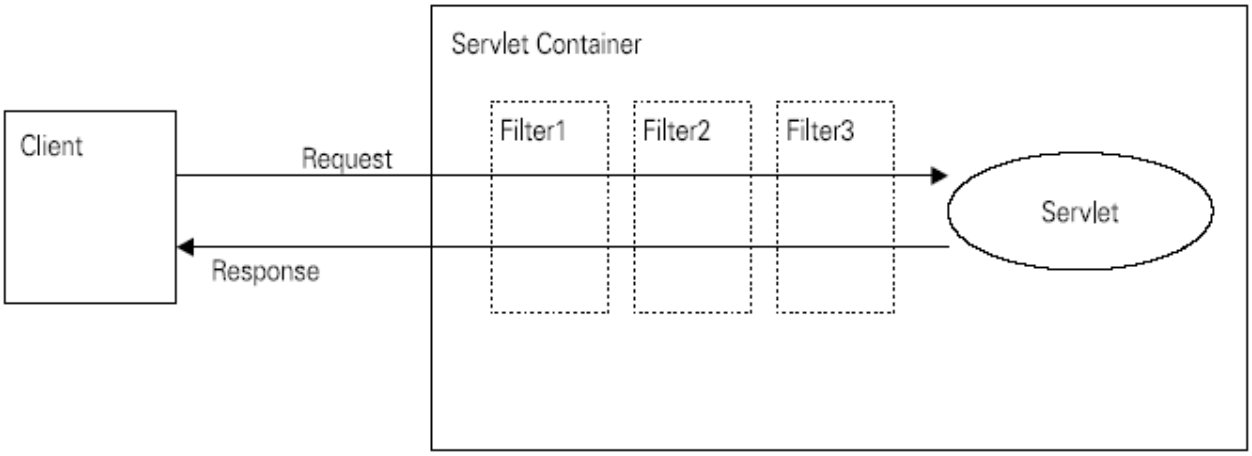
Фильтры

Фильтр – это объект специального вида, который используется для предобработки запросов и постобработки ответов, которые поступают к сервлету от клиента и обратно. Обычно с помощью фильтров реализуют некоторые вспомогательные функции, которые не имеют прямого отношения к бизнес-логике приложения и к функциональности сервлета, и которые хотелось бы при необходимости изменять, никак не затрагивая при этом сервлет. При этом существование фильтра абсолютно прозрачно как для клиента, так и для сервлета. Связывание фильтров с сервлетами выполняется в установочном дескрипторе.

С помощью фильтров часто реализуют следующие функции: контроля доступа (аутентификация), журналирование и аудит, кодирование/декодирование информации (сжатие, шифрование, преобразование формата и проч.), XSLT-преобразование, кэширование.

Фильтры могут быть выстроены в цепочки: один фильтр может участвовать во множестве цепочек. Перед сервлетом может быть выстроена только одна цепочка фильтров. При наличии цепочки фильтров перед сервлетом, запрос, относящийся к этому сервлету, сперва последовательно проходит обработку фильтрами от первого до последнего, и только потом попадает к сервлету. Генерируемый сервлетом ответ перед тем, как быть переданным клиенту, проходит цепочку фильтров в обратном направлении. Фильтр может принять решение самостоятельно обработать запрос и сгенерировать ответ, не передавая запрос далее по цепочке.



С точки зрения реализации фильтр – это класс, который реализует интерфейс `javax.servlet.Filter`, в котором всего три метода: `init()`, `destroy()` и `doFilter()`. Объекты фильтров, так же как и сервлетов, создаются и уничтожаются сервером приложений автоматически (часто при запуске/останове сервера приложений).

Метод `init()` вызывается сразу после создания объекта фильтра сервером приложений для его инициализации. Этот метод для фильтров играет ту же роль, что метод `Servlet.init()` для сервлетов.

Метод `destroy()` вызывается непосредственно перед уничтожением экземпляра фильтра. Этот метод для фильтров играет ту же роль, что метод `Servlet.destroy()` для сервлетов.

Метод `doFilter()` вызывается собственно для предобработки запроса, поступившего к сервлету, с которым связан экземпляр фильтра, и постобработки ответа, сгенерированного этим сервлетом.

На этапе инициализации экземпляра фильтра сервер приложений передает ему объект типа `javax.servlet.FilterConfig` как параметр метода `init()`. Объект типа `FilterConfig` для фильтров играет ту же роль, что и объект типа `ServletConfig` для

сервлетов: через этот объект сервлет может получить доступ к конфигурационным параметрам, указанным для фильтра в установочном дескрипторе, а также к объектной инфраструктуре сервлета.

Интерфейс `FilterConfig` содержит методы `getInitParameterNames()`, `getInitParameter()` и `getServletContext()`, которые аналогичны одноименным методам интерфейса `ServletConfig`.

У метода `doFilter()` три параметра: первые два представляют собой запрос (`ServletRequest`) и ответ (`ServletResponse`), они аналогичны параметрам метода `Servlet.service()`. Третий параметр – объект типа `javax.servlet.FilterChain`, который используется для передачи запроса на обработку далее по цепочке. Необходимость в этом объекте обусловлена тем, то ни фильтр, ни сервлет ничего не знают о наличии друг друга, а также о структуре цепочек фильтров. Собственно передача запроса далее по цепочке (следующему фильтру или сервлету, если данный фильтр в цепочке последний) выполняется методом `FilterChain.doFilter()`, в качестве параметров которому передаются запрос и ответ.

Для каждого фильтра сервер приложений поддерживает только один экземпляр вне зависимости от того, в скольких цепочках он участвует и со сколькими сервлетами связан, а это означает, что экземпляры фильтров работают в мультипоточном режиме.

В некоторых случаях фильтр может самостоятельно сформировать ответ на запрос. При этом метод `FilterChain.doFilter()` не вызывается, а запрос не достигает целевого сервлета. Такое поведение имеет смысл, если в процессе предобработки запроса произошла ошибка.

Стандартная библиотека платформы Java EE не предусматривает какого-либо базового класса для фильтров, аналогичного `GenericServlet` или `HttpServlet`, которые используются для сервлетов. При необходимости такие базовые классы разрабатываются в рамках конкретных проектов.

Если с используемым для обработки ошибок сервлетом связана цепочка фильтров, то при конфигурировании этой цепочки можно указать, будет ли она задействована в случае перенаправления на этот сервлет запросов в результате возникновения ошибки в другом веб-компоненте или нет. Также можно указать, будет ли цепочка фильтров задействована при перенаправлении запросов методами `RequestDispatcher.forward()` или `RequestDispatcher.include()` на сервлет или нет.

В некоторых ситуациях фильтр может выполнять роль контроллера вместо сервлета. Рассмотрим пример, когда необходимо по запросу сформировать представление одних и тех же данных в разных форматах, и для вывода каждого формата (XML, HTML, простой текст) создается отдельная JSP-страница. Если для выбора формата представления в сервлет передается некоторый параметр, то при добавлении JSP-страницы, формирующей представление в новом формате, может потребоваться изменить исходный код сервлета. При использовании фильтра количество форматов представления данных можно увеличивать, не затрагивая бизнес-логику, что является его преимуществом перед сервлетом.

Пример фильтра

Фильтр `AuthFilter` предназначен для веб-приложения, использующего подробную информацию о пользователях. Объект пользователя после успешной аутентификации должен сохраняться в контексте сессии, что и выполняет фильтр `AuthFilter`. При настройке веб-приложения следует указать использование данного фильтра для обработки всех запросов.

```
public class AuthFilter implements Filter {
    private FilterConfig filterConfig = null;
```

```
    public AuthFilter() {
    }

    public void destroy() {
    }

    public void init(FilterConfig filterConfig) {
        this.filterConfig = filterConfig;
    }

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

        if (((HttpServletRequest) request).getSession()
                .getAttribute("current_user") == null) {
            String username = ((HttpServletRequest) request).getRemoteUser();
            if (username != null) {
                User user = new UserDAO().findUser(username);
                ((HttpServletRequest) request).getSession()
                    .setAttribute("current_user", user);
            }
        }

        chain.doFilter(request, response);
    }
}
```

Интерфейсы прослушивания событий.

Эти интерфейсы позволяют реализующим их объектам получать уведомления от сервера приложений о возникновении тех или иных событий. Уведомление оформляется вызовом метода соответствующего интерфейса у реализующих его объектов.

Методу обработки события сервер приложений передает параметры, содержащие информацию о событии. Для обработки событий того или иного вида необходимо написать класс, реализующий соответствующий интерфейс прослушивания событий, поместить этот класс в веб-модуль и описать его определенным образом в установочном дескрипторе.

В пакете `javax.servlet` определено четыре интерфейса прослушивания событий: `ServletContextListener`, `ServletContextAttributeListener`, `ServletRequestListener` и `ServletRequestAttributeListener`.

Интерфейс `ServletContextListener` позволяет отслеживать события, связанные с созданием и уничтожением веб-контекста (объекта типа `ServletContext`). Для этого предусмотрены метод `contextInitialized()`, который вызывается при создании веб-контекста еще до того, как инициализируется любой из веб-компонентов или фильтров веб-приложения, и метод `contextDestroyed()`, который вызывается при уничтожении веб-контекста, то есть когда сервер «завершает работу» веб-приложения, после чего его компоненты становятся недоступны клиенту. Обоим методам в качестве параметра передается объект типа `ServletContextEvent`, из которого методом `getServletContext()` можно получить объект того самого веб-контекста, с которым связано возникшее событие.

Интерфейс `ServletContextAttributeListener` позволяет отслеживать события, связанные с изменением атрибутов контекста веб-приложения. Для этого предусмотрены методы `attributeAdded()`, вызываемый при добавлении атрибута в веб-контекст, `attributeRemoved()` – при удалении атрибута из веб-контекста, и `attributeReplaced()` – при изменении значения атрибута. В качестве параметра этим методам передается объект типа `ServletContextAttributeEvent`, из которого методами `getName()` и `getValue()`

можно получить имя и значение атрибута, с которым связано соответствующее событие. Если метод `getValue()` вызывается при обработке изменения значения атрибута, то возвращается его значение до изменения.

Интерфейс `ServletRequestListener` позволяет отследить моменты начала и окончания обработки запроса веб-приложением. Момент начала обработки – это момент поступления запроса к любому из веб-компонентов. Момент окончания – это момент завершения обработки последним вовлеченным в обработку компонентом или фильтром. В данном интерфейсе предусмотрены методы `requestInitialized()` для отслеживания начала обработки запроса и `requestDestoyed()` для отслеживания окончания обработки запроса. В качестве параметра обоим методам передается объект типа `ServletRequestEvent`, из которого методами `getServletContext()` и `getServletRequest()` можно получить объект веб-контекста приложения, которому был направлен запрос, и собственно объект запроса, с которым связаны эти события.

Интерфейс `ServletRequestAttributeListener` позволяет отслеживать события, связанные с изменением атрибутов запроса. Он аналогичен интерфейсу `ServletcontextAttributeListener` и содержит методы `attributeAdded()`, `attributeRemoved()` и `attributeReplaced()`. В качестве параметра этим методам передается объект типа `ServletRequestAttributeEvent`, аналогичный объекту типа `ServletContextAttributeEvent`.

В пакете `javax.servlet.http` определено четыре интерфейса прослушивания: `HttpSessionListener`, `HttpSessionAttributeListener`, `HttpSessionBindingListener` и `HttpSessionActivationListener`.

Интерфейс `HttpSessionListener` позволяет отследить моменты начала и окончания сеанса работы пользователя с веб-приложением. Здесь предусмотрены методы `sessionCreated()`, вызываемый в момент начала сеанса, и `sessionDestroyed()` – в момент окончания сеанса. Обоим методам передается объект типа `HttpSessionEvent`, из которого методом `getSession()` можно получить тот объект веб-сессии (`HttpSession`) , с которым связано событие. Интерфейс `HttpSessionListener` для сессии играет ту же роль, что и `ServletContextListener` для веб-контекста.

Интерфейс `HttpSessionAttributeListener` позволяет отслеживать события, связанные с изменением атрибутов веб-сессии. Он аналогичен интерфейсу `ServletContextAttributeListener` и содержит методы `attributeAdded()`, `attributeRemoved()` и `attributeReplaced()`. В качестве параметра этим методам передается объект типа `HttpSessionBindingEvent`, из которого методами `getSession()`, `getName()` и `getValue()` можно получить объект веб-сессии (`HttpSession`), а также имя и значение атрибута, с которым связано соответствующее событие.

Интерфейс `HttpSessionBindingListener` реализуется классами, объекты которых используются как значения атрибутов веб-сессии. Эти объекты могут отследить моменты, когда они помещаются или удаляются из сессии. При помещении в сессию у этих объектов вызывается метод `valueBound()`, а при удалении – метод `valueUnbound()`. В качестве параметра этим методам передается объект типа `HttpSessionBindingEvent`.

Интерфейс `HttpSessionActivationListener` используется для отслеживания событий, связанных с активацией и деактивацией объекта веб-сессии. Активация и деактивация может выполняться сервером приложений для эффективного управления ресурсами при большой нагрузке на сервер. При деактивации состояние сессии сбрасывается в постоянное хранилище (на диск), а объект уничтожается. Активация – это обратный процесс, когда состояние сессии восстанавливается из постоянного хранилища. Здесь предусмотрены методы `sessionWillPassivate()`, вызываемый непосредственно перед деактивацией сессии, и `sessionDidActivate()`, вызываемый сразу после активации сессии. Обоим методам в качестве параметра передается объект типа `HttpSessionEvent()`.