



CORSO DI LAUREA IN INFORMATICA

TECNOLOGIE SOFTWARE

PER IL WEB

SERVLET – II PARTE

a.a 2021/22

Pagine di errore

```
<error-page>
  <error-code>404</error-code>
  <location>/commons/redirectToError.jsp</location>
</error-page>
<error-page>
  <error-code>500</error-code>
  <location>/commons/fatalError.jsp</location>
</error-page>
```

- Si possono gestire anche le eccezioni:

```
<error-page>
  <exception-type>javax.servlet.ServletException</exception-type>
  <location>/error.html</location>
</error-page>
```

- Dalle Servlet 3.0 in poi:

```
<error-page>
  <location>/general-error.html</location>
</error-page>
```

Gestione dello stato (di sessione)

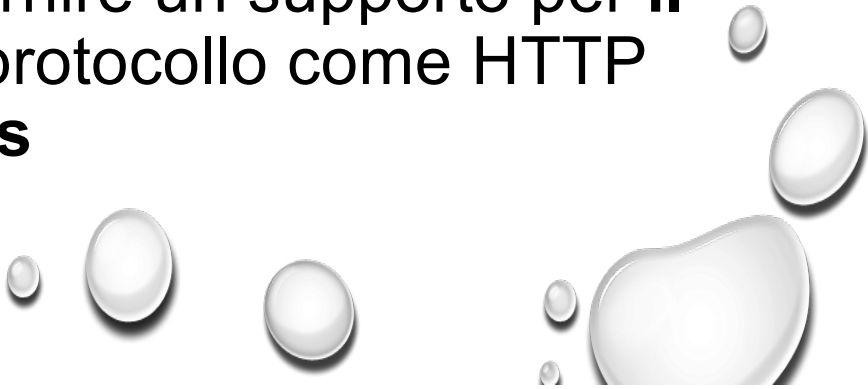
- **HTTP è un protocollo stateless:** non fornisce in modo nativo meccanismi per il mantenimento dello stato tra diverse richieste provenienti dallo stesso client
- Le applicazioni Web hanno spesso bisogno di stato. Sono state definite due tecniche per mantenere traccia delle informazioni di stato:
 1. **uso dei cookie: meccanismo di basso livello**
 2. **uso della sessione (session tracking): meccanismo di alto livello**
- La sessione rappresenta un'utile astrazione ed essa stessa può far ricorso a tre meccanismi base di implementazione:
 1. Cookie
 2. URL rewriting
 3. Hidden form

Session Tracking and E-Commerce

- Perché il monitoraggio della sessione?
- Quando i clienti di un negozio on-line aggiungono un articolo al loro carrello, come fa il server a sapere cosa c'è già nel carrello?
- Quando i clienti decidono di procedere alla cassa, come può il server determinare quale carrello precedentemente creato è il loro?



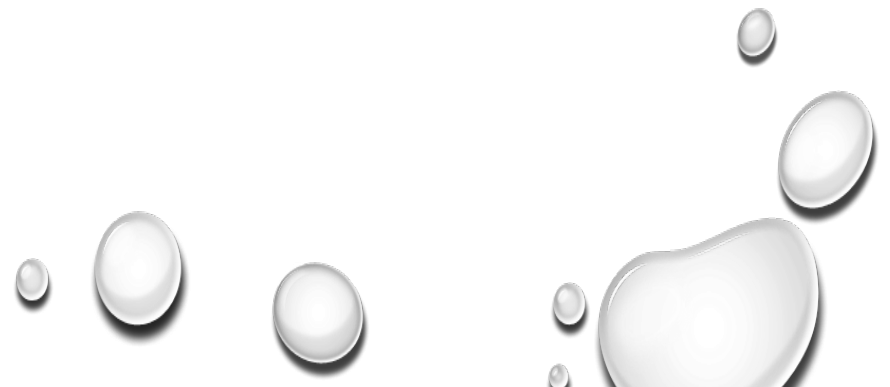
I COOKIE

- Parallelamente alle sequenze request/response, il protocollo prevede una struttura dati che si muove come un token, dal client al server e viceversa: **il cookie**
 - I cookie possono essere generati dall'applicazione lato server (dal programmatore web) e restituiti dal client (il browser)
 - Dopo la loro creazione vengono sempre passati ad ogni trasmissione di request e response
 - Hanno come scopo quello di fornire un supporto per **il mantenimento di stato** in un protocollo come HTTP che è essenzialmente **stateless**
- 

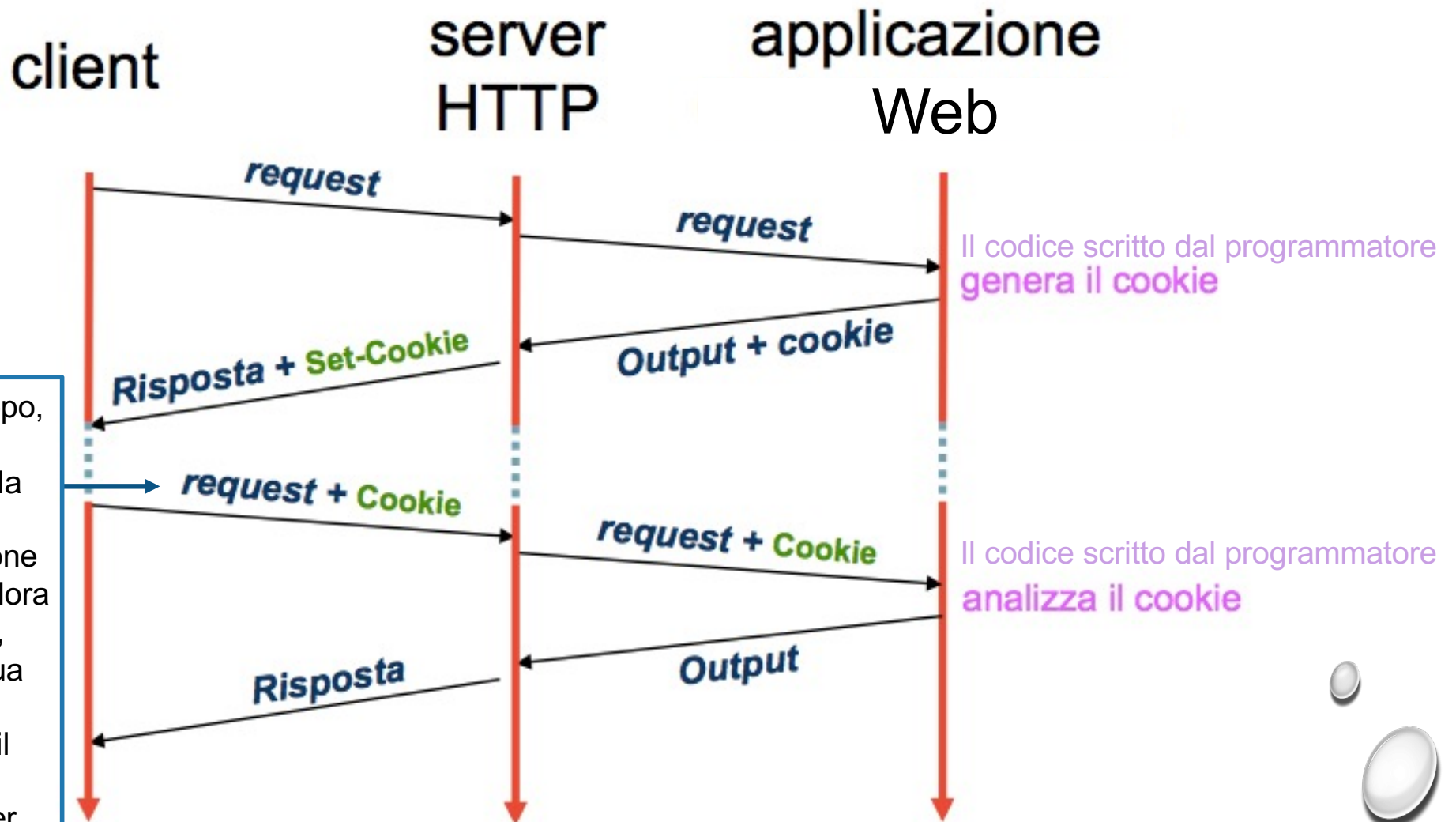


HEADER DEI cookie

- I cookies usano due header, uno nella **risposta**, ed uno nelle **richieste** successive:
 - **Set-Cookie**: header nel messaggio di **HTTP response**, il client può memorizzarlo e rispedirlo alla prossima richiesta
 - **Cookie**: header nel messaggio di **HTTP request**. Il client decide se spedirlo sulla base del nome del documento, dell'indirizzo IP del server, e dell'età del cookie



Interazione con i cookie



Tempo dopo, l'utente accede alla stessa applicazione web ed allora il browser, quasi a sua insaputa, inserisce il cookie nell'header «Cookie» della richiesta

La classe cookie

- Un cookie contiene un certo numero di informazioni, tra cui:
 - una coppia nome/valore
 - Caratteri non utilizzabili: [] () = , " / ? @ : ;
 - il dominio Internet dell'applicazione che ne fa uso
 - path dell'applicazione
 - una expiration date espressa in secondi (-1 indica che il cookie non sarà memorizzato su file associato, 60*60*24 indica 24 ore)
 - un valore booleano per definirne il livello di sicurezza
- La classe Cookie modella il cookie HTTP
 - L'applicazione web recupera i cookie dalla **request** utilizzando il metodo **getCookies()**
 - L'applicazione web aggiunge i cookie alla **response** utilizzando il metodo **addCookie()**

Esempi di uso di cookie

- Con il metodo **setSecure(true)** il client viene forzato a inviare il cookie solo su protocollo sicuro (HTTPS)

creazione

```
Cookie c = new Cookie("MyCookie", "test");  
c.setSecure(true);  
c.setMaxAge(-1);  
c.setPath("/");  
response.addCookie(c);
```

lettura

```
Cookie[] cookies = request.getCookies();  
if(cookies != null)  
{  
    for(int j=0; j<cookies.length(); j++)  
    {  
        Cookie c = cookies[j];  
        out.println("Un cookie: " +  
            c.getName()+"="+c.getValue());  
    }  
}
```

Esempi di uso di cookie (2)

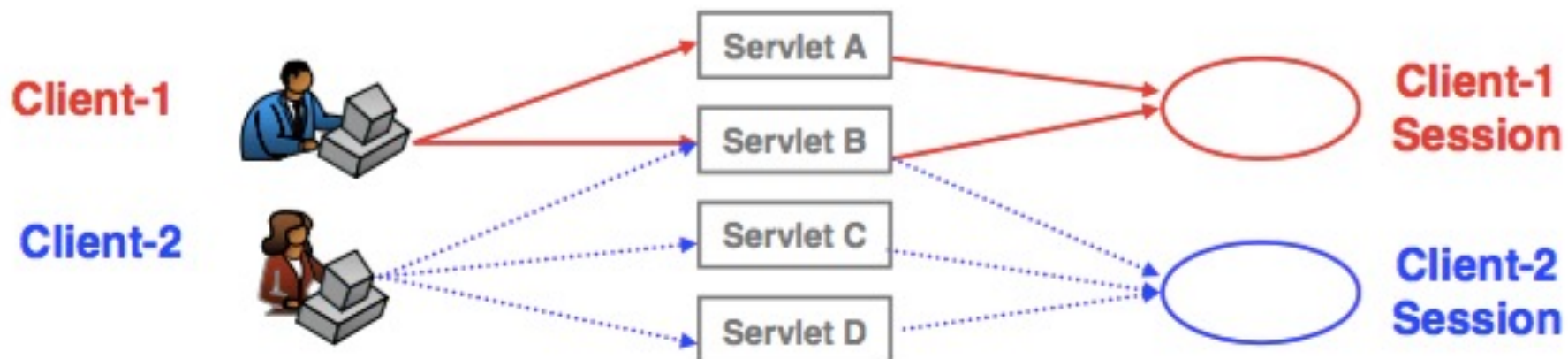
- Per eliminare un cookie è sufficiente seguire i seguenti tre passi:
 - Leggere un cookie già esistente e memorizzarlo nell'oggetto Cookie
 - Impostare l'età del cookie a zero utilizzando il metodo `setMaxAge()`
 - Aggiungere di nuovo questo cookie nell'intestazione della risposta

- Es:

```
Cookie[] cookies = null;  
cookies = request.getCookies();  
Cookie cookie = cookies[i] //i-esimo Cookie  
cookie.setMaxAge(0);  
response.addCookie(cookie);
```

Uso della sessione

- La sessione Web è un'entità gestita dal Web Container
- *È **condivisa** fra tutte le richieste provenienti dallo **stesso client**: consente di mantenere, quindi, informazioni di stato (di sessione)*
- Ogni sessione consiste di un **oggetto** contenente dati di varia natura ed è identificata in modo univoco da un **session ID**
- Viene usata dai componenti di una Web application per mantenere lo stato del client durante le molteplici interazioni dell'utente con la Web application

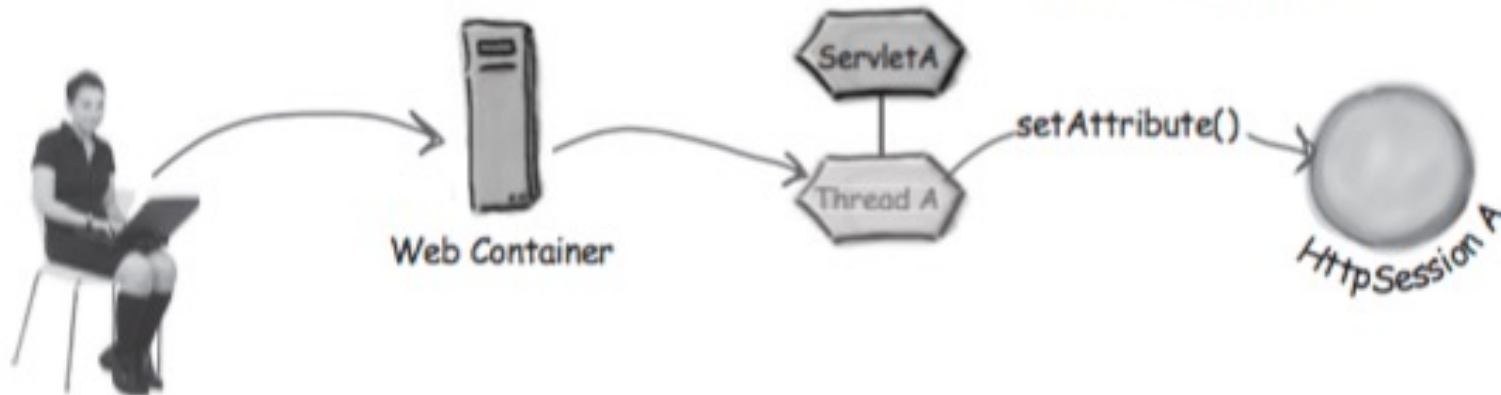


①

Diane selects "Dark" and hits the submit button.

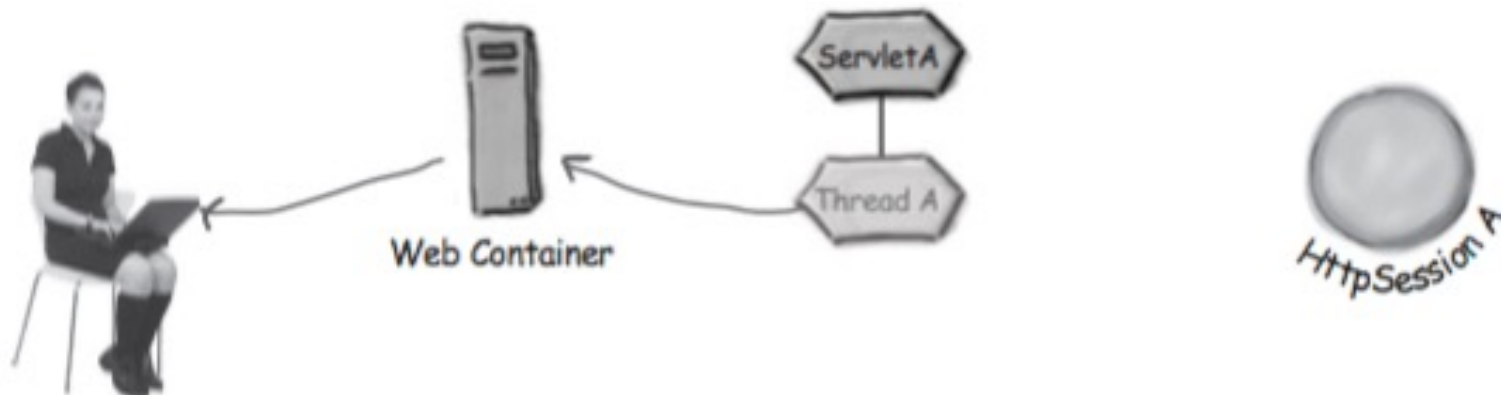
The Container sends the request to a new thread of the BeerApp servlet.

The BeerApp thread finds the session associated with Diane, and stores her choice ("Dark") in the session as an attribute.



②

The servlet runs its business logic (including calls to the model) and returns a response... in this case another question, "What price range?"



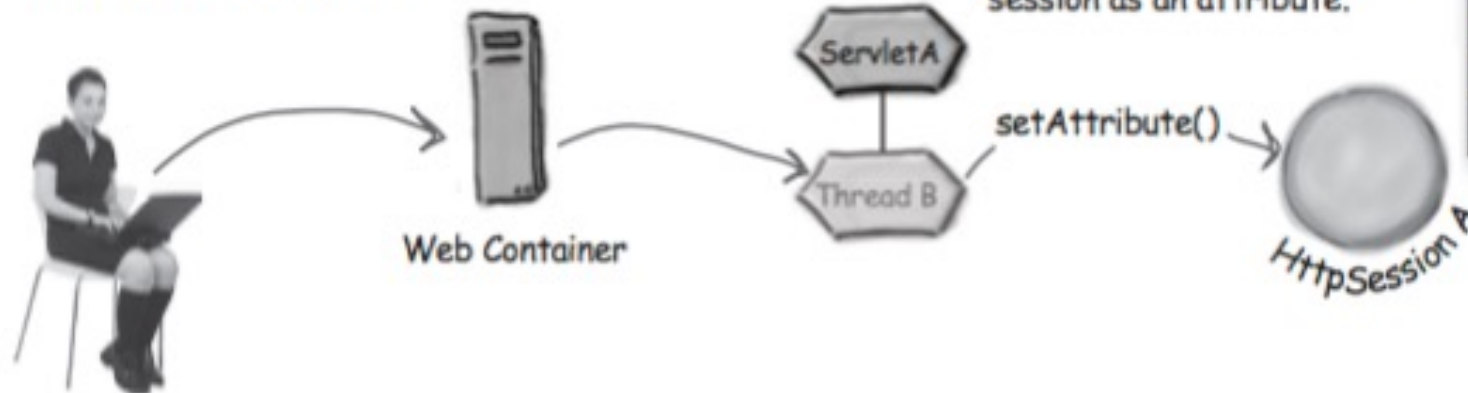
③

Diane considers the new question on the page, selects "Expensive" and hits the submit button.

The Container sends the request to a new thread of the BeerApp servlet.

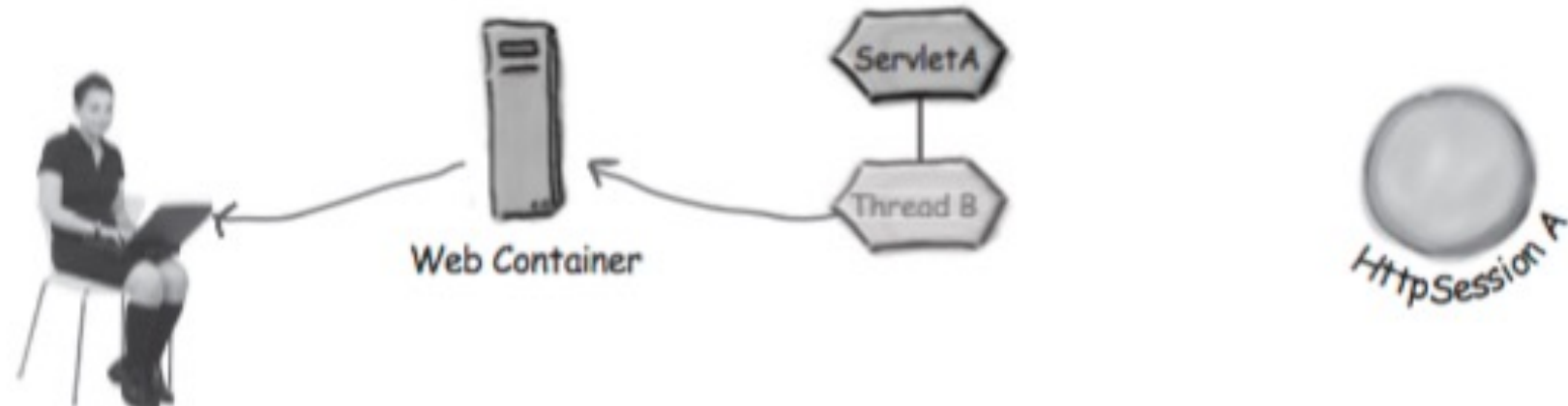
The BeerApp thread finds the session associated with Diane, and stores her new choice ("Expensive") in the session as an attribute.

Same client
Same servlet
Different request
Different thread
Same session



④

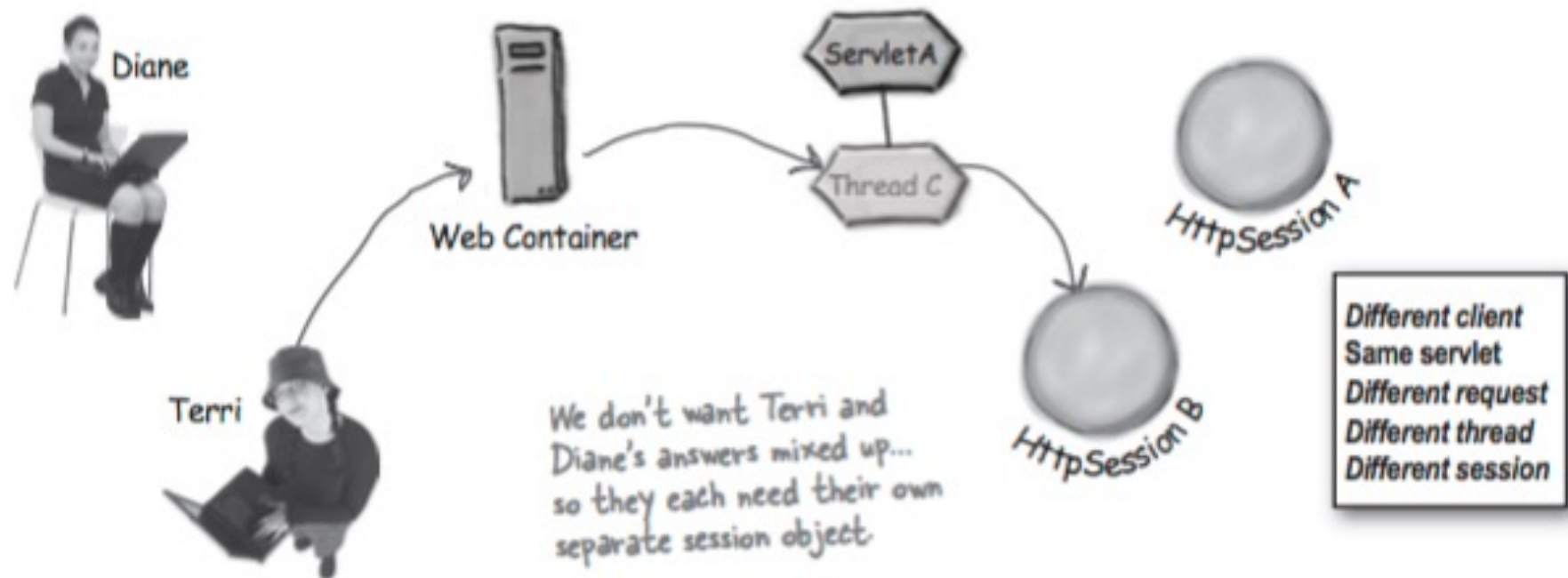
The servlet runs its business logic (including calls to the model) and returns a response... in this case another question.



- ⑤ Diane's session is still active, but meanwhile Terri selects "Pale" and hits the submit button.

The Container sends Terri's request to a new thread of the BeerApp servlet.

The BeerApp thread starts a new Session for Terri, and calls `setAttribute()` to store her choice ("Pale").



Accesso alla sessione

- L'accesso avviene mediante l'interfaccia **HttpSession**
- Per ottenere un riferimento ad un oggetto di tipo HttpSession si usa il metodo **getSession()** dell'interfaccia **HttpServletRequest**

public HttpSession getSession(boolean createNew);

- Valori di createNew:
 - **true**: ritorna la sessione esistente o, se non esiste, ne crea una nuova
 - **false**: ritorna, se possibile, la sessione esistente, altrimenti ritorna null

- Uso del metodo in una Servlet:

HttpSession session = request.getSession(true);

- Per recuperare l'ID della sessione:

```
HttpSession ssn = request.getSession();
if(ssn != null){
    String ssnId = ssn.getId();
    System.out.println("Your session Id is : "+ ssnId);
}
```

Gestione del contenuto di una sessione

- Si possono memorizzare **dati specifici dell'utente negli attributi della sessione** (coppie nome/valore)
- Consentono di memorizzare e recuperare oggetti

```
Cart sc = (Cart)session.getAttribute("shoppingCart");  
sc.addItem(item);  
...  
session.removeAttribute("shoppingCart");  
...  
session.setAttribute("shoppingCart", new Cart());  
...  
Enumeration e = session.getAttributeNames();  
while(e.hasMoreElements())  
    out.println("Key; " + (String)e.nextElement());
```


Altre operazioni con le sessioni

- **String getId()** restituisce l'ID di una sessione
- **boolean isNew()** dice se la sessione è nuova
- **void invalidate()** permette di invalidare (*distruggere*) una sessione
- **long getCreationTime()** dice da quanto tempo è attiva la sessione (in millisecondi)
- **long getLastAccessedTime ()** dà informazioni su quando è stata utilizzata l'ultima volta

```
String sessionId = session.getId();  
if(session.isNew())  
    out.println("La sessione e' nuova");  
session.invalidate();  
out.println("Millisec:" + session.getCreationTime());  
out.println(session.getLastAccessedTime());
```

Session tracking basics

...

```
HttpSession session = request.getSession();
synchronized(session) {
    SomeClass value = (SomeClass) session.getAttribute("someID");
    if (value == null) {
        value = new SomeClass();
    }
    doSomethingWith(value);
    session.setAttribute("someID", value);
}
```

To Synchronize or not to Synchronize?

- Non ci sono **race conditions** quando più utenti diversi accedono contemporaneamente alla pagina (perchè?)
- Sembra praticamente impossibile che lo stesso utente acceda concorrentemente alla sua sessione
- L'ascesa di Ajax rende necessaria la sincronizzazione
 - Con le chiamate Ajax, è abbastanza probabile che due richieste dallo stesso utente possano arrivare concorrentemente

Accumulating a list of user data

...

```
HttpSession session = request.getSession();
synchronized (session) {
    @SuppressWarnings("unchecked")
    List<String> previousItems = (List<String>) session.getAttribute("previousItems");
    if (previousItems == null) {
        previousItems = new ArrayList<String>();
    }
    String newItem = request.getParameter("newItem");
    if (newItem != null) {
        previousItems.add(newItem);
    }
    session.setAttribute("previousItems", previousItems);
}
```

...

```

protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    HttpSession session = request.getSession();
    synchronized (session) {
        String heading;
        Integer accessCount = (Integer) session.getAttribute("accessCount");
        if (accessCount == null) {
            accessCount = 0;
            heading = "Welcome, Newcomer";
        } else {
            heading = "Welcome Back";
            accessCount = accessCount + 1;
        }
        session.setAttribute("accessCount", accessCount);

        PrintWriter out = response.getWriter();
        out.println
            ("<!DOCTYPE html>" +
             "<html>" +
             "<head><title>Session Tracking Example</title></head>" +
             "<body>" +
             "<h1>" + heading + "</h1>" +
             "<h2>Information on Your Session:</h2>" +
             "<table border='1'>" +
             "<tr>" +
             "    <th>Info Type</th><th>Value</th>" +
             "</tr><tr>" +
             "    <td>ID</td><td>" + session.getId() + "</td>" +
             "</tr><tr>" +
             "    <td>Creation Time</td><td>" + new Date(session.getCreationTime()) + "</td>" +
             "</tr><tr>" +
             "    <td>Time of Last Access</td><td>" + new Date(session.getLastAccessedTime()) + "</td>" +
             "</tr><tr>" +
             "    <td>Number of Previous Accesses</td><td>" + accessCount + "</td>" +
             "</tr></table>" +
             "</body></html>");
    }
}

```

Session ID e URL Rewriting

- *Il session ID è usato per identificare le richieste provenienti dallo stesso utente e mapparle sulla corrispondente sessione*
- **Una tecnica per trasmettere l'ID è quella di includerlo in un cookie (session cookie):** sappiamo però che non sempre i cookie sono attivati nel browser
- **Un'alternativa è rappresentata dall'inclusione del session ID nella URL:** si parla di **URL rewriting**
- È buona prassi codificare sempre le URL generate dalle Servlet usando il metodo **encodeURL()** di **HttpServletResponse**
 - Usato per garantire una gestione corretta della sessione
 - Se il server sta usando i cookie, ritorna l'URL non modificato
 - Se il server sta usando l'URL rewriting, prende in input un URL, e se l'utente ha i cookie disattivati, codifica l'id di sessione nell'URL
- Il metodo encodeURL() dovrebbe essere usato per:
 - hyperlink ()
 - form (<form action="...">)
- Es:

```
String url = "order-page.html";  
url = response.encodeURL(url);
```


URL-Rewriting

- Idea
 - Client appends some extra data on the end of each URL that identifies the session
 - Server associates that identifier with data it has stored about that session
 - Es: **http://host/path/file.html;jsessionid=1234**
- Advantage
 - *Works even if cookies are disabled or unsupported*
- Disadvantages
 - Must encode all URLs that refer to your own site
 - All pages must be dynamically generated
 - Fails for bookmarks and links from other sites

Hidden form fields

- Idea:

`<input type="hidden" name="session" value="...">`

- Advantage

- *Works even if cookies are disabled or unsupported*

- Disadvantages

- Lots of tedious processing
- **All pages must be the result of form submissions**

Servlet context

- Ogni Web application esegue in un **contesto**:
 - corrispondenza 1:1 tra una Web application e suo contesto
- L'interfaccia **ServletContext** è la vista della Web application (del suo contesto) da parte della Servlet
- Si può ottenere un'istanza di tipo ServletContext all'interno della Servlet utilizzando il metodo **getServletContext()**
 - Consente di accedere ai **parametri di inizializzazione** e agli **attributi del contesto**
 - Consente di accedere alle risorse statiche della Web application (es. immagini) mediante il metodo **InputStream getResourceAsStream(String path)**
- **IMPORTANTE:** *Servlet context viene condiviso tra tutti gli utenti, le richieste e le Servlet facenti parte della stessa Web application*

Parametri di inizializzazione del contesto

- Parametri di inizializzazione del contesto definiti all'interno di elementi di tipo **context-param** in web.xml

```
<web-app>
  <context-param>
    <param-name>feedback</param-name>
    <param-value>feedback@deis.unibo.it</param-value>
  </context-param>
  ...
</ web-app >
```

- Sono accessibili a tutte le Servlet della Web application

```
...
ServletContext ctx = getServletContext();
String feedback =
ctx.getInitParameter("feedback");
...
```

Attributi di contesto

- Gli attributi di contesto sono accessibili a tutte le Servlet e funzionano come variabili “globali”
- Vengono gestiti a runtime:
 - possono essere creati, scritti e letti dalle Servlet
- Possono contenere oggetti anche complessi (serializzazione/deserializzazione)

scrittura

```
ServletContext ctx = getServletContext();  
ctx.setAttribute("utente1", new User("Giorgio Bianchi"));  
ctx.setAttribute("utente2", new User("Paolo Rossi"));
```

lettura

```
ServletContext ctx = getServletContext();  
Enumeration aNames = ctx.getAttributeNames();  
while (aNames.hasMoreElements())  
{  
    String aName = (String)aNames.nextElement();  
    User user = (User) ctx.getAttribute(aName);  
    ctx.removeAttribute(aName);  
}
```

Attenzione: scope DIFFERENZIATI (scoped objects)

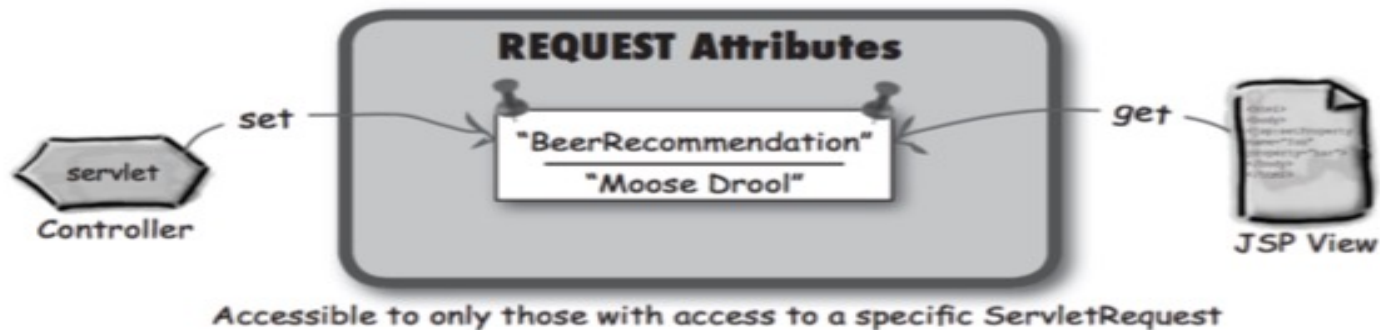
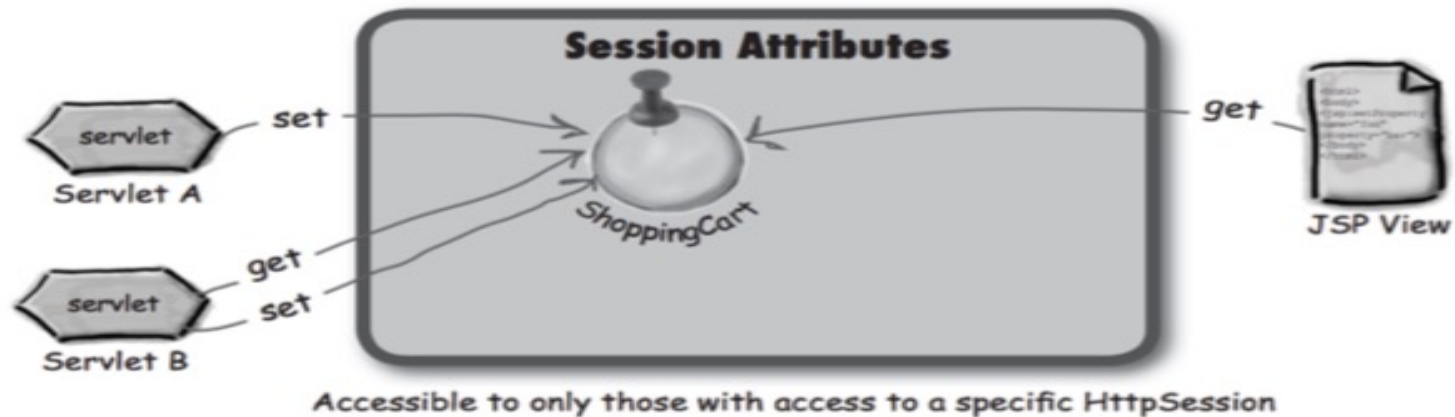
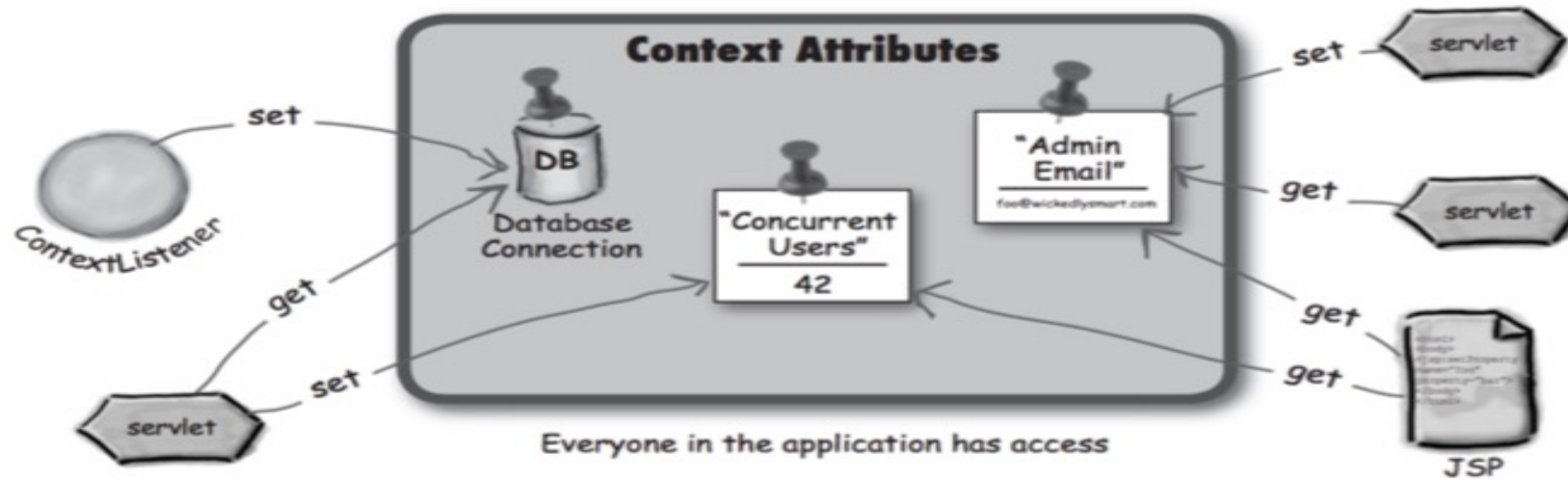
- Gli oggetti di tipo **ServletContext**, **HttpSession**, **HttpServletRequest** forniscono metodi per immagazzinare e ritrovare oggetti nei loro rispettivi ambiti (scope)
- Lo scope è definito dal **tempo di vita** (lifespan) e dall'**accessibilità** da parte delle Servlet

<u>Ambito</u>	<u>Interfaccia</u>	<u>Tempo di vita</u>	<u>Accessibilità</u>
Request	HttpServletRequest	Fino all'invio della risposta	Servlet corrente e ogni altra pagina inclusa o in forward
Session	HttpSession	Lo stesso della sessione utente	Ogni richiesta dello stesso client
Application	ServletContext	Lo stesso dell'applicazione	Ogni richiesta alla stessa Web app anche da clienti diversi e per servlet diverse

Funzionalità degli scoped object

- Gli oggetti scoped forniscono i seguenti metodi per immagazzinare e ritrovare oggetti nei rispettivi ambiti (scope):
 - **void setAttribute(String name, Object o);**
 - **Object getAttribute(String name);**
 - **void removeAttribute(String name);**
 - **Enumeration getAttributeNames();**

The Three Scopes: Context, Request, and Session



Ridirezione del browser

- È possibile inviare al browser una risposta che lo forza ad accedere ad un'altra pagina/risorsa (*ridirezione*)
- Possiamo ottenere agendo sull'oggetto **response** invocando il metodo
 - **public void sendRedirect(String url);**
- *Scenario: In un sito di commercio elettronico, una volta che selezioniamo il prodotto, siamo pronti per l'acquisto e clicchiamo su “paga”, il browser reindirizza alla rispettiva pagina di pagamento online. Qui, la risposta del sito di shopping lo reindirizza alla pagina di pagamento e un nuovo URL può essere visto nel browser.*
- Es:

```
String name=request.getParameter("name");  
response.sendRedirect("https://www.google.co.in/#q="+name);
```

Includere una risorsa (include)

- Per includere una risorsa si ricorre a un oggetto di tipo **RequestDispatcher** che può essere richiesto al contesto indicando la risorsa da includere
- Si invoca quindi il metodo **include** passando come parametri **request** e **response** che vengono così condivisi con la risorsa inclusa
 - Se necessario, l'URL originale può essere salvato come un attributo di request

può essere anche una pagina JSP

```
RequestDispatcher dispatcher =  
    getServletContext().getRequestDispatcher("/inServlet");  
dispatcher.include(request, response);
```


Inoltro (forward)

- Si usa in situazioni in cui una Servlet si occupa di parte dell'elaborazione della richiesta e delega a qualcun altro la gestione della risposta
 - *Attenzione: in questo caso la **risposta** è di **competenza esclusiva** della risorsa che riceve l'inoltro*
 - Se nella prima Servlet è stato fatto un accesso a `ServletOutputStream` o `PrintWriter` si ottiene una `IllegalStateException` (ovvero, se anche la prima Servlet vuole scrivere nella risposta)
- Si deve ottenere un oggetto di tipo **RequestDispatcher** da request passando come parametro il nome della risorsa
- Si invoca quindi il metodo **forward** passando anche in questo caso **request** e **response**

può essere anche una pagina JS

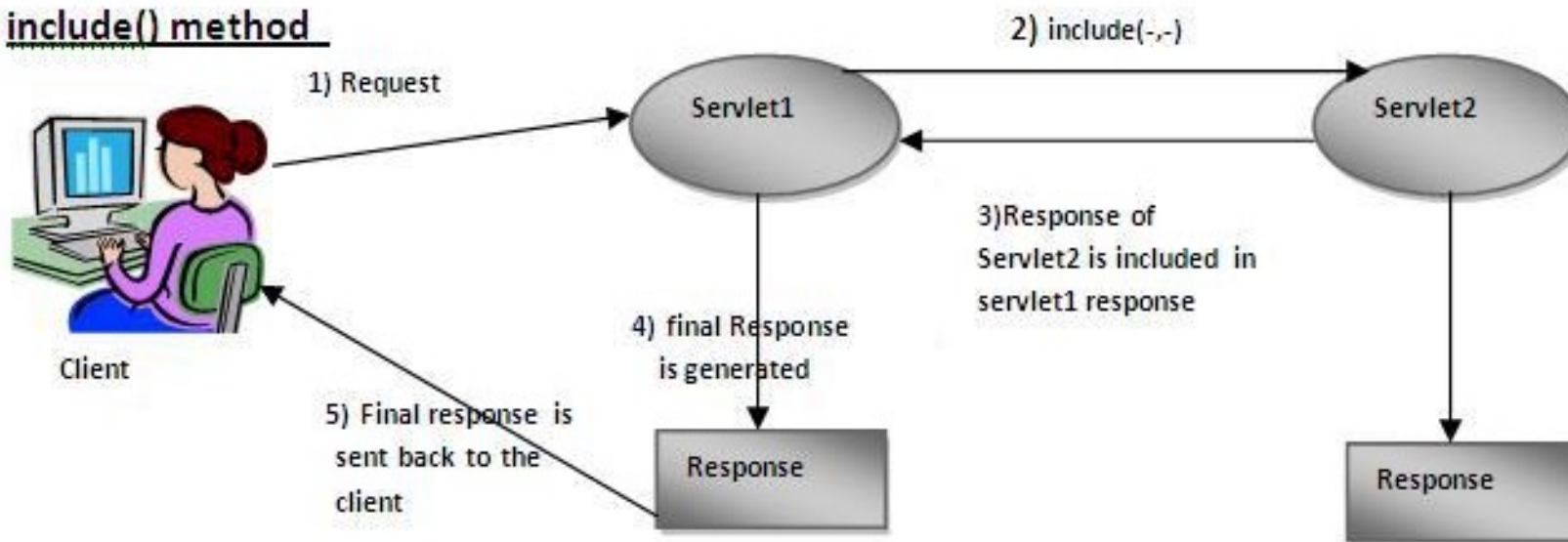
```
RequestDispatcher dispatcher =  
    getServletContext().getRequestDispatcher("/inServlet");  
dispatcher.forward(request, response);
```

Include e forward

- **Scenario (include):** When the current Servlet has done some of its job and is using another Servlet (or JSP) to help with display or other work. If the Servlet using the RequestDispatcher has already written to the response, then you will need to use the include to send control to another Servlet, not the forward. Also, if you want to spread the display of a page across multiple pages, use an Include for each part of a page
 - **Example:** Servlet handles a request from the client and does the necessary work. It then uses `RequestDispatcher#include` to add a Header response to all pages, since the header and banner on a page is the same for all pages on the the site. It then includes a second JSP which displays the content that is specific to this request. It finally includes a third page that acts as a Footer for all pages on the site
- **Scenario (forward):** When the current Servlet is done its job, and hasn't done anything with the response, you Forward the request and response to another Servlet (or JSP) to finish any work and to display results
 - **Example:** A Servlet retrieves information from a database and stores it in the request object. It then forwards the request to a JSP to display the data. User presses refresh, and the control goes to the Servlet, which again retrieves the data from the database and puts it in the request, forwards to the JSP, and the JSP sees the data

Include e forward

include() method



forward() method:

