb”.
- ◆ Gere o projeto e o abra com o NetBeans.

No NetBeans, crie um *package* “com.senactds.cookie Restful.controller”. Por enquanto, será necessário apenas esse pacote, no qual será criada a classe “CookiesController”. Nela, use o seguinte código:

```
import jakarta.servlet.http.Cookie;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.web.bind.annotation.CookieValue;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
@RequestMapping("/cookies")
public class CookiesController {
    @GetMapping("/grava")
    public String registraCookie(HttpServletResponse response) {
        Cookie c = new Cookie("user-id", "123abc");
        response.addCookie(c);
        return "Gravando cookie";
    }
    @GetMapping("/le")
    public String leCookie(@CookieValue(name="user-id", defaultValue="nenhum-valor") String userId) {
        return "cookie: " + userId;
    }
}
```

A maior diferença está na natureza dos retornos (os textos são o conteúdo da resposta) e no uso da anotação `@RestController`. De resto, o processo é idêntico ao adotado com Spring MVC. É possível testar rodando o projeto e usando o navegador ou o Postman para as requisições.

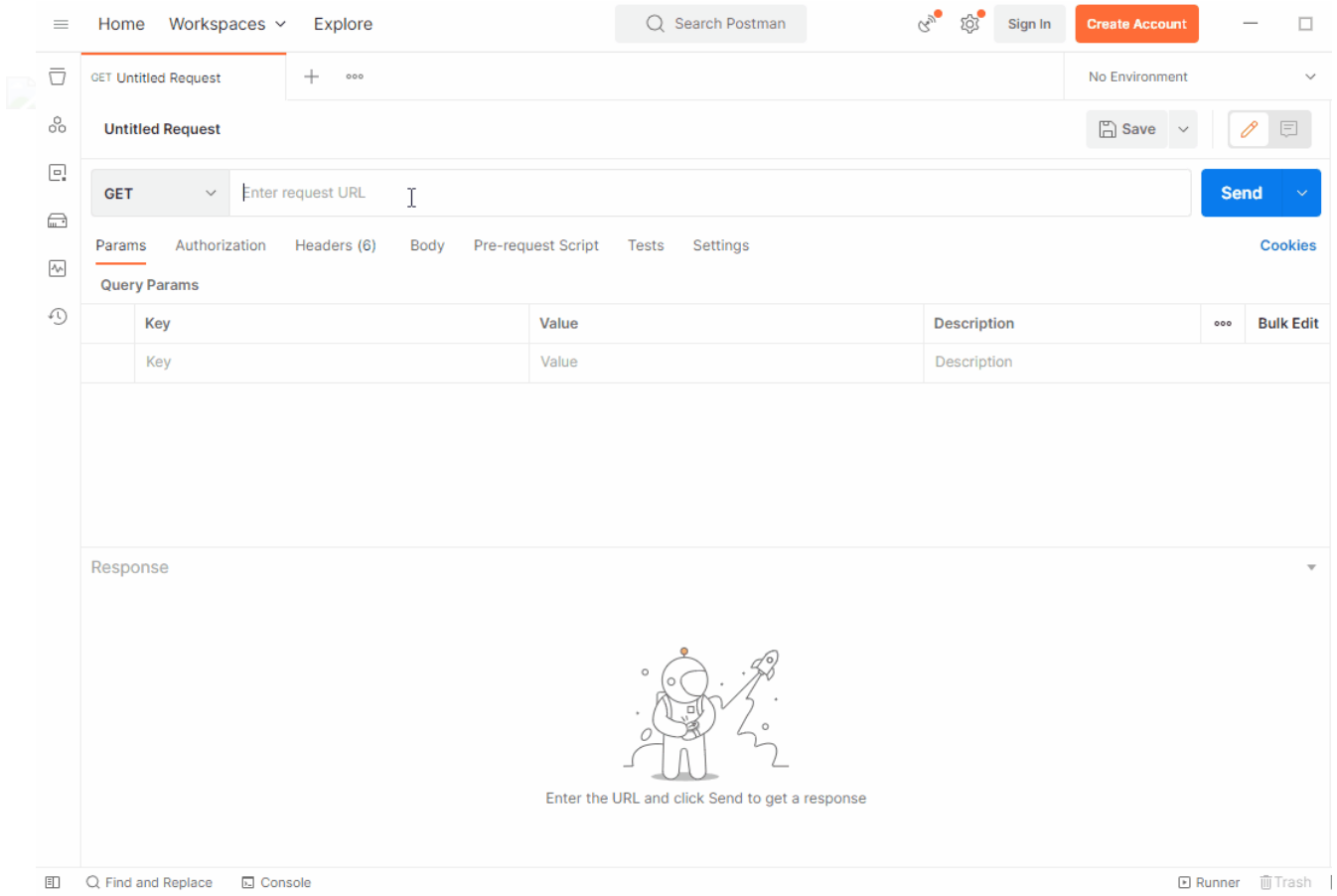


Figura 5 – Página Postman

Fonte: Postman (2023)

A operação de exclusão de *cookies* também obedece ao já estudado anteriormente.

Sessões (*sessions*)

Outro meio de se guardar informações entre requisições HTTP em um *site* é o uso de sessões ou *Sessions*. Trata-se de um meio de armazenamento, semelhante a *cookies*, mas que ocorre diretamente no servidor.

Em termos gerais, diz-se que uma sessão indica o tempo que um usuário visita e navega por um *website*, iniciando quando ele requisita a primeira página e terminando quando ele não requisita mais nenhuma página por um tempo determinado (chamado **timeout**). Informações podem ser armazenadas em formato **chave-valor** (ou seja, se obtém uma informação a partir de um identificador) no servidor para persistirem enquanto durar a sessão. Durante a sessão, o usuário pode requisitar quantas e quais páginas quiser e as informações estarão presentes para as páginas durante toda essa navegação.

Sessões são normalmente usadas para manter **autenticado** um usuário. A frase “Sua sessão expirará em breve” é comum em *websites*, indicando que uma sessão foi usada para manter os dados do usuário e que ela tem um tempo de expiração (o *timeout*). Os dados de sessão podem ser armazenados diretamente no servidor *web* (Tomcat ou Glassfish, por exemplo), em mecanismos de armazenamento temporários de dados (Redis, por exemplo) ou em banco de dados. É comum que as aplicações *web* façam uso de *cookies* para identificar a sessão de um usuário.

Em Java, o meio mais direto e comum de usar sessão é por meio das classes `HttpSession` e `HttpServletRequest`, a partir de `HttpServletRequest`, é recuperado o objeto de sessão da aplicação (`HttpSession`), que será usado para incluir, atualizar ou excluir informações que persistem entre requisições.

```
HttpServletRequest request;  
HttpSession ses = request.getSession();  
ses.setAttribute("user", "Meu Nome");
```

Os principais métodos da classe `HttpSession` são:

- ◆ *`getAttribute(String)`*: retorna o valor armazenado para uma chave informada por parâmetro. O retorno é do tipo `Object` e precisa ser convertido para o tipo adequado.
- ◆ *`setAttribute(String, Object)`*: inclui na sessão uma informação (parâmetro tipo `Object`) vinculada a uma chave (parâmetro tipo `String`).
- ◆ *`removeAttribute(String)`*: remove da sessão a informação vinculada à chave informada pelo parâmetro.
- ◆ *`getAttributeNames()`*: recupera uma lista de chaves armazenadas na sessão.
- ◆ *`getMaxInactiveInterval()`*: retorna o timeout da sessão (em segundos).
- ◆ *`setMaxInactiveInterval(int)`*: altera o timeout da sessão.
- ◆ *`getId()`*: recupera o identificador único da sessão.
- ◆ *`invalidate()`*: invalida uma sessão por completo, excluindo todos os valores armazenados nela.

É hora de exercitar com projetos MVC e Rest.

Aplicabilidade: sessões com Spring MVC



Crie a partir do Spring Initializr (<https://start.spring.io>) um novo projeto Spring MVC:

- ◆ Project: Maven.
- ◆ Language: Java.
- ◆ Spring Boot: maior versão que não esteja com “SNAPSHOT”, “M” ou “RC”. Na data da produção deste conteúdo, a versão é 3.0.6.
- ◆ Group: com.senactds.
- ◆ Artifact e Name: sessionmvc.
- ◆ Packaging: Jar.
- ◆ Java: versão instalada em sua máquina (aqui é usada a 17).
- ◆ Dependencies: incluir “Spring Web” (categoria Web) e “Thymeleaf” (categoria Template Engines).

Primeiramente, crie um pacote `com.senactds.sessionmvc.controller` e, nele, crie a classe `SessionController`, com o seguinte código:

```
package com.senactds.sessionmvc.controller;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpSession;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
@RequestMapping("/sessao")
public class SessionController {
    @RequestMapping("/grava")
    public String gravaSessao(HttpServletRequest request, Model model, String nome){
        HttpSession sessao = request.getSession();
        if(sessao != null){
            sessao.setAttribute("nome-usuario", nome);
            model.addAttribute("mensagem", "Gravando Sessao[nome]");
        }else{
            model.addAttribute("mensagem", "Sessão nula");
        }
        return "sessao";
    }
}
```

Atenção ao código da ação `gravaSessao()`:

- ◆ É necessário ter por parâmetro um objeto do tipo `HttpServletRequest`, que será automaticamente preenchido pelo mecanismo do Spring.
- ◆ A partir desse parâmetro, você pode recuperar o objeto `HttpSession` desta conexão.
- ◆ É boa prática verificar se a sessão não está nula. Nesse caso, use o método `setAttribute()` para atribuir o valor recebido pelo parâmetro “nome” à chave “nome-usuario”.
- ◆ Em seguida, inclua no objeto “model” uma mensagem a ser transmitida à camada de visão, informando sucesso na operação. Essa mensagem será de “Sessão nula” caso não haja sessão no *request*.

Em seguida, crie o método para leitura, também na classe `SessionController`:

```
@RequestMapping("/le")
public String leSessao(HttpServletRequest request, Model model){
    HttpSession sessao = request.getSession();
    String nome = "";
    if(sessao != null && sessao.getAttribute("nome-usuario") != null)
        nome = (String) sessao.getAttribute("nome-usuario");
    model.addAttribute("mensagem", "Sessao['nome'] = " + nome);
    return "sessao";
}
```

Para a leitura, você também precisa de um parâmetro `HttpServletRequest`, do qual é extraída a sessão e depois obtido o valor a partir do método `getAttribute()`. Esse método retorna um tipo `Object`, genérico, então é necessário converter para o tipo correto, como visto no código onde se usa “(String)”.

Por fim, crie a visão `sessao.html` na pasta “templates”:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>
```

```
<title>Sessões</title>
<meta charset="UTF-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
</head>
<body>
  <h1>Testando Sessão</h1>
  <p th:text="${mensagem}" />
</body>
</html>
```

Há um atributo de modelo “mensagem” para preencher a resposta das ações de *back-end*. Você já pode testar rodando o projeto e usando no navegador a URL `http://localhost:8080/sessao/grava?nome=Maria` – está sendo gravado o valor “Maria” na chave “nome-usuario” da sessão. O resultado é uma tela simples:

```
Testando Sessão
Gravando Sessao[nome]
```

Depois use a URL `http://localhost:8080/sessao/le` e a frase na página deve ser “Sessao['nome'] = Maria”, indicando que o valor foi salvo em sessão.

```
Testando Sessão
Sessao['nome'] = Maria
```

Agora com o funcionamento básico de sessões, você pode criar algo mais útil, como uma autenticação simples. Primeiramente, crie no pacote “com.senactds.sessionmvc.controller;” a classe `AutenticacaoController` e use como página inicial uma tela de *login*.

```
package com.senactds.sessionmvc.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class AutenticacaoController {
    @RequestMapping("/")
    public String index(){
        return "login";
    }
}
```


}



Em seguida, crie a visão login.html na pasta “templates”.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Sessões</title>
    <meta charset="UTF-8"/>
    <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  </head>
  <body>
    <h1>Login</h1>
    <form th:action="@{/autentica}" method="POST">
      <p><input type="text" name="login" id="txtLogin" placeholder="Login" />
    </p>
      <p><input type="password" name="senha" id="pswSenha" placeholder="Senha" /></p>
      <input type="submit" value="Autenticar" />
    </form>
  </body>
</html>
```

Você já pode testar essa ação rodando o projeto e usando a URL <http://localhost:8080>. No entanto, a funcionalidade principal será implementada agora. Primeiramente, crie um objeto do tipo `Usuario`, com atributos “login” e “senha” em um novo pacote “`com.senactds.sessionmvc.model`”.

```
package com.senactds.sessionmvc.model;
public class Usuario {
    private String login;
    private String senha;
    public String getLogin() { return login; }
    public void setLogin(String login) { this.login = login; }
    public String getSenha() { return senha; }
    public void setSenha(String senha) { this.senha = senha; }
}
```

Em seguida, crie um novo método `autenticar()` na classe `AutenticacaoController`.

```
@PostMapping("/autentica")
```

```
public String autentica(HttpServletRequest request, Usuario usuario){
    HttpSession sessao = request.getSession();
    if(sessao != null && usuario.getLogin().equals("adm") && usuario.getSenha().equals
("123")){
        sessao.setAttribute("usuario", usuario.getLogin());
        return "protegida";
    }
    return "login";
}
```

São necessários `import com.senactds.sessionmvc.model.Usuario;` `import jakarta.servlet.http.HttpServletRequest;` `import jakarta.servlet.http.HttpSession;` e `import org.springframework.web.bind.annotation.PostMapping;`

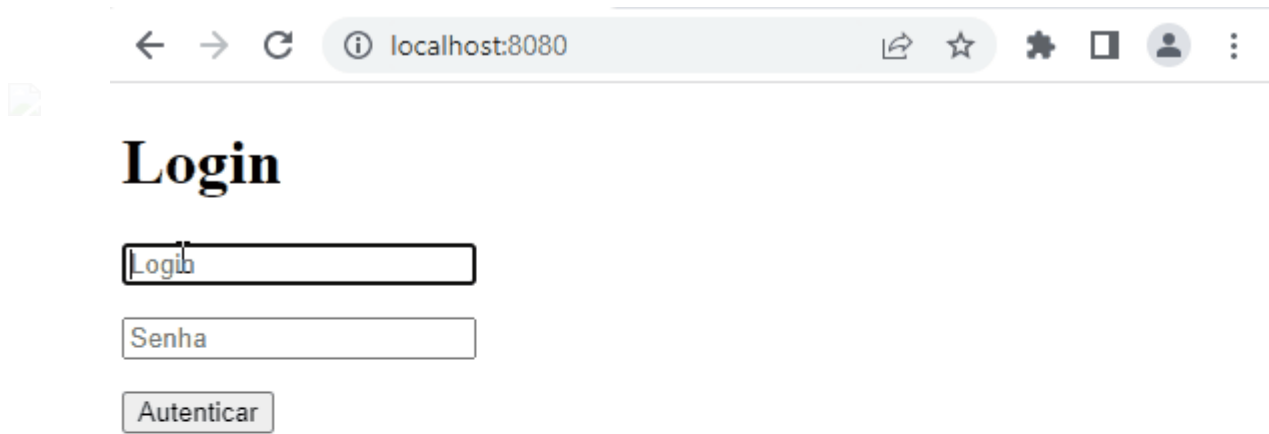
A lógica de autenticação é bem simples e apenas didática: o código aceitará apenas o usuário “adm” com senha “123” para a autenticação. Caso esses sejam os valores recebidos pelo POST, o usuário será direcionado à visão “protegida”, destinada apenas a usuário *logado*. Caso contrário, ele retorna à página de *login*.

Agora crie a visão protegida.html na pasta “templates”.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sessões</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
    <h1>Você está autenticado!</h1>
    <p>Se você está vendo esta página, então você está autenticado</p>
    <p><a href="/logoff">Clique aqui para fazer logoff</a></p>
  </body>
</html>
```

Já foi incluído na página um *link* para realizar *logoff*, mas essa funcionalidade será implementada em seguida.

Por agora, pare e teste rodando o projeto e informando *login* correto e incorreto.



The screenshot shows a web browser window with the address bar displaying 'localhost:8080'. The page content includes a large heading 'Login', a text input field with the placeholder 'Login', another text input field with the placeholder 'Senha', and a button labeled 'Autenticar'.

Figura 6 – Retorno página

Fonte: Senac EAD (2023)

Agora crie uma ação que leve direto para a visão “protegida” e verifique a autorização para essa página. Na classe `AutenticacaoController`, crie o seguinte método:

```
@RequestMapping("/protegida")
public ModelAndView acessaProtegida(HttpServletRequest request){
    HttpSession sessao = request.getSession();
    if(sessao != null && sessao.getAttribute("usuario") != null && sessao.getAttribute
("usuario").equals("adm")){
        return new ModelAndView("protegida");
    }
    return new ModelAndView("redirect:/");
}
```

É necessário import `org.springframework.web.servlet.ModelAndView`;

A verificação é simples: primeiramente, obtenha a sessão, verifique se a chave “usuário” está preenchida e é “adm”; caso afirmativo, permita que seja apresentada a visão “protegida”; caso negativo, volta ao *login*.

Em vez de texto, opte aqui por retornar o objeto ModelAndView, que permite um redirecionamento para a página de *login* (inclusive alterando a URL no navegador) quando o usuário não estiver autenticado.

Mais uma vez, pare e teste antes de continuar a codificação. Tente acessar “localhost:8080/protegida” sem *login*, depois tente de novo após se autenticar com sucesso.

Por fim, vamos implementar o *logout* incluindo o seguinte método na classe AutenticacaoController.

```
@RequestMapping("/logout")
public ModelAndView fazLogout(HttpServletRequest request){
    HttpSession sessao = request.getSession();
    if(sessao != null )
        sessao.removeAttribute("usuario");
    return new ModelAndView("redirect:/");
}
```

Novamente, o método é simples: apenas use o removeAttribute() de HttpSession e depois retorne o usuário à página de *login*. Como o *link* da visão protegida.html aponta para “/logout”, já é possível testar de novo.



Figura 7 – Retorno página

Fonte: Senac EAD (2023)

Importante ressaltar que, por padrão, Sessões ficam armazenadas em memória volátil do servidor, e não no navegador do usuário como *cookies*. Assim, ao reiniciar uma aplicação *web* com sessão, ela será excluída. Existem implementações de sessão que usam banco de dados ou outros recursos que evitam essa situação.

Crie um novo projeto Spring MVC com base no exemplo de *cookies* para preferências do *site*. Mas, em vez de armazenar os dados em *cookies*, guarde-os em sessões.

Aplicabilidade: sessão com projeto RESTful

Assim como acontece com *cookies*, sessões não são muito recomendadas para aplicações RESTful por quebrar a premissa *stateless* desse tipo de abordagem. No entanto, é possível utilizar sem muitos problemas sessões, e o procedimento é exatamente o mesmo aplicado para Spring MVC.

Para o exemplo a seguir, crie um novo projeto:

- ◆ No *site* do Spring Initializr, configure “Project: Maven”, “Spring Boot” na maior versão que não seja Snapshot, RC ou M2, “Group: com.senactds” “Artifact” e “Name” com valores “sessionrestful”, “Packaging: Jar” e “Java: 17”, ou versão correspondente à instalada em sua máquina.
- ◆ Em “Dependencies” inclua “SpringWeb”.
- ◆ Gere o projeto e o abra com o NetBeans.

Crie, em seguida, um pacote “com.senactds.sessionrestful.controller” e, nele, uma classe SessaoController. A seguir, veja o código completo para a classe.

```
package com.senactds.sessionrestful.controller;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpSession;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
@RequestMapping("/sessoes")
public class SessaoController {
    @GetMapping("/grava")
    public String gravaSessao(HttpServletRequest request, String nome){
        HttpSession sessao = request.getSession();
        if(sessao != null){
            sessao.setAttribute("usuario", nome);
            return "Sessão atualizada";
        }
        return "Sem sessão";
    }
    @GetMapping("/le")
    public String leSessao(HttpServletRequest request){
        HttpSession sessao = request.getSession();
        if(sessao != null){
            return "Valor da sessão: " + (String)sessao.getAttribute("usuario");
        }
        return "Sem sessão";
    }
    @GetMapping("/exclui")
    public String excluiSessao(HttpServletRequest request){
        HttpSession sessao = request.getSession();
        if(sessao != null){
            sessao.removeAttribute("usuario");
        }
        return "Excluído";
    }
}
```

Note que não há diferença com relação ao manejo de sessões relacionadas ao Spring MVC. Você pode rodar a aplicação e testar no navegador ou no Postman realizando esta sequência de requisições:



`http://localhost:8080/sessoes/grava`

`http://localhost:8080/sessoes/le?nome=Joaquim` (deve aparecer a frase “Valor da sessão: Joaquim” no retorno)

`http://localhost:8080/sessoes/exclui`

`http://localhost:8080/sessoes/le` (deve aparecer a frase “Valor da sessão: null” no retorno)

(imgs/infografico2_mobile.png)

Encerramento

Você concluiu seus estudos de *cookies* e sessões em Java. São recursos que devem ser usados pontualmente e que implicam em algumas questões, como o fato de que *cookies* podem simplesmente ser excluídos pelo usuário no navegador dele e que sessões podem sumir se o servidor reiniciar. Na maior parte do tempo, *cookies* serão aplicados para personalização ou troca de informações entre *sites* e sessões para autenticação e armazenamento temporário de dados que fazem parte de alguma ação do usuário, como um carrinho de compras, por exemplo.

De todo modo, são duas ferramentas importantes para o desenvolvedor web, que, sabendo como usá-las, poderá aplicá-las em soluções para problemas específicos.