**Chapter 9**

In the previous chapter we have learnt about ServletContextListener, but Listeners are not only for context events!! Where there’s a *lifecycle moment*, there’s usually a *listener* to hear about it. Besides context events, you can listen for events related to context *attributes*, servlet requests and attributes, and HTTP sessions and session attributes.

There is no need to learn about each and every listener interfaces, just for reference take a look over list of interfaces.

|  |  |
| --- | --- |
| **Scenario** | **Listener Interface** |
| You want to know if an attribute in a web app context has been added, removed or replaced. | ServletContextAttributeListener |
| You want to know how many concurrent users there are. In other words, you want to track the active sessions. | HttpSessionListener |
| You want to know each time a request comes in, so that you can log it. | ServletRequestListener |
| You want to know when a request attribute has been added, removed, or replaced. | ServletRequestAttributeListener |
| You have an attribute class (a class for an object that will be stored as an attribute) and you want objects of this type to be notified when they are bound to or removed from a session. | HttpSessionBindingListener |
| You want to know when a session attribute has been added, removed, or replaced. | HttpSessionAttributeListener |
| You want to know if a context has been created or destroyed. | ServletContextListener |
| You have an attribute class, and you want objects of this type to be notified when the session to which they’re bound is migrating to and from another JVM. | HttpSessionActivationListener |

Here we have a list of eight listeners, as of now we are not going to implement them all but we will have a look over a special type of listener “HttpSessionBindingListener”, we have called this listener a special type because usually other listeners are java classes through which we can determine when any type of attribute(primitive or complex type) has been added, removed or replaced in a Context or Session, but HttpSessionBindingListener exists so that the attribute itself can find out when it has been added to or removed from a Session(as of now don't worry about Session, we will cover this in the future).

This time our sample Employee object is also a Listener, listening for when the Employee itself is added or removed from a Session(NOTE : binding listeners are not registered in the Deployement Descriptor or web.xml, it just happens automatically).

API has used the word “bound” and “unbound” to mean “added to” and “removed from”.

public class Employee implements HttpSessionBindingListener {

private Integer empId;

private String empName;

public Employee() {

}

public Employee(Integer empId, String empName) {

super();

this.empId = empId;

this.empName = empName;

}

public Integer getEmpId() {

return empId;

}

public void setEmpId(Integer empId) {

this.empId = empId;

}

public String getEmpName() {

return empName;

}

public void setEmpName(String empName) {

this.empName = empName;

}

@Override

public void valueBound(HttpSessionBindingEvent event) {

System.out.println(this.toString() + " has added into session");

}

@Override

public void valueUnbound(HttpSessionBindingEvent event) {

System.out.println(this.toString() + " has removed from session");

}

@Override

public String toString() {

return "Employee [empId=" + empId + ", empName=" + empName + "]";

}

}

**Attributes :**

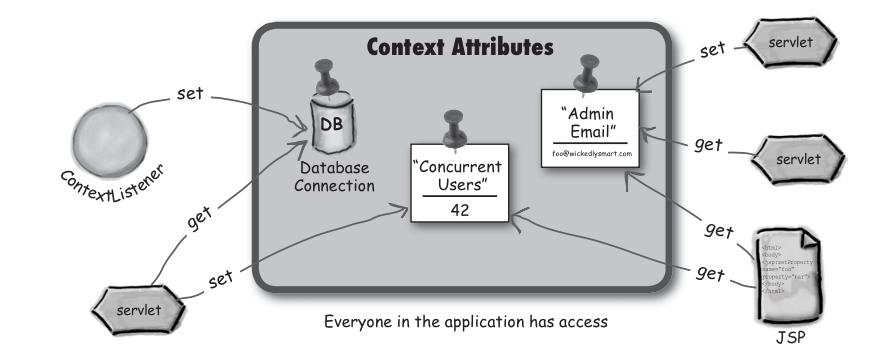
We have seen how a database connection object can stick to ServletContext as an attribute, so that other parts of the app could get it, earler we have discussed how servlets was able to stick the data/object as an request attribute(usually HttpServletrequest) so that other JSPs/views could get the value. So basically what is this attribute?

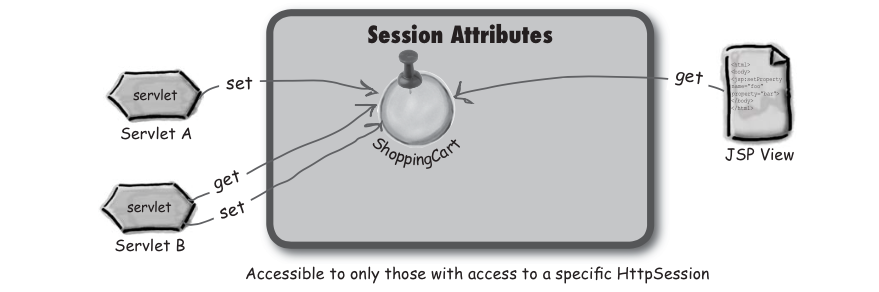
An attribute is an object set into one of three other servlet API objects – ServletContext, HttpServletRequest or HttpSession. We can think of it as simply a name/value pair (where the name is a String and the value is an Object) in a map instance variable. In reality we don't know or care how it's actually implemented, all we really care about is the “scope” in which the attribute exists, in other words, who can see it and how long does it live.

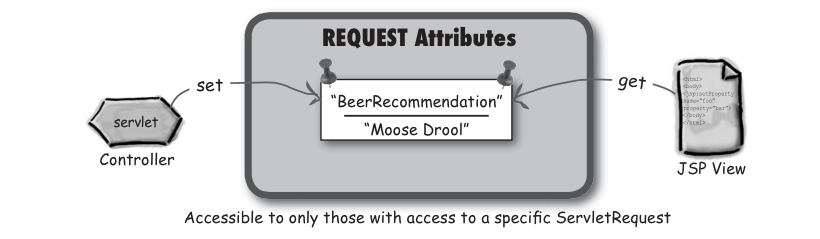
One thumb rule one should remember that **Attributes are not parameters!!**

Another important thing is that who can see/access it, in order to control that nature we have three scopes for attributes

**The Three Scopes : Context,Request and Session**

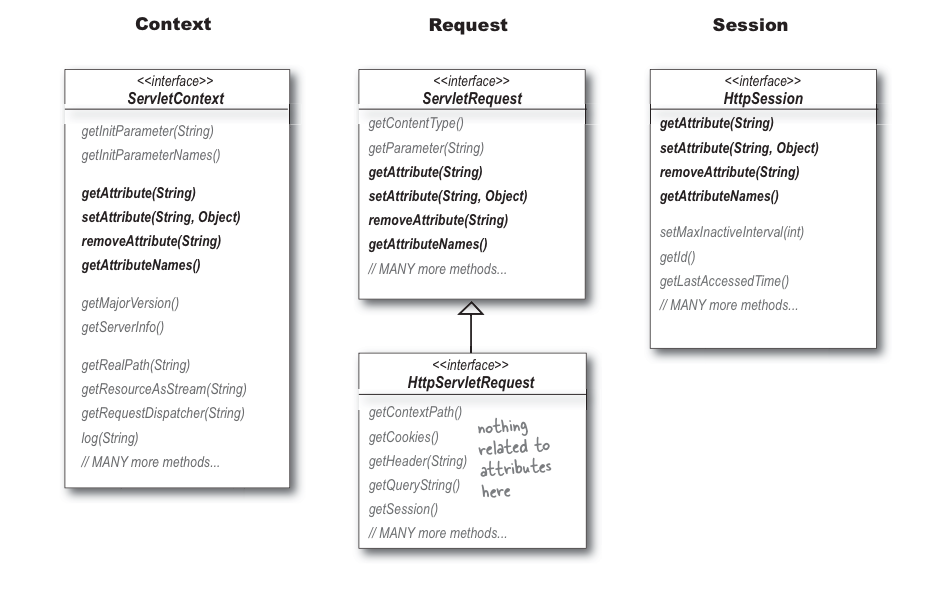






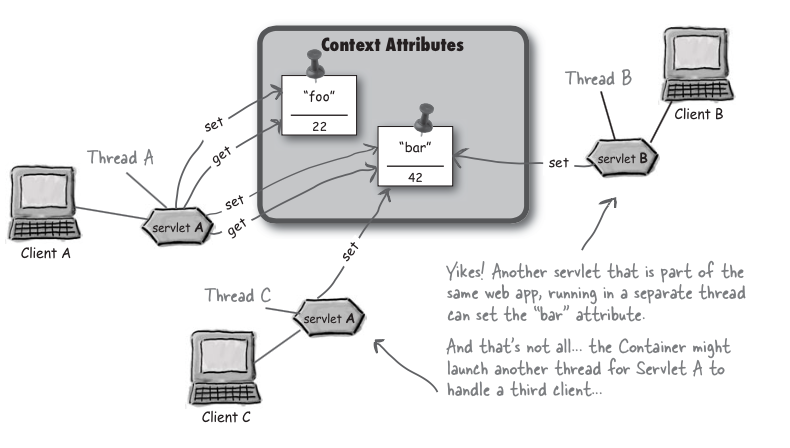
**Attribute API**

The three attribute scopes—context, request, and session—are handled by the ServletContext, ServletRequest, and HttpSession interfaces. The API methods for attributes are exactly the same in every interface.



**Context Scope Of Attributes Are Not Thread Safe :**

Remember, everyone in the app has access to context attributes, that means multiple servlets and multiple servlets means we might have multiple threads since requests are concurrently handled, each in a separate thread. This happens regardless of whether the requests are coming in for the same or different servlets.

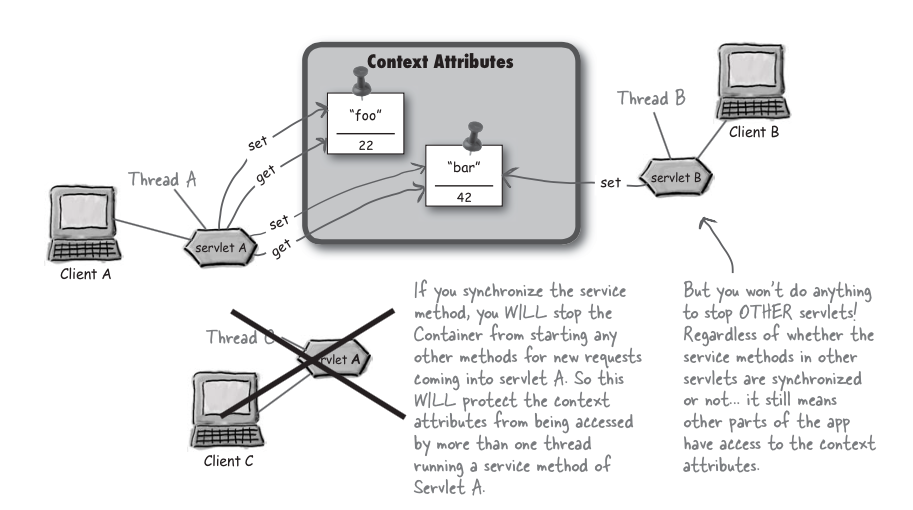


**How do we make context attributes thread-safe?**

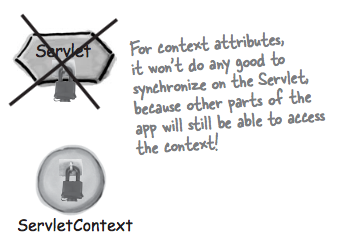
First thought that comes to our mind is why not synchronize service method(doGet or doPost),but when we ponder over this solution we come to conclusion that this mechanism is a spectacularly BAD idea. First reason is as we know that synchronizing the service method will kill our concurrency, but also it desn't solve our problem!!!.

Synchronizing the service method means that only one thread in a servlet can be running at a time... but it doesn’t stop other servlets or JSPs from accessing the attribute!

Synchronizing the service method would stop other threads from the same servlet from accessing the context attributes, but it won’t do anything to stop a completely different servlet.



**We don't need a lock on the servlet!!.. we need the lock on the context**



The typical way to protect the context attribute is to synchronize ON the context object itself. If everyone accessing the context has to first get the lock on the context object, then you’re guaranteed that only one thread at a time can be getting or setting the context attribute. But... there’s still an if there. It only works if all of the other code that manipulates the same context attributes ALSO synchronizes on the ServletContext. If code doesn’t ask for the lock, then that code  
is still free to hit the context attributes. But if you’re designing the web app, then you can decide to make everyone ask for the lock before accessing the attributes.

Here mentioned below, sample code that full fills the above mentioned strategy!!!!



**Are Session attributes thread-safe :**

We haven’t talked about HTTP sessions in detail yet (we will in the Sessions chapter), but you already know that a session is an object used to maintain conversational state with a client. The session persists across multiple requests from the same client. But it’s still just one client we’re talking about.

And if it’s one client, and a single client can be in only one request at a time, doesn’t that automatically mean that sessions are thread-safe? In other words, even if multiple servlets are involved, at any given moment there’s only one request from that particular client... so there’s only one thread operating on that session at a time. Right?

Even though both servlets can access the Session attributes in separate threads, each thread is a separate request. So it looks safe.

Unless...Can you think of a scenario in which there could be more than one request at the same time, from the same client?

***What do you think? Are session attributes guaranteed thread-safe?***

The client could open a new browser window! So the Container can still use the same session for a client, even though it’s coming from a different instance of the browser?

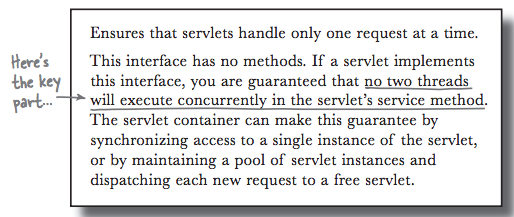
Look at the technique we used to protect the context attributes. What did we do?

You can do the same thing with session attributes, by synchronizing on the HttpSession object!

**

If you remember, we already discussed about Multi Threading issues in the Doc “6.Servlets\_Part\_Three.docx”, here we are picking up this topic with more description, if you again remember that we discussed about SingleThreadModel interface, YESS!! it is not an important topic, but digging into it will only add upto our knowledge.

Here’s what the servlet specification says about the SingleThreadModel (or STM) interface:



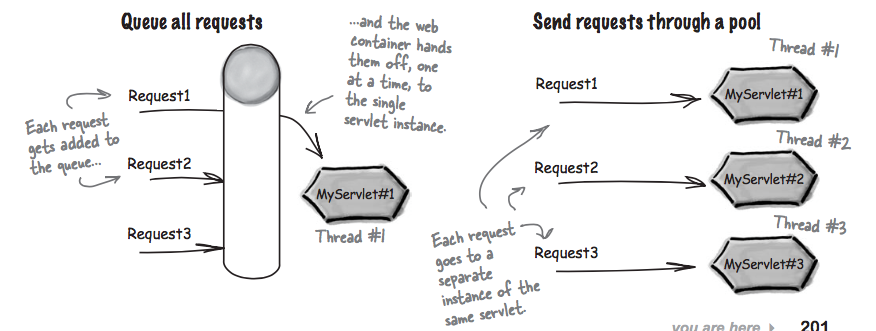
*But how does the web container guarantee a servlet gets only one request at a time?*

The web container vendor has a choice. The container can maintain a single servlet, but queue every request and process one request completely before allowing the next request to proceed. Or the container can create a pool of servlet instances and process each request concurrently, one per servlet instance.

Which STM strategy do you think is better?

In first, it seems that if container create a pool of servlet instances and then the conatiner can process one request with one servlet instance and another request with a second instance. Each request will handle in parallel.

Let me remind us the servlet spec defines that a single servlet declaration in the deployment descriptor becomes a single object instance at runtime, but now using the STM interface, this definition is no longer valid.



What if one of the instance variables is meant to record how many requests have been processed. The counter variable would have several different counts, and none of them would be right... only the summation of them is correct.

Which STM methodology used, is the decision of vendor, because of this we may get some questions!!!

*Q : How will the STM strategy change how I write my servlet code?*

*A : If the container uses a queuing strategy, then the “single servlet instance” semantics still hold and you do not need to make any code changes. But if the container uses a pooling strategy, then the semantics of some instance variables might change. For example, if you have an instance variable that holds a “request counter,” then that variable no longer can be counted on when multiple servlet instances are created in the pool. In this case, you could choose to make the counter variable a class variable instead.*

*Q : But are class variables thread-safe?*

*A : No, they are not, and the STM mechanism does not help with class variables. Yes, it protects instance variables from concurrent access, but by pooling multiple instances the semantics of the servlet changes. Furthermore, STM does not help with other variable or attribute scopes. You are on your own...*

***Q : So what good is using the SingleThreadModel?***

***A : None, really. Which is why STM has been deprecated from the servlet API!***

So the key point regarding Multi Threadding problem of servlets is

**Instance variables are n't thread safe!!!**

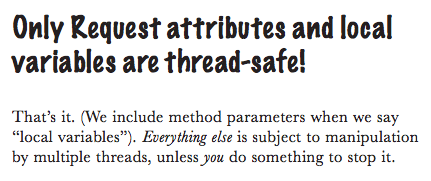
If you have multiple clients making requests on that servlet, that means multiple threads running that servlet code. And all threads have access to the servlet’s instance variables, so instance variables aren’t thread-safe.

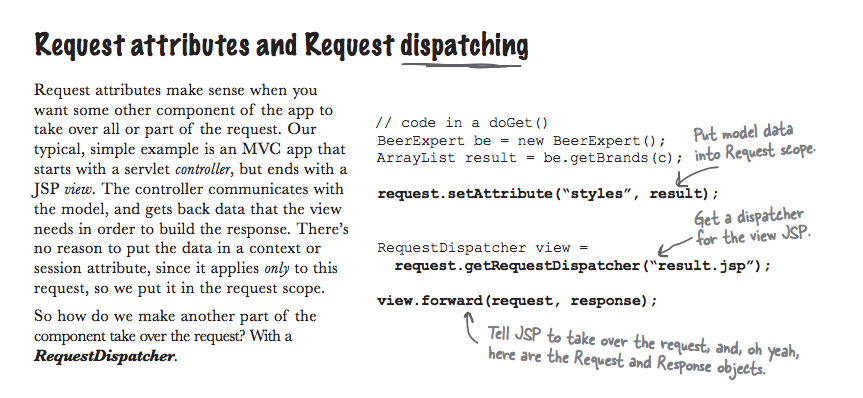
So if we are not supposed to use SingleThreadModel or synchronize the service method as both can hamper the performance a lot and can bring a web app to its knees, then how do we make instance variables thread safe???

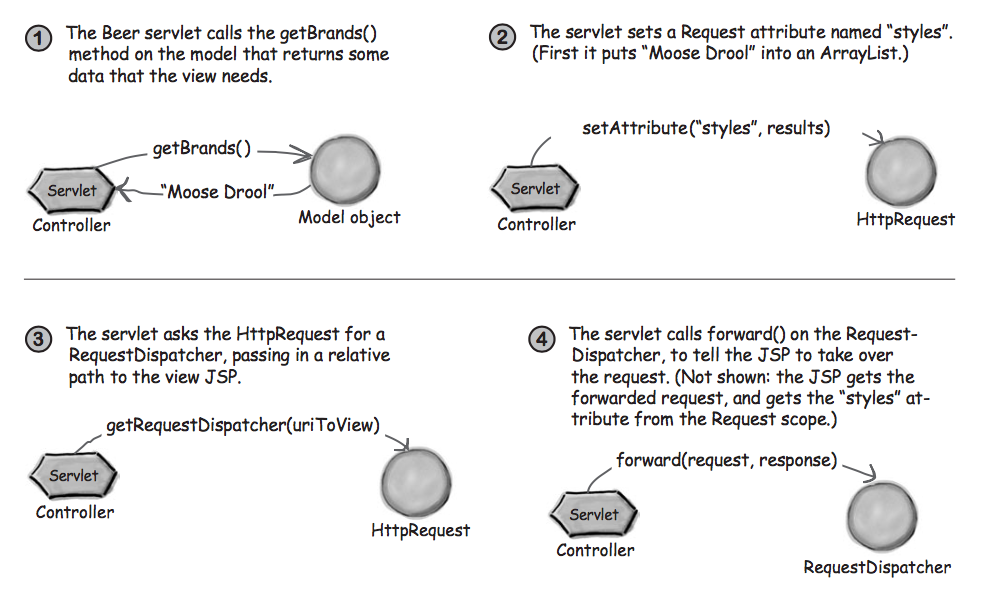
We don’t! Look at a well-written servlet, and chances are we won’t find any instance variables. Or at least any that are non-final. (And since we’re a Java programmer we know that even a final variable can still be manipulated unless it’s immutable.)

So just don’t use instance variables if you need thread-safe state, because all threads for that servlet can step on instance variables.

So the thumb rule is :

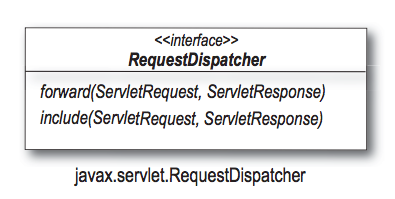


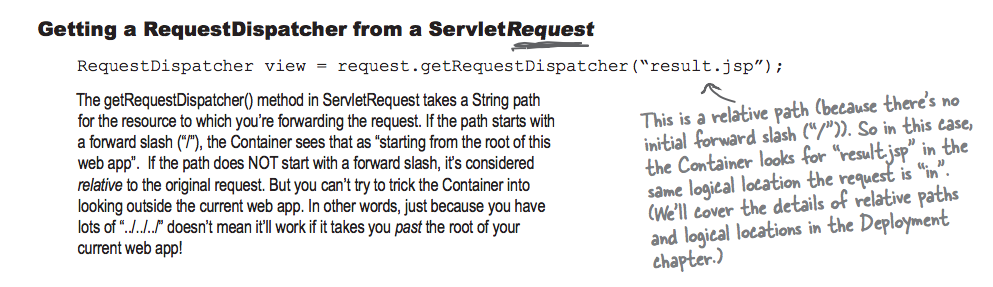


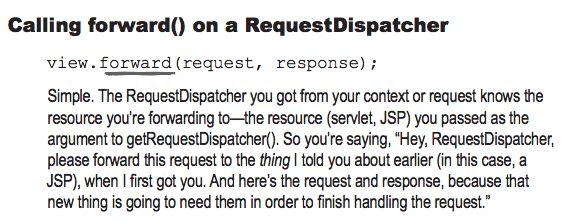
****

**Request Dispatcher :**

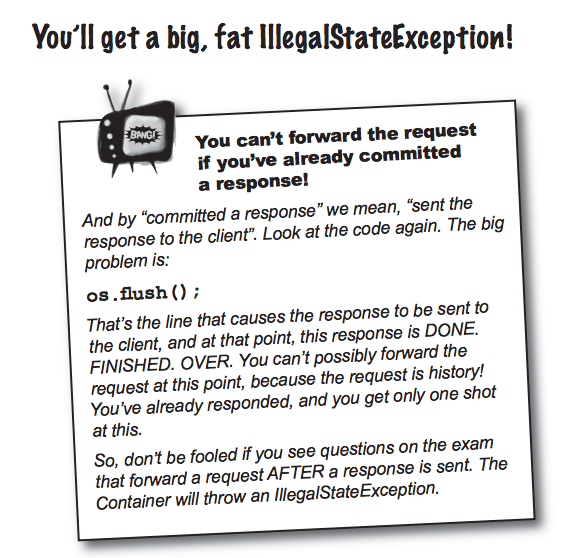
RequestDispatchers have only two methods—forward() and include(). Both take the request and response objects (which the component you’re forwarding to will need to finish the job). Of the two methods, forward() is by far the most popular. It’s very unlikely you’ll use the include method from a controller servlet; however, behind the scenes the include method is being used by JSPs in the <jsp:include> standard action (which we’ll review in the coming chapters). You can get a RequestDispatcher in two ways: from the request or from the context. Regardless of where you get it, you have to tell it the web component to which you’re forwarding the request. In other words, the servlet or JSP that’ll take over.

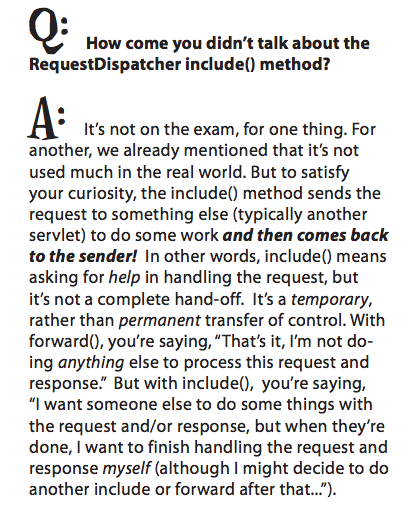












Another scenario for the include is The forward method redirects to another resource which is called through a resource. The response of calling resource is not sent to the client. It is lost. What should be done if we want to retain the response of calling resource and called resource? What if we want to show them both to the client?

**Summary :**

