计算机信息工程学院

2019 届毕业设计外文阅读与翻译

毕业设计题目 基于 Spring Boot 的房屋出租出售系统的设计与实现

外文翻译题目 Advanced Filtering Techniques

专业 软件工程 **班级** 15 软卓

姓名 吴勇伟 **学号** 15030428

指导教师 蒋巍 **职称** 讲师

评价参考	是否按要求完成外文翻译，译文的准确性，译文的总体质量，翻译工作量（英文不少于 15000 个印刷符号）；格式符合指导书要求。
指导教师意见	签名： 得分(5 分制)： 日期：

过滤器 4：Ad Hoc 验证过滤器

Tomcat 5 和几乎所有符合 Servlet 2.4 标准的容器都具有广泛的身份验证和授权支持，因此理论上几乎不应该需要自己去实现身份验证过滤器。但实际上，总是有空间在所选资源上应用 ad hoc 认证过滤器，而不会影响应用程序的其余部分或涉及设置 JDBC 领域的开销。

您应始终分析手头的问题，看看是否可以通过服务器的本机身份验证支持更好地解决问题。在您需要对所选资源进行简单，临时保护的情况下，AdHocAuthenticate 过滤器可能是最佳选择。

此过滤器的操作非常简单。它会在客户端浏览器上触发基本身份验证，几乎所有已知的浏览器，包括最早的版本，都支持基本身份验证。它的工作原理如下：

1.客户端尝试访问受保护资源。

2.服务器检查客户端的请求，以确定“授权”标头中是否有任何授权数据。

3.如果未找到授权数据，服务器将发回 HTTP 状态代码 401（未授权访问）和带 WWW-authenticate 的标头：BASIC realm = <realm>，其中 realm 是将显示给客户端的文本字符串。

4.客户端弹出登录屏幕，用户应在其中输入用户名和密码。

5.客户端使用简单的 base64 编码对用户名和密码进行编码，并将两者都发送到服务器。

6.服务器检查客户端请求以确定是否存在任何授权数据，必要时解码 base64 编码的密码。如果没有授权数据，则返回步骤 3。

7.服务器验证密码并允许或拒绝访问。

注意，基本身份验证不是很安全，因为 base64 编码很容易被破译。但是，对于只需要保护资源不被临时的应用程序访问就已经足够了。有关不同类型身份验证的更多详细信息，请参阅第 11 章。

AdHocAuthenticate 过滤器识别两个密码：一个用于“常规”用户，另一个用于特权“金牌成员”用户。两个密码都配置为过滤器的初始参数。如果用户使用“gold member”密码登录，则会创建一个布尔属性并将其附加到请求。您将在稍后（在管道处理过滤器部分中）了解如何使用此属性。现在，让我们关注此过滤器的身份验证操作。

AdHocAuthenticateFilter 类

以下是对该类源代码的简要分析：

```
package com.apress.projsp20.ch10;
    import java.io.*;
    import javax.servlet.*;
    import javax.servlet.http.*;
    import java.util.Map;
    import sun.misc.*;
```

无需在此过滤器中包含响应或请求。请注意将用于保存两个只读密码的实例变量：adhocPassword 和 adhocGoldPassword。这些变量由容器通过 init（）方法从 web.xml 值初始化：

```
public final class AdHocAuthenticateFilter implements Filter {

        private FilterConfig filterCofig = null;
        private String adhocPassword = null;
        private String adhocGoldPassword = null;

        public void doFilter(ServletRequest request, ServletResponse response,FilterChain chain)
                throws IOException, ServletException {
          if (filterConfig == null) return;
```

您将请求和响应强制转换为其 HTTP servlet 版本，以访问和操作与其关联的标头：

```
HttpServletRequest myReq = (HttpServletRequest) request;
        HttpServletResponse myResp = (HttpServletResponse) response;
```

如果您没有找到授权数据，请执行基本身份验证请求：

```
String authString = myReq.getHeader("Authorization");
        if (authString == null) {
           myResp.addHeader("WWW-Authenticate", "BASIC realm=\"PJSP2\"");
           myResp.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
           return;
        } else { // authenticate
```

如果找到授权数据，则解码用户名和密码。以下子字符串（6）跳过授权标头的常量字符串 Basic，到达 base64 编码的用户名和密码的开头：

```
BASE64Decoder decoder = new BASE64Decoder（）;
String enString = authString.substring（6）;
String decString = new String（decoder.decodeBuffer（enString））;
int idx = decString.indexOf（“：”）;
String uid = decString.substring（0，idx）;
String pwd = decString.substring（idx + 1）;
```

您可以调用方法 externalGoldAuthenticate（）和 externalAuthenticate（）来分别对“gold member”用户和“常规”用户执行实际身份验证。 在生产中，可以通过外部服务器进行身份验证。 成功的身份验证允许访问受保护资源。 此外，“gold member”身份验证将导致将布尔属性附加到请求。 验证失败将导致登录对话框在客户端的浏览器上再次弹出：

```
        if (externalGoldAuthenticate(uid,pwd)) {
           request.setAttribute("goldmember", new Boolean(true));
        } else {
           if (!externalAuthenticate(uid,pwd)) {
              myResp.addHeader("WWW-Authenticate", "BASIC realm=\"PJSP2\"");
      myResp.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
              return;
           } // of if
        } // of else
     } // of outer else
```

过滤器 5：请求处理管道中的过滤器

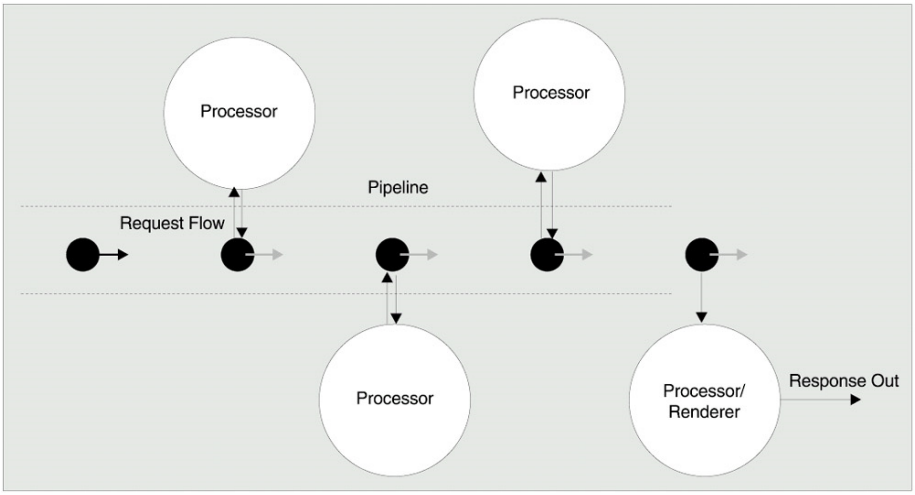
到目前为止，本章的重点是过滤器，它可以控制请求流（停止或让它通过）生成对请求的响应。由于它们的应用领域可能很广，这些滤波器设计风格仅部分涵盖了滤波器的潜在应用范围。事实上，这些过滤器直接负责响应 HTML 页面最终外观的某些部分，这常常使得它们难以组合（以增值组件方式链接）。这是因为，根据定义，它们特定于它们直接生成的页面输出。

管道数据处理模型提供了一种新的过滤器样式，我们的第五个和最后一个过滤器就是例证。此模型的第一个启用程序是 servlet <filter-mapping>定义的新<dispatcher>子元素。新元素使过滤器能够参与管道请求处理模型的每个阶段。在这个模型中，过滤器本身就是通过管道传输请求的真正的处理器。

了解管道模型

流水线数据处理模型有许多不同的名称。 它通常被认为是模型 - 视图 - 控制器（MVC）Web 应用程序设计的支持元素，有时它被称为应用程序设计的推送模型，与更传统的拉模型相反。

这个想法很简单，但你必须超越传统的 Web 应用程序智慧来思考大局。 考虑一下这个模型的例子：



请求进入系统并沿管道穿梭。当它遍历管道时，一系列处理器可以访问请求的内容。每个处理器对请求执行一些任务，然后它附加一些新属性或修改现有属性作为该工作的结果。请求保持完整，直到它到达管道的最后阶段，其中称为渲染器的特殊处理器检查对请求执行的所有工作并生成最终响应（渲染器在 MVC 范例中也称为视图组件，因为它们本身就是负责最终呈现给用户）。你可以看出为什么这通常被称为“推”模型;数据属性由处理器提取，随请求一起沿管道推送，并仅在最后阶段呈现。

管道模型的另一个常见类比是输送带或装配线，其中每个处理器沿着带或线镜像工人（或机器人）。

一些非常理想的管道模型的属性如下：

请求在遍历管道时保持不变，只在附加的属性（有时称为装饰器）上执行工作。

数据管理，业务逻辑和表示逻辑可以干净地分成不同的处理器和渲染器。

处理器可以设计为完全可组合（即以任何顺序链接）或高度专业化（仅适用于由其他处理器创建或修改的特定属性）。

多个渲染器可以组成最终输出（例如通过 XSLT 将数据转换为 XML 到 HTML）。

请求的状态随着请求通过管道传播。处理器和渲染器根本不跟踪依赖于请求的状态。这使得可以在必要时在多个物理服务器之间复制或穿梭请求及其状态。

通常，使用管道模型设计请求处理逻辑提供以下内容：

清洁，组件化的处理器和易于重复使用的渲染器

应用程序中数据管理，业务逻辑和表示逻辑之间的明确分离

更易于维护的应用程序

更适应不断变化的业务需求的应用程序

可通过容器技术和硬件进步扩展的强大应用程序

性能高度可优化的 Web 应用程序

最后一点可能不会立即显示，但管道中任何不依赖于其他结果的处理器，原则上可以同时执行，（例如，可能在两个物理处理器上）和任何具有结果独立的处理器组其他人也可以同时执行。

过去，容器实施的限制阻碍了使用清洁的管道设计模型。 JSP 2.0 和 Servlet 2.4 具有新功能，使创建流水线应用程序的梦想更接近现实。

将过滤器插入管道

使用<dispatcher>子元素可以将过滤器插入到请求处理管道中。回想一下，<dispatcher>元素现在允许过滤器拦截请求调度程序的 forward（）和 include（）调用。这些是 Servlet 2.3 容器以前无法使用

的处理管道中的其他位置，其中过滤器可以为您工作。您已经在最后一章中了解了这种机制是如何工作的，所以让我们在这里看到它的实际应用。

首先，您将重新访问上一章中看到的 SimpleFilter 类。它只是在日志中写了几行，让你知道它已被调用。现在，您将更改它以处理通过管道传输的请求。

现在只需更改附加属性，而不是直接生成日志输出。修改将查看请求中是否存在名为 MsgOut 的属性。如果它确实存在，那么代码将附加“：SimpleFilter”;否则，MsgOut 将设置为“SimpleFilter”（没有前导冒号）。过滤器（处理器）第一次对请求进行操作时，附加的 MsgOut 属性将设置为“SimpleFilter”。第二次之后，它将包含“SimpleFilter：SimpleFilter”，并在第三次之后包含“SimpleFilter：SimpleFilter：SimpleFilter”。

最终，MsgOut 属性将显示在 HTML 页面上，因此您可以看到此过滤器在通过管道传输时对特定请求的操作次数。你会在中找到代码：

com.press.projsp20.ch10.SimpleFilter class:

```
package com.apress.projsp20.ch10;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public final class SimpleFilter implements Filter {
   private FilterConfig filterConfig = null;

   public void doFilter(ServletRequest request, ServletResponse response,
                                FilterChain chain)
            throws IOException, ServletException {
      if (filterConfig == null) return;
      filterConfig.getServletContext().log("in SimpleFilter");
      Object curVal = request.getAttribute("MsgOut");

      if (curVal == null) {
         request.setAttribute("MsgOut", new String("SimpleFilter"));
      } else {
         request.setAttribute("MsgOut", (String) curVal + " :SimpleFilter");
      }
      chain.doFilter(request, response);
      filterConfig.getServletContext().log("leaving SimpleFilter");
   }
```

您还将修改 FindProd.jsp 文件（管道中的渲染器处理器）以显示新属性。 JSP 2.0 的 EL 非常适合创建渲染器 --从附加属性值呈现 HTML 的 JSP 页面。 你将这个修改过的文件命名为 Sub.jsp：

```
<html>
   <head></head>
   <body>
      <h1>You have accessed this page from the ${param.DEPT} department!</h1>
      ${param.MsgForwarder}<br/>
      ${requestScope.MsgOut}
```

```
    </body>
</html>
```

附加到请求的 MsgOut 属性在此处使用 EL 显示。 请注意，还会显示一个名为 MsgForwarder 的参数。 稍后当您使用请求调度程序的 forward（）操作时，您将看到使用此参数。

默认 REQUEST-Only 筛选：Servlet 2.3 兼容性

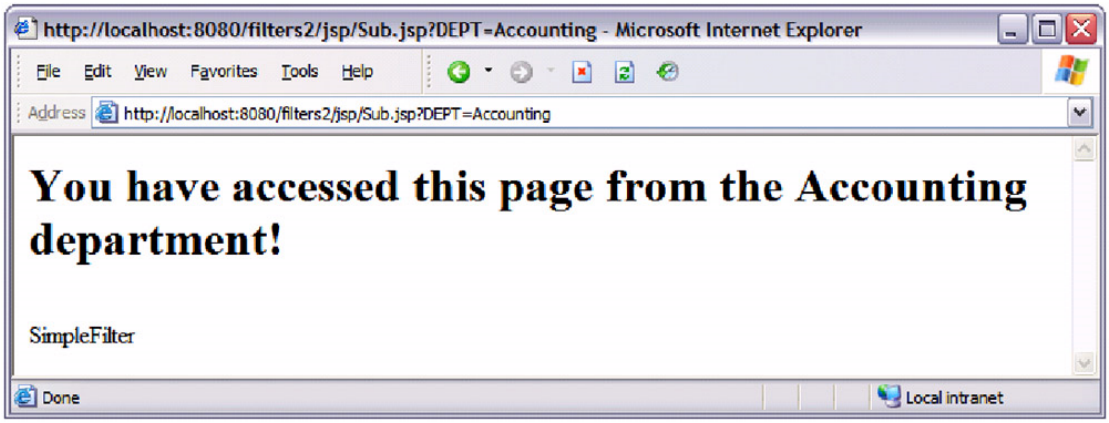
要指定仅对直接来自容器外部的请求执行过滤，可以在过滤器的<filter-mapping>元素中添加以下<dispatcher>元素：

```
<dispatcher>REQUEST</dispatcher>
```

例如，您可能在 web.xml 中具有以下内容：

```
<filter>
        <filter-name>Simple Push Filter</filter-name>
        <filter-class>com.apress.projsp20.ch10.SimpleFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>Simple Push Filter</filter-name>
        <url-pattern>/jsp/*</url-pattern>
        <dispatcher>REQUEST</dispatcher>
    </filter-mapping>
```

现在，如果您打开 URL http：// localhost：8080 / filters2 / jsp / Sub.jsp？DEPT = Accounting，您将看到如下页面：



请注意页面上的 MsgOut 属性的 EL 呈现 - 您知道过滤器已被调用一次。 这种行为就是在 Servlet 2.4 标准之前使用过滤器支持的所有行为。 如果出于兼容性原因未指定任何<dispatcher>子元素，那么它也是 Tomcat 5 的默认行为。

INCLUDE-Only Filtering

使用 Tomcat 5 和 Servlet 2.4，您可以指定您的过滤器仅适用于包含的请求。 通过将 web.xml 更改为此来尝试此操作：

```
    <filter-mapping>
        <filter-name>Simple Push Filter</filter-name>
        <url-pattern>/jsp/*</url-pattern>
        <dispatcher>INCLUDE</dispatcher>
    </filter-mapping>
```

重启 Tomcat 并尝试通过以下 URL 直接访问 Sub.jsp：

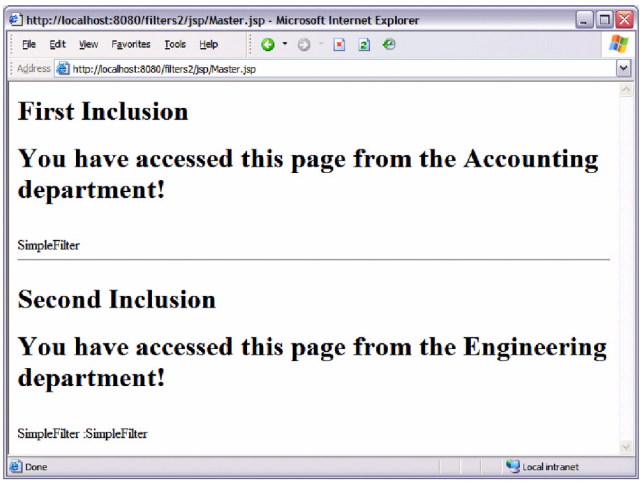
http://localhost:8080/filters2/jsp/Sub.jsp?DEPT=Accounting
请注意，MsgOut 属性不存在，表示结果页面尚未通过 SimpleFilter。

通过在<dispatcher>子元素中指定 INCLUDE，您说过滤器应该仅映射到包含的请求。 要查看此操作，请查看 jsp / Master.jsp 文件：

```
<html>
        <head></head>
        <body>

          <h1>First Inclusion</h1>
          <jsp:include page="/jsp/Sub.jsp" flush="true">
            <jsp:param name="DEPT" value="Accounting"/>
          </jsp:include>
          <hr/>

          <h1>Second Inclusion</h1>
          <jsp:include page="/jsp/Sub.jsp" flush="true">
            <jsp:param name="DEPT" value="Engineering"/>
          </jsp:include>

        </body>
      </html>
```

此 JSP 文件对请求调度程序进行两次调用，每次都包含您一直使用的/jsp/Sub.jsp。这两个 include（）动作的输出合并为输出响应。 请注意，您还要为每个包含提供不同的 DEPT 参数。

现在导航到 http：// localhost：8080 / filters2 / jsp / Master.jsp，结果页面应与以下内容匹配：



请注意 MsgOut 属性值 - 它反映了 SimpleFilter 的操作。 在第二次包含之后，第一次包含 Sub.jsp 和“SimpleFilter：SimpleFilter”之后，该属性为“SimpleFilter”。 该请求已由 SimpleFilter 操作两次，每次包含在 Master.jsp 中。

FORWARD-Only Filtering

要使过滤器仅对转发的请求起作用，请按如下所示修改 web.xml 文件：

```
  <filter-mapping>
        <filter-name>Simple Push Filter</filter-name>
```

```
        <url-pattern>/jsp/*</url-pattern>
        <dispatcher>FORWARD</dispatcher>
    </filter-mapping>
```

由于在<disptacher>元素中指定了 FORWARD，因此只有转发的请求才会传递给过滤器。 要查看此操作，请查看 jsp / Forwarder.jsp 文件：

```
<jsp:forward page="/jsp/Sub.jsp">
    <jsp:param name="DEPT" value="Accounting"/>
    <jsp:param name="MsgForwarder" value="Forwarded from forwarder.jsp"/>
</jsp:forward>
```

此 JSP 页面只是将请求转发给 Sub.jsp，并设置 DEPT 和 MsgForwarder 参数。 这两个参数都使用 EL 显示在 Sub.jsp 中。

重新加载应用程序或重新启动 Tomcat。 现在依次尝试以下网址：

http://localhost:8080/filters2/jsp/Sub.jsp?DEPT=Accounting
http://localhost:8080/filters2/jsp/Master.jsp
http://localhost:8080/filters2/jsp/Forwarder.jsp

对于前两个 URL，请注意不再过滤直接请求和包含的请求。 第三个 URL 应该产生类似于这样的页面：

您可以从此实验中看到，只过滤了转发的请求。

结合调度程序操作

当然，您可以使用多个<dispatcher>元素来指示要应用的过滤器的多个位置。 例如：

```
    <filter-mapping>
        <filter-name>Simple Push Filter</filter-name>
        <url-pattern>/jsp/*</url-pattern>
        <dispatcher>FORWARD</dispatcher>
        <dispatcher>INCLUDE</dispatcher>
        <dispatcher>REQUEST</dispatcher>
    </filter-mapping>
```

现在，尝试前三个网址，您将看到以下内容：

URL 1：单独请求时，过滤器在 Sub.jsp 上处于活动状态。
URL 2：过滤器在 Master.jsp PLUS 中处于活动状态，每个包含的 Sub.jsps。
URL 3：过滤器在 Forwarder.jsp 上处于活动状态 PLUS 转发的 Sub.jsp。

Filter 4: An Ad Hoc Authentication Filter

Tomcat 5 and almost all Servlet 2.4–compliant containers come with extensive authentication and authorization support, so in theory there should be little need to implement your own authentication filter. In practice, however, there's almost always room to apply an ad hoc authentication filter on a selected resource without affecting the rest of the application or involving the overhead of setting up, say, JDBC realms.

You should always analyze the problem at hand to see if it would be better solved by the native authentication support of the server. In those cases in which you need simple, temporary protection of selected resources, the AdHocAuthenticate filter can be the best choice.

The action of this filter is straightforward. It triggers basic authentication on the client browser. Almost all known browsers, including even the earliest versions, support basic authentication. It works like this:

1. A client attempts to access a protected resource.
2. The server examines the client's request to determine if there's any authorization data in the "Authorization" header.
3. If authorization data isn't found, the server sends back HTTP status code 401 (unauthorized access) and a header with WWW-authenticate: BASIC realm=<realm>, where realm is a text string that will be displayed to the client.
4. The client pops up a login screen in which the user should enter a username and password.
5. The client encodes the username and password using simple base64 encoding and sends both to the server.
6. The server examines the client request to determine if there's any authorization data, decoding the base64-encoded password if necessary. If there's no authorization data, it goes back to step 3.
7. The server verifies the password and either allows or rejects access.

Note Basic authentication isn't very secure, because base64 encoding can easily be deciphered. However, for applications that just need to protect resources from casual access, it's usually sufficient. For more details on different kinds of authentication, see Chapter 11.

The AdHocAuthenticate filter recognizes two passwords: one for "regular" users and one for privileged, "gold member" users. Both passwords are configured as initial parameters for the filter. If a user logs on using the "gold member" password, a Boolean attribute is created and attached to the request. You'll learn later (in the pipeline processing filters section) how this attribute is used. For now, let's focus on the authentication action of this filter.

The AdHocAuthenticateFilter Class

Here's a brief analysis of the source code for this class:

```
package com.apress.projsp20.ch10;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
import java.util.Map;
import sun.misc.*;
```

There's no need to wrap the response or request in this filter. Note the instance variable that will be used to hold the two read-only passwords: adhocPassword and adhocGoldPassword. These variables are initialized by the container from the web.xml values via the init() method:

```
public final class AdHocAuthenticateFilter implements Filter {

    private FilterConfig filterCofig = null;
    private String adhocPassword = null;
    private String adhocGoldPassword = null;
    public void doFilter(ServletRequest request, ServletResponse response,
                                    FilterChain chain)
            throws IOException, ServletException {
      if (filterConfig == null) return;
```

You cast the request and response to their HTTP servlet versions to access and manipulate the headers associated with them:

```
        HttpServletRequest myReq = (HttpServletRequest) request;
        HttpServletResponse myResp = (HttpServletResponse) response;
```

Here you perform the basic authentication request if you don't find authorization data:

```
        String authString = myReq.getHeader("Authorization");
        if (authString == null) {
          myResp.addHeader("WWW-Authenticate", "BASIC realm=\"PJSP2\"");
           myResp.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
           return;
        } else { // authenticate
```

If you find authorization data, you decode the username and password. The following substring(6) skips over the constant string Basic of the authorization header, getting to the beginning of the base64-encoded username and password:

```
        BASE64Decoder decoder = new BASE64Decoder();
        String enString = authString.substring(6);
        String decString = new String(decoder.decodeBuffer(enString));
        int idx = decString.indexOf(":");
        String uid = decString.substring(0, idx);
        String pwd = decString.substring(idx + 1);
```

You call the methods externalGoldAuthenticate() and externalAuthenticate() to perform the actual authentication for "gold member" users and "regular" users, respectively. In production, authentication via an external server could be implemented here. Successful authentication allows access to the

protected resource. In addition, "gold member" authentication will result in the, ahem, goldmember Boolean attribute being attached to the request. Failed authentication will cause the login dialog box to pop up again on the client's browser:

```
            if (externalGoldAuthenticate(uid,pwd)) {
               request.setAttribute("goldmember", new Boolean(true));
            } else {
               if (!externalAuthenticate(uid,pwd)) {
                  myResp.addHeader("WWW-Authenticate", "BASIC realm=\"PJSP2\"");
myResp.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
                  return;
               } // of if
            } // of else
         } // of outer else
```

Filter 5: A Filter in the Request Processing Pipeline

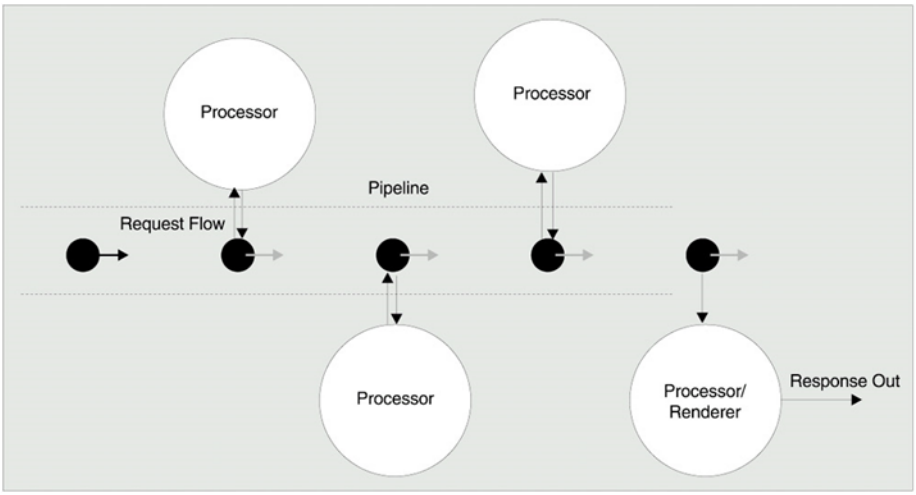
Thus far, this chapter's focus has been on filters that either control the flow of a request (stopping it or letting it through) or generate the response to a request. Vast as their fields of application may be, these filter design styles only partially cover the potential spectrum of applications for filters. The fact that these filters are directly responsible for some portion of the final appearance of the response HTML page often makes them difficult to compose (chain in a value-added component fashion). This is because, by definition, they're specific to the page output that they directly generate.

The pipeline data-processing model provides a new style of filter that our fifth and final filter exemplifies. The first enabler for this model is the new <dispatcher> child element of the servlet <filter-mapping> definition. The new element enables filters to participate in every stage of the pipeline request-processing model. In this model, filters are themselves bona fide processors for a request traveling through a pipeline.

Understanding the Pipeline Model

There are many different names given to the pipelined data-processing model. It's often identified as the enabling element of Model-View-Controller (MVC) web application design, and sometimes it's referred to as the push model of application design, in contrast with the more conventional pull model.

The idea is simple, but you have to think beyond the conventional web application wisdom to get the bigger picture. Consider this illustration of the model:



A request enters the system and is shuttled along a pipeline. As it traverses the pipeline, a sequence of processors has access to the content of the request. Each processor performs some task on the request, and then it either attaches some new attribute or modifies an existing attribute as the result of that work. The request remains intact until it hits the final stage of the pipeline, in which special processors called renderers examine all the work being done to the request and produce the final response (renderers are also called the view component in the MVC paradigm because they alone are responsible for the final presentation to the user). You can see why this is often called the "push" model; data attributes are fetched by processors, pushed along the pipeline with the request, and rendered only at the final stage.

Another common analogy for the pipeline model is a conveyer belt or assembly line, where each processor mirrors the workers (or robots) along the belt or line.

Some of the highly desirable properties of a pipeline model are as follows:

- The request stays intact as it traverse the pipeline, with work being carried out only on the attached attributes (sometimes called decorators).
- Data management, business logic, and presentation logic can be cleanly separated into different processors and renderers.
- The processors can be designed to be completely composable (that is, chained in any order) or highly specialized (to work only with specific attributes created or modified by other processors).
- Multiple renderers can compose the final output (such as data to XML to HTML via XSLT).
- The state of the request *travels with the request* through the pipeline. The processors and renderers don't keep track of the request-dependent state at all. This makes it possible to duplicate or shuttle the request and its state between multiple physical servers if necessary.

In general, designing request-handling logic using the pipeline model provides the following:

- Clean, componentized processors and renderers that are readily reusable
- Strong and clear separation between data management, business logic, and presentation logic in an application
- Applications that are more maintainable

- Applications that are more adaptable to changing business requirements
- Robust applications that will be scalable with container technology and hardware advances
- Web applications whose performance is highly optimizable

The last point may not be immediately obvious, but any processors in the pipeline that don't depend on others' outcomes can in principle be executed concurrently (perhaps on two physical processors, for instance) and any grouping of processors that has an outcome independent of others can also execute concurrently.

In the past, limitations in container implementation have prevented the use of a clean pipeline design model. JSP 2.0 and Servlet 2.4 have new features that bring the dream of creating a pipelined application closer to reality.

Inserting Filters into the Pipeline
The <dispatcher> subelement enables you to insert filters into the request processing pipeline. Recall that the <dispatcher> element now allows filters to intercept the request dispatcher's forward() and include() calls. These are additional locations in the processing pipeline previously unavailable with Servlet 2.3 containers, where filters can go to work for you. You've learned how this mechanism works in the last chapter, so let's see it in action here.

First, you'll revisit the SimpleFilter class that you saw in the last chapter. All it did was write a couple of lines to the log, letting you know that it had been invoked. Now you'll change it to process the request traveling through the pipeline.

Instead of directly generating log output, it will now simply change attached attributes. The modification looks to see if an attribute named MsgOut exists in the request. If it does exist, then the code will append " : SimpleFilter" ; otherwise, MsgOut will be set to "SimpleFilter" (without the leading colon). The first time the filter (processor) operates on a request, the attached MsgOut attribute will be set to "SimpleFilter". After the second time, it will contain "SimpleFilter :SimpleFilter", and after the third time, "SimpleFilter :SimpleFilter :SimpleFilter".

Eventually, the MsgOut attribute will be displayed on an HTML page, so you can see the number of times that this filter has operated on the specific request as it traveled through the pipeline. You'll find the code in the

com.apress.projsp20.ch10.SimpleFilter class:

```
package com.apress.projsp20.ch10;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public final class SimpleFilter implements Filter {
private FilterConfig filterConfig = null;
public void doFilter(ServletRequest request, ServletResponse response,FilterChain chain)throws
IOException, ServletException {
```

```
    if (filterConfig == null) return;
    filterConfig.getServletContext().log("in SimpleFilter");
    Object curVal = request.getAttribute("MsgOut");
    if (curVal == null) {
    request.setAttribute("MsgOut", new String("SimpleFilter"));
        } else {
    request.setAttribute("MsgOut", (String) curVal + " :SimpleFilter");
        }

  chain.doFilter(request, response);
  filterConfig.getServletContext().log("leaving SimpleFilter");
    }
```
You'll also revamp your FindProd.jsp file (the renderer processor in your pipeline) to display the new attribute. JSP 2.0's EL is great for creating renderers—that is, JSP pages that render HTML from attached attribute values. You'll name this modified file Sub.jsp:

```
<html>
  <head></head>
  <body>
    <h1>You have accessed this page from the ${param.DEPT} department!</h1>
    ${param.MsgForwarder}<br/>
    ${requestScope.MsgOut}
  </body>
</html>
```

The MsgOut attribute, attached to the request, is displayed using EL here. Note that a parameter called MsgForwarder is also displayed. You'll see the use of this parameter later when you work with the forward()
action of the request dispatcher.

Default REQUEST-Only Filtering: Servlet 2.3 Compatibility

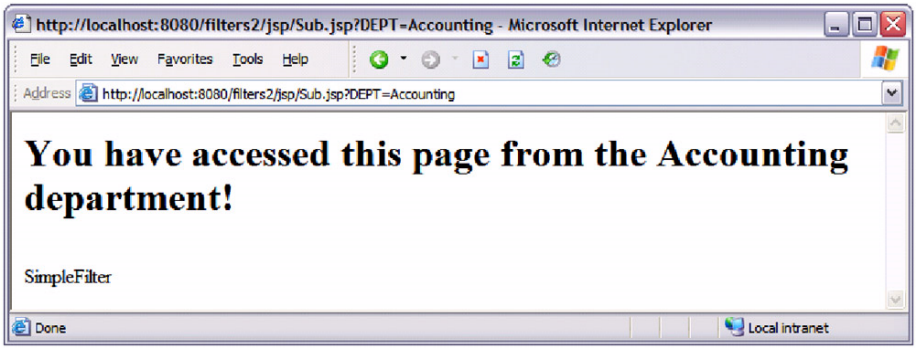
To specify that filtering is to be performed only for requests that come directly from outside of the container, you can add the following <dispatcher> element inside the filter's <filter-mapping> element:

```
<dispatcher>REQUEST</dispatcher>
```

For example, you might have the following in web.xml:

```
<filter>
  <filter-name>Simple Push Filter</filter-name>
  <filter-class>com.apress.projsp20.ch10.SimpleFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>Simple Push Filter</filter-name>
  <url-pattern>/jsp/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

Now, if you open the URL http://localhost:8080/filters2/jsp/Sub.jsp?DEPT=Accounting, you'll see a page like this:



Note the EL rendering of the MsgOut attribute on the page—you know that the filter has been called once. This behavior is all that you have with filter support prior to the Servlet 2.4 standard. It's also the default behavior with Tomcat 5 if you don't specify any <dispatcher> subelement for compatibility reasons.

INCLUDE-Only Filtering

With Tomcat 5 and Servlet 2.4, you can specify that your filter only work on included requests. Try this out by changing web.xml to this:

```
    <filter-mapping>
      <filter-name>Simple Push Filter</filter-name>
      <url-pattern>/jsp/*</url-pattern>
      <dispatcher>INCLUDE</dispatcher>
    </filter-mapping>
```

Restart Tomcat and try accessing Sub.jsp directly via the following URL:

http://localhost:8080/filters2/jsp/Sub.jsp?DEPT=Accounting

Notice that the MsgOut attribute isn't present, indicating that the resulting page hasn't passed through SimpleFilter.

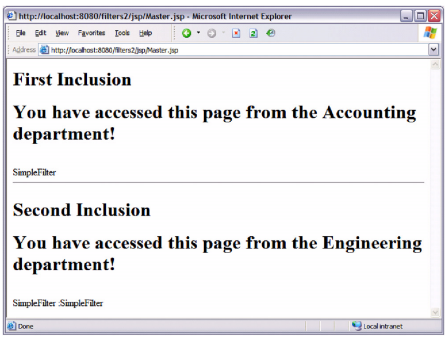
By specifying INCLUDE in the <dispatcher> subelement, you're saying that the filter should map only to included requests. To see this in action, take a look at the jsp/Master.jsp file:

```
<html>
  <head></head>
  <body>

    <h1>First Inclusion</h1>
    <jsp:include page="/jsp/Sub.jsp" flush="true">
      <jsp:param name="DEPT" value="Accounting"/>
    </jsp:include>
    <hr/>
```

```
        <h1>Second Inclusion</h1>
        <jsp:include page="/jsp/Sub.jsp" flush="true">
          <jsp:param name="DEPT" value="Engineering"/>
        </jsp:include>

    </body>
</html>
```

This JSP file makes two calls to the request dispatcher, each time to include the /jsp/Sub.jsp that you've been working with. The output from both of these include() actions is merged as the output response. Note that you're also supplying a different DEPT parameter with each inclusion.

Now navigate to http://localhost:8080/filters2/jsp/Master.jsp, and your resulting page should match the following:



Note the MsgOut attribute value–it reflects the action of SimpleFilter. The attribute is "SimpleFilter" after the first inclusion of Sub.jsp and "SimpleFilter : SimpleFilter" after the second inclusion. The request has been operated on by SimpleFilter twice, once for each inclusion in Master.jsp.

FORWARD-Only Filtering

To make the filter operate only on forwarded requests, modify the web.xml file as follows:

```
        <filter-mapping>
          <filter-name>Simple Push Filter</filter-name>
          <url-pattern>/jsp/*</url-pattern>
          <dispatcher>FORWARD</dispatcher>
        </filter-mapping>
```

Because FORWARD is specified in the <disptacher> element, only forwarded requests will be passed to the filter. To see this in action, look at the jsp/Forwarder.jsp file:

```
    <jsp:forward page="/jsp/Sub.jsp">
      <jsp:param name="DEPT" value="Accounting"/>
      <jsp:param name="MsgForwarder" value="Forwarded from forwarder.jsp"/>
    </jsp:forward>
```

This JSP page simply forwards the request to Sub.jsp, and sets the DEPT and MsgForwarder parameters. Both parameters are displayed in Sub.jsp using EL.

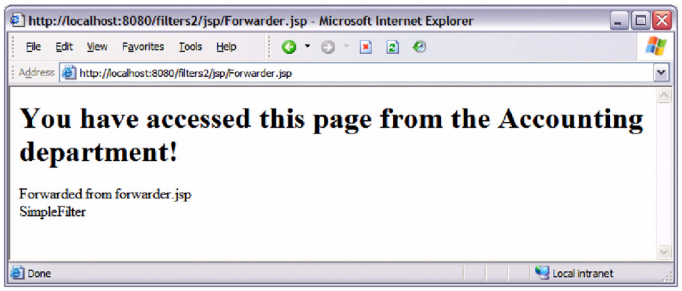
Reload the application or restart Tomcat. Now try the following URLs in turn:

http://localhost:8080/filters2/jsp/Sub.jsp?DEPT=Accounting

http://localhost:8080/filters2/jsp/Master.jsp

http://localhost:8080/filters2/jsp/Forwarder.jsp

For the first two URLs, notice that direct request and included requests are no longer being filtered. The third URL should result in a page similar to this:



You can see from this experiment that only the forwarded request is filtered.

Combining Dispatcher Actions

Of course, you can use more than one <dispatcher> element to indicate multiple locations for the filter to apply to. For example:

```
<filter-mapping>
  <filter-name>Simple Push Filter</filter-name>
  <url-pattern>/jsp/*</url-pattern>
  <dispatcher>FORWARD</dispatcher>
  <dispatcher>INCLUDE</dispatcher>
  <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

Now, try the three previous URLs, and you'll observe the following:

- URL 1: The filter is active on Sub.jsp when requested alone.
- URL 2: The filter is active on Master.jsp PLUS each of the included Sub.jsps.
- URL 3: The filter is active on Forwarder.jsp PLUS the forwarded Sub.jsp.