Java Technologies
Web Filters

The Context

- Upon receipt of a request, various processings may be needed:
  - Is the user authenticated?
  - Is there a valid session in progress?
  - Is the IP trusted, is the user's agent supported, ...?
- When sending a response, the result may require various processings:
  - Add some additional design elements.
  - Trim whitespaces, etc.

Example

In the login controller:

```
User user = new User();
user.setName(request.getParameter("userName"));
user.setPassword(request.getParameter("userPassword"));
session.setAttribute("user", user);
```

In <u>every</u> web component that requires a valid user:

```
User user = (User) session.getAttribute("user");
if (user == null) {
  response.sendRedirect("login.jsp");
  return;
}
// ok, we have a user in the session
// ...
```

crosscutting concern

The Concept of Filters

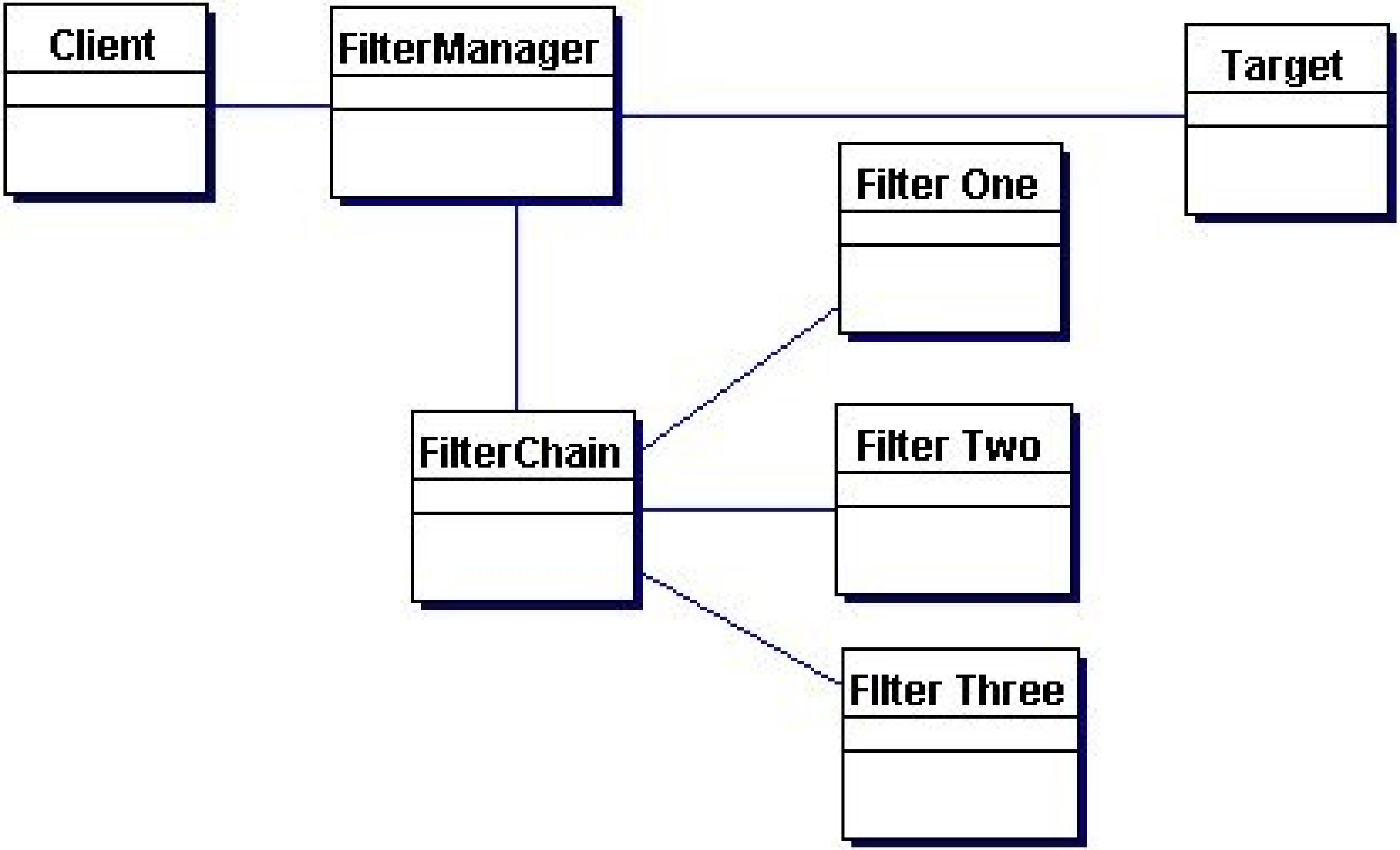
We need a component that:

- Dynamically intercepts requests and responses
  - preprocessing / postprocessing
- Provides reusable functionalities that can be "attached" to any kind of web resource
- Can be used declarative, in a plug-in manner
- Is (usually) independent (does not have any dependencies on other web resource for which it is acting as a filter)

Common Usages

- Authentication
- Logging and auditing
- Image conversion, scaling, etc.
- Data compression, encryption, etc.
- Localization
- Content transformations (for example, XSLT)
- Caching
- ...

Intercepting Filter Design Pattern



Java EE Filter Architecture

- An API for <u>creating</u> the filters
  - *javax.servlet.Filter* interface

- A method for <u>configuring</u> and <u>plugging-in</u> the filters (mapping them to other resources)
  - *declarative* (in web.xml or using @WebFilter)

- A mechanism for <u>chaining</u> the filters
  - *javax.servlet.FilterChain*

javax.servlet.Filter interface

```
public interface Filter() {
  /**
  * Called by the web container to indicate to a filter
  * that it is being placed into service. */
  void init(FilterConfig filterConfig);

  /**
  * The doFilter method of the Filter is called by the container
  * each time a request/response pair is passed through the chain
  * due to a client request for a resource at the end of the chain */
  void doFilter(ServletRequest request,
                ServletResponse response,
                FilterChain chain);

  void destroy();
}
```

Example: Logging

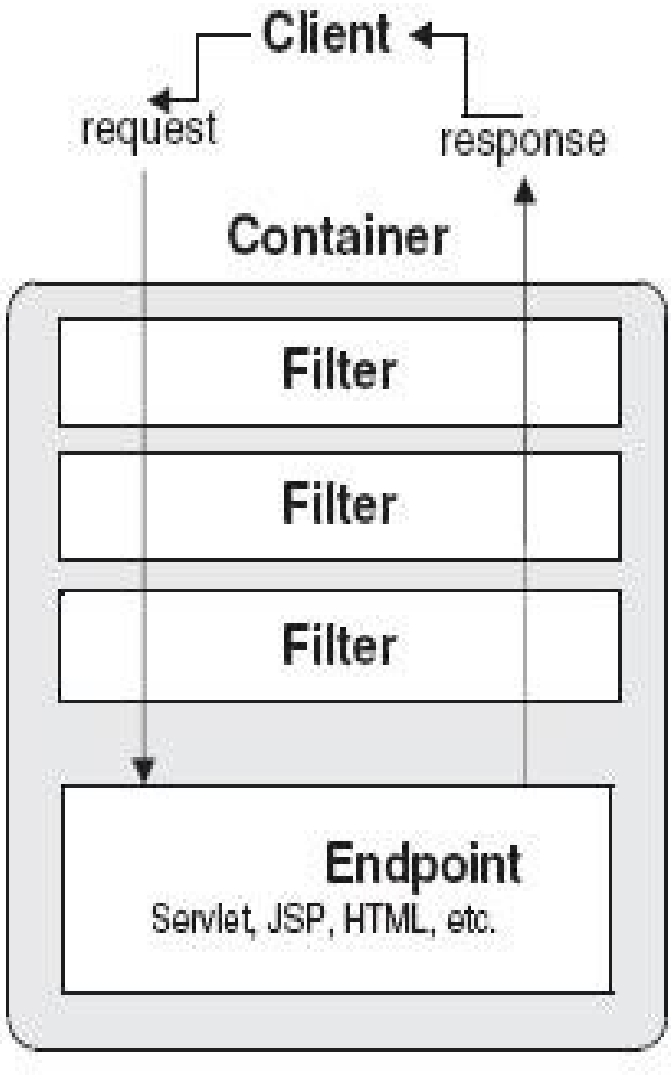
```
@WebFilter(urlPatterns = {"/*"})
public class LogFilter implements Filter {

  public void doFilter(ServletRequest req, ServletResponse res,
                       FilterChain chain)
                       throws IOException, ServletException {
    HttpServletRequest request = (HttpServletRequest) req;

    // Find the IP of the request
    String ipAddress = request.getRemoteAddr();

    // Write something in the log
    System.out.println(
       "IP: " + ipAddress + ", Time: " + new Date().toString());

    chain.doFilter(req, res);
  }
}
```

Example: Character Encoding

```
public void init(FilterConfig filterConfig) throws ServletException {
   //read the character encoding from a filter initialization parameter
   this.encoding = filterConfig.getInitParameter("encoding");
   // for example: UTF-8 or ISO 8859-16 or Windows-1250 etc.
}

public void doFilter(ServletRequest request,
                     ServletResponse response, FilterChain chain)
                     throws IOException, ServletException {
   if (encoding != null) {
     //useful if the browser does not send character encoding information
     //in the Content-Type header of an HTTP request
     request.setCharacterEncoding(encoding);
   }
   chain.doFilter(request, response);
}
```

You may want to read: “The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)” by Joel Spolsky

javax.servlet.FilterChain interface

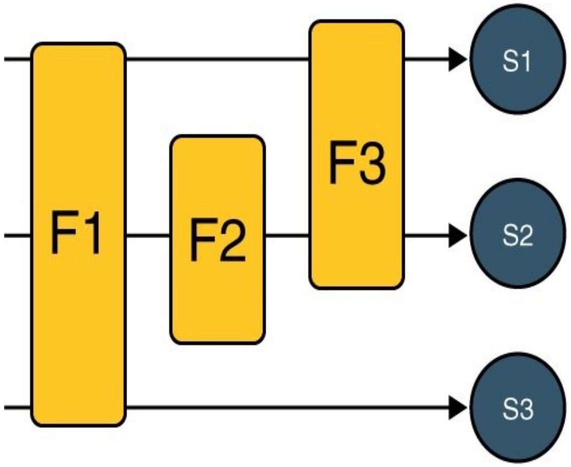


```
public interface FilterChain() {

  void doFilter(
    ServletRequest request,
    ServletResponse response);

}
```

Specifying Filter Mappings

web.xml

```
<filter>
   <filter-name>HelloFilter</filter-name>
   <filter-class>somepackage.HelloFilterImpl</filter-class>
   <init-param>
      <param-name>greeting</param-name>
      <param-value>Hello World!</param-value>
   </init-param>
</filter>
<filter-mapping>
   <filter-name>HelloFilter</filter-name>
   <url-pattern>/hello/*</url-pattern>
</filter-mapping>
```



many-to-many

```
@WebFilter(
  filterName = "HelloFilter",
  urlPatterns = {"/hello/*"},
  initParams = {
   @WebInitParam(greeting = "Hello World!")}
)
public class HelloFilterImpl implements Filter {
 …
}
```

The generic structure of a filter

```java
public class GenericFilter implements Filter {
  public void doFilter(ServletRequest request, ServletResponse response,
                       FilterChain chain)
                       throws IOException, ServletException {
    doBeforeProcessing(request, response);
    Throwable problem = null;
    try {
      chain.doFilter(request, response);
    } catch(Throwable t) {
      problem = t;
    }

    doAfterProcessing(request, response);
    if (problem != null) {
      processError(problem, response);
    }
  }
...
}
```

Example: Count and Measure

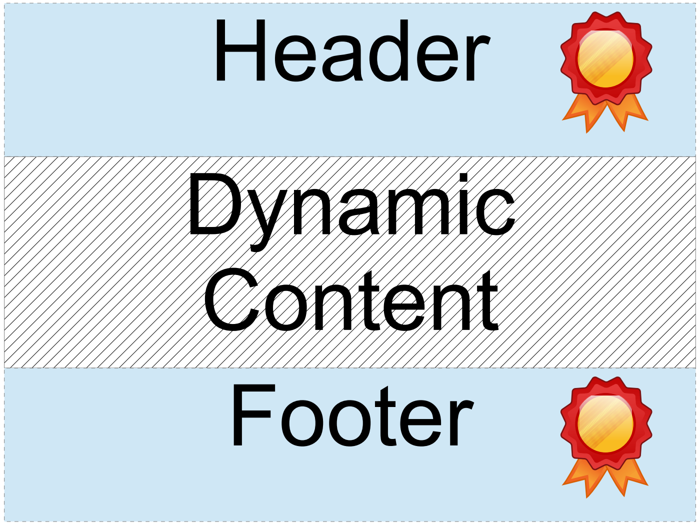
```
@WebFilter(urlPatterns = {"/someComponent"})
public class ResponeTimeFilter implements Filter {
  private AtomicInteger counter = new AtomicInteger();

  public void doFilter(ServletRequest req, ServletResponse res,
                       FilterChain chain)
                       throws IOException, ServletException {
    // Count the requests
    int n = counter.addAndGet(1);

    // Start the timer
    long t0 = System.currentTimeMillis();

    chain.doFilter(req, res);

    // Stop the timer
    long t1 = System.currentTimeMillis();

    app.log("Request " + n + " took " + (t1 - t0) + "ms");
  }
}
```

Filtering the response

The Problem:

Modify the content of the response

- chain.doFilter(
  request, response)
- response
  - getOutputStream
  - getWriter



Decorator Design Pattern

- You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.
- *Decorator Design Pattern*: Attach additional responsibilities to an object dynamically, without altering its structure (class signature).
- *Wrapper*

Decorator example: Java IO

```
public interface Reader {
  int read();
}

public class FileReader implements Reader {
  public int read() { ... }
}

public class BufferedReader implements Reader {
  private FileReader in;
  public BufferedReader(FileReader in} {
    this.in = in;            //receive the original object
  }
  public int read() {
    return in.read();        // inherit old functionality
  }
  public String readLine() { // create new functionality
  ...
  }
}
```

```
Reader original = new FileReader("someFile");

Reader decorated = new BufferedReader(reader);
```

HTTP Wrappers

- Decorating the request
  - **HttpServletRequestWrapper**
  - *implements HttpServletRequest*

```
ServletRequestWrapper wrapper = new HttpServletRequestWrapper(req) {
     @Override
     public String getLocalName() {
         return "localhost";
     }
 };
 chain.doFilter(wrapper, response);
```

- Decorating the response
  - **HttpServletResponseWrapper**
  - *implements HttpServletResponse*

Creating a Response Wrapper

```
public class SimpleResponseWrapper
                           extends HttpServletResponseWrapper {

    private final StringWriter output;

    public SimpleResponseWrapper(HttpServletResponse response) {
        super(response);
        output = new StringWriter();
    }

    @Override
    public PrintWriter getWriter() {
        // Hide the original writer
        return new PrintWriter(output);
    }

    @Override
    public String toString() {
        return output.toString();
    }
}
```

Decorating the response

```
@WebFilter(filterName = "ResponseDecorator", urlPatterns = {"/*"})
public class ResponseDecorator implements Filter {

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

    SimpleResponseWrapper wrapper
            = new SimpleResponseWrapper((HttpServletResponse) response);

    //Send the decorated object as a replacement for the original response
    chain.doFilter(request, wrapper);

    //Get the dynamically generated content from the decorator
    String content = wrapper.toString();

    // Modify the content
    content += "<p> Mulțumim!";

    //Send the modified content using the original response
    PrintWriter out = response.getWriter();
    out.write(content);
  }
  ...
}
```

Conclusions

The *filter mechanism* provides a way to encapsulate common functionality in a component that can reused in many different contexts.

Filters are easy to write and configure as well as being portable and reusable.

Java EE™

Java Technologies
Web Listeners

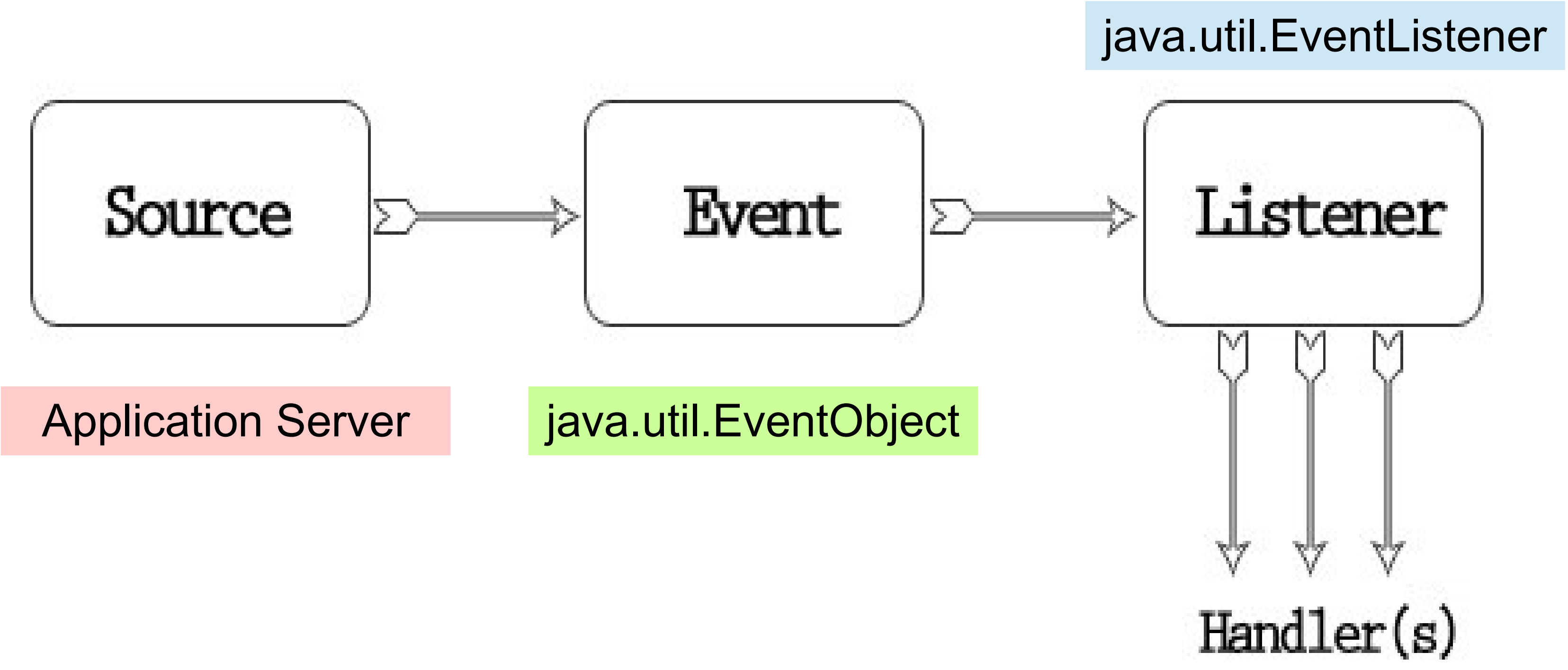
The Context

- Web Applications have a **life cycle:**
  - *they are deployed to a server and initialized*
  - *they receive requests, create sessions*
  - *they are destroyed*
- **The application server** manages that life cycle
- What if we want to :
  - set an attribute in the application scope at initialization time?
  - create a database connection whenever a client starts a session? etc.

The Concept of Listeners

- **Observe and respond to key events:**
  - Lifecycle changes
  - Attribute changes
  - ...Not only incoming requests
- Provide reusable functionalities that can be "attached" to any application
- Can be used declarative, in a plug-in manner
- More efficient resource management and automated processing based on event status.

Event Driven Programming



java.util.EventListener

Application Server

java.util.EventObject

Event-Driving Programming Model

Example: *ServletContextListener*

```
@WebListener()
public class AppListener implements ServletContextListener {
  private static long startupTime = 0L;

  /* Application Startup Event */
  public void contextInitialized(ServletContextEvent ce) {
    startupTime = System.currentTimeMillis();
  }

  /* Application Shutdown Event */
  public void contextDestroyed(ServletContextEvent ce) {}

  public static Date getStartupTime() {
    return startupTime;
  }
}
```

Example: *HttpSessionListener*

```
@WebListener()
public class SessionCounter implements HttpSessionListener {
  private static int users = 0;

  /* Session Creation Event */
  public void sessionCreated(HttpSessionEvent httpSessionEvent) {
    users ++;
  }

  /* Session Invalidation Event */
  public void sessionDestroyed(HttpSessionEvent httpSessionEvent) {
    users --;
  }

  public static int getConcurrentUsers() {
    return users;
  }
}
```

“Plugging in” a Web Listener

- web.xml

```
<web-app>
  ...
  <listener>
      <listener-class>
          util.listeners.AppListener
      </listener-class>

      <listener-class>
          util.listeners.SessionCounter
      </listener-class>
  </listener>
  ...
</web-app>
```

- @WebListener() annotation

Listeners

- ServletContextListener
- ServletRequestListener
- HttpSessionListener

- ServletContextAttributeListener
- ServletRequestAttributeListener
- HttpSessionAttributeListener
- HttpSessionBindingListener

- HttpSessionActivationListener

- AsyncListener

Monitoring Session Attributes

Receiving notification events about HttpSession attribute changes:

```
@WebListener()
public class MySessionAttributeListener
    implements HttpSessionAttributeListener {

  public void attributeAdded(HttpSessionBindingEvent event) {
    System.out.println("attribute added: " + event.getValue());
  }

  public void attributeRemoved(HttpSessionBindingEvent event) {
    System.out.println("attribute removed: " + event.getValue());
  }

  public void attributeReplaced(HttpSessionBindingEvent event) {
    System.out.println("attribute replaced: " + event.getValue());
  }
}
```

Monitoring at Object Level

Notifications generated whenever an object is bound to or unbound from a session.

```
public class MyBindingListener implements HttpSessionBindingListener {
  private String data;
  public MyBindingListener(String data) {
    this.data = data;
  }

  public void valueBound(HttpSessionBindingEvent event) {
    System.out.println("hello from object: " + data);
  }

  public void valueUnbound(HttpSessionBindingEvent event) {
    System.out.println("by bye from object: " + data);
  }

  @Override
  public String toString() {
    return data;
  }
}
```

Example

Consider the sequence:

```
<%
session.setAttribute("demo", new demo.MyBindingListener("demo"));
session.removeAttribute("demo");
%>
```

The previous two listeners will display:

```
hello from watched object: test
attribute added: test
by bye from watched object: test
attribute removed: test
```

Session *Passivation* and *Activation*

Passivation is the process of controlling memory usage by removing relatively unused sessions from memory while storing them in persistent storage. Restoring these sessions is called **activation.**

```
public class MyHttpSessionActivationListener

      implements HttpSessionActivationListener {

  public void sessionWillPassivate(HttpSessionEvent se) {
    //cleanup and store something into persistent storage
  }

  public void sessionDidActivate(HttpSessionEvent se) {
    //init and retrieve something from persistent storage
  }
}
```

Asynchronous Processing

- Normally: a **server thread per client request**.
- Heavy load conditions → large amount of threads → running out of memory or exhausting the pool of container threads.
- **Scalable web applications** → no threads associated with a request are sitting idle, so the container can use them to process new requests.
- Common scenarios in which a thread associated with a request can be sitting idle:
  - ➔ the thread needs to **wait for a resource** to become available or process data before building the response (database acces, remote web service)
  - ➔ the thread needs to **wait for an event** before generating the response. (wait for a message, new information from another client, etc)
  - ➔ the thread performs a **long-running operation.**
- Blocking operations limit the scalability of web applications. **Asynchronous processing** refers to assigning these blocking operations to a new thread and returning the thread associated with the request immediately to the container.

Long-Running Servlets

```
@WebServlet("/LongRunningServlet")
public class LongRunningServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException {
    long startTime = System.currentTimeMillis();
    //---------------
    longProcessing();
    //---------------
    long endTime = System.currentTimeMillis();
    //-------------------------------------
    PrintWriter out = response.getWriter();
    out.write("Success!");
    //-------------------------------------
    System.out.println("Time: " + (endTime - startTime) + " ms");
  }
  private void longProcessing() {
    try {
      Thread.sleep(10000); //10 seconds
    } catch (InterruptedException e) { }
  }
}
```

must be performed in a separate thread

must be postponed

Asynchronous Servlets

```
@WebServlet(urlPatterns = "/AsyncLongRunningServlet",
            asyncSupported = true)
public class AsyncLongRunningServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException {
    long startTime = System.currentTimeMillis();
    AsyncContext asyncCtx = request.startAsync();
    asyncCtx.addListener(new AppAsyncListener());
    asyncCtx.setTimeout(20000);
    ThreadPoolExecutor executor = (ThreadPoolExecutor)request
        .getServletContext().getAttribute("executor");
    executor.execute(new AsyncRequestProcessor(asyncCtx));
    long endTime = System.currentTimeMillis();
    System.out.println("Time: " + (endTime - startTime) + " ms");
  }
}
```

monitor the execution

the actual processing

The Request Processing Thread

```
public class AsyncRequestProcessor implements Runnable {
  private AsyncContext asyncContext;
  public AsyncRequestProcessor(AsyncContext asyncCtx) {
    this.asyncContext = asyncCtx;
  }
  public void run() {
    //----------------
    longProcessing();
    //----------------
    try {
      PrintWriter out = asyncContext.getResponse().getWriter();
      out.write("Success!");
    } catch (IOException e) {}
    asyncContext.complete();
  }
  private void longProcessing() {
    try {
      Thread.sleep(10000);
    } catch (InterruptedException e) {}
  }
}
```

Completes the asynchronous operation and closes the response associated with this asynchronous context.

Monitoring the Async Execution

```
@WebListener
public class AppAsyncListener implements AsyncListener {

  public void onStartAsync(AsyncEvent event) throws IOException {
  }
  public void onComplete(AsyncEvent event) throws IOException {
  }
  public void onTimeout(AsyncEvent event) throws IOException {
    System.out.println("AppAsyncListener.onTimeout");
    ServletResponse response =
      event.getAsyncContext().getResponse();
    PrintWriter out = response.getWriter();
    out.write("TimeOut Error in Processing");
  }

  public void onError(AsyncEvent event) throws IOException {
  }
}
```

Creating the ThreadPoolExecutor

```
@WebListener
public class AppContextListener implements ServletContextListener {
  public void contextInitialized(ServletContextEvent servletContextEvent) {
    // create the thread pool
    ThreadPoolExecutor executor =
        new ThreadPoolExecutor(100, 200,
           50000L, TimeUnit.MILLISECONDS,
           new ArrayBlockingQueue<Runnable>(100));
        //int corePoolSize, int maximumPoolSize,
        //long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue
    servletContextEvent.getServletContext()
      .setAttribute("executor",executor);
  }

  public void contextDestroyed(ServletContextEvent servletContextEvent)
    ThreadPoolExecutor executor =
      (ThreadPoolExecutor) servletContextEvent
      .getServletContext().getAttribute("executor");
    executor.shutdown();
  }
}
```