Introduction to J2EE

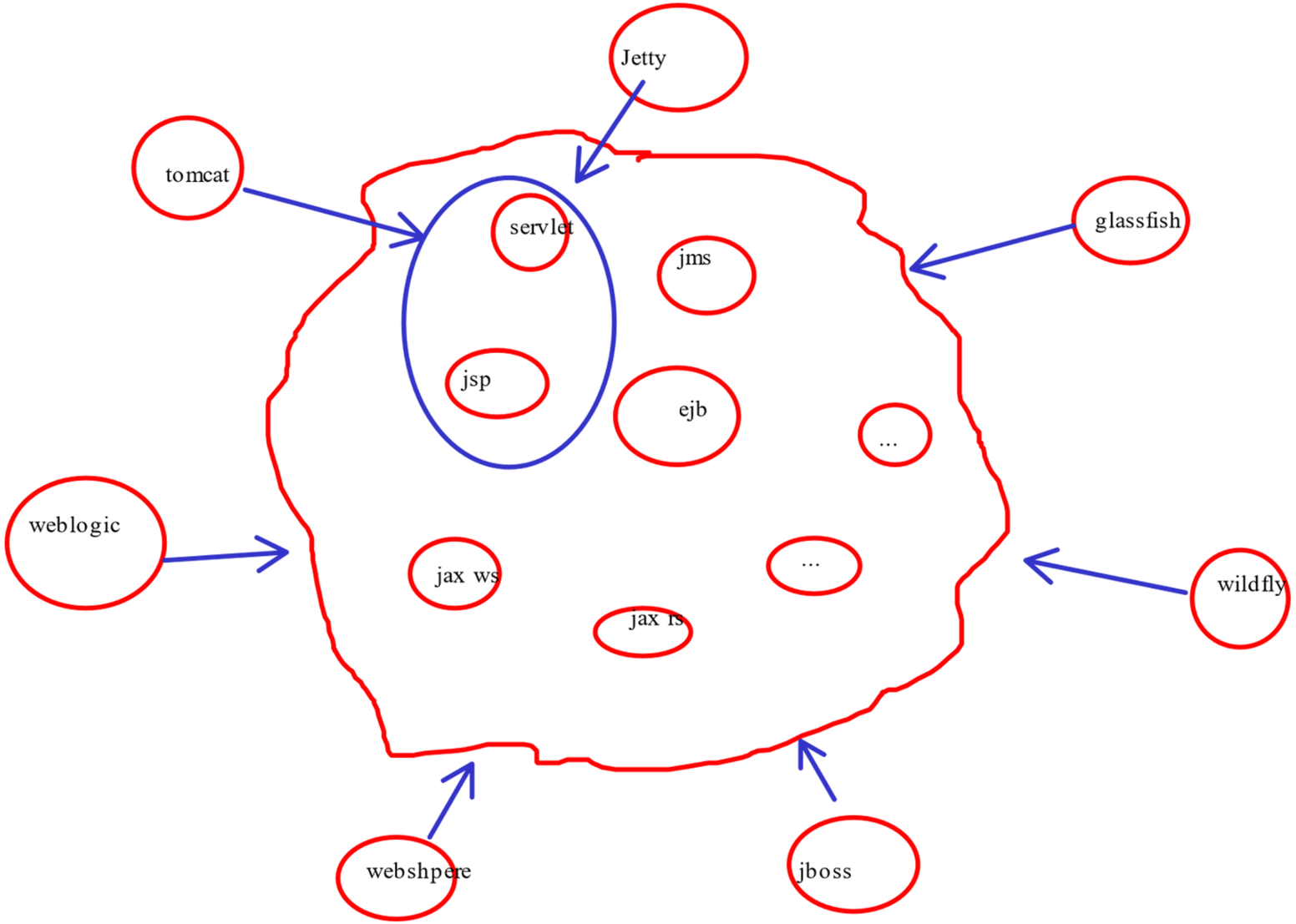
Servlet API

JCP

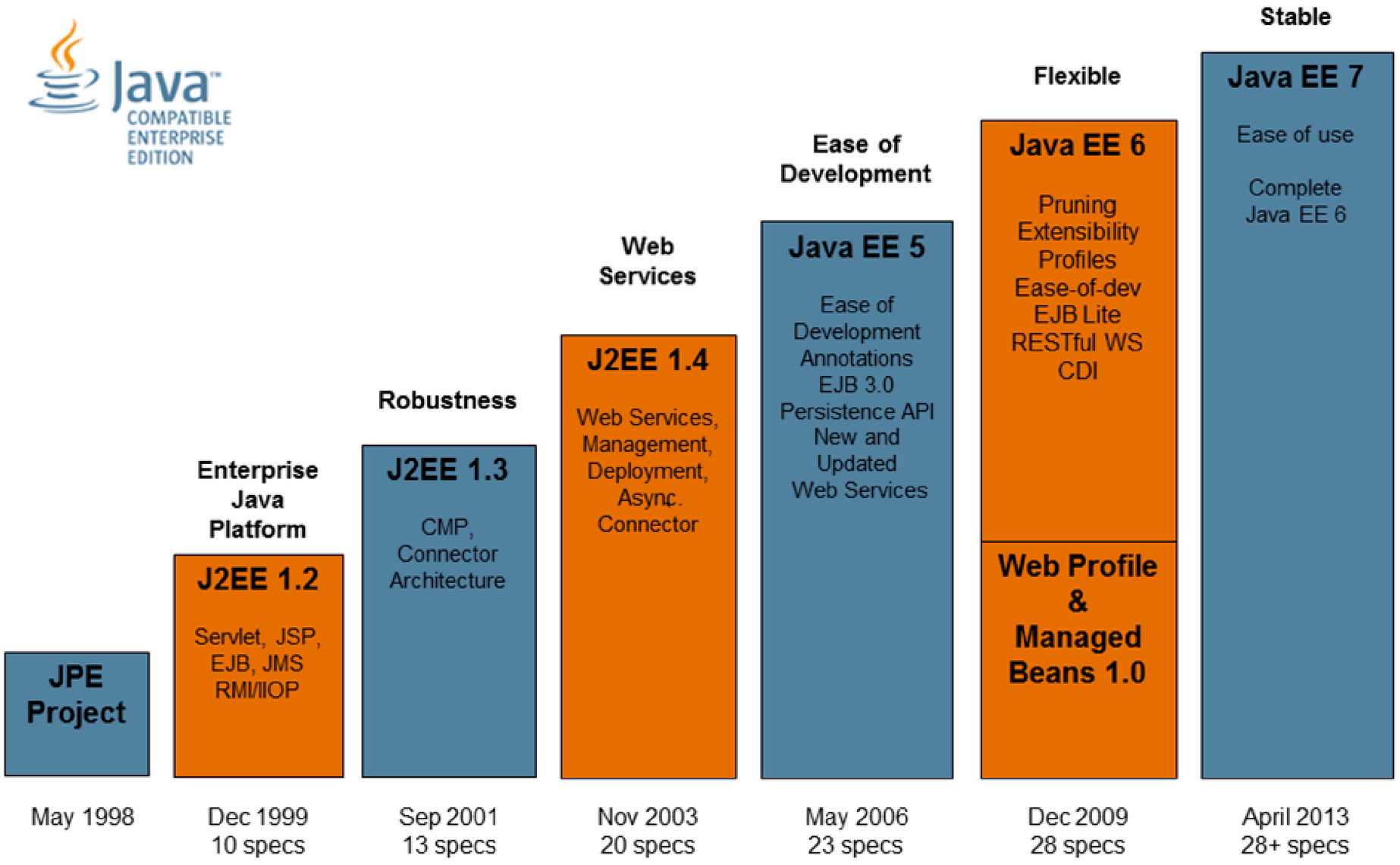
JSR

What is J2EE? Introduction

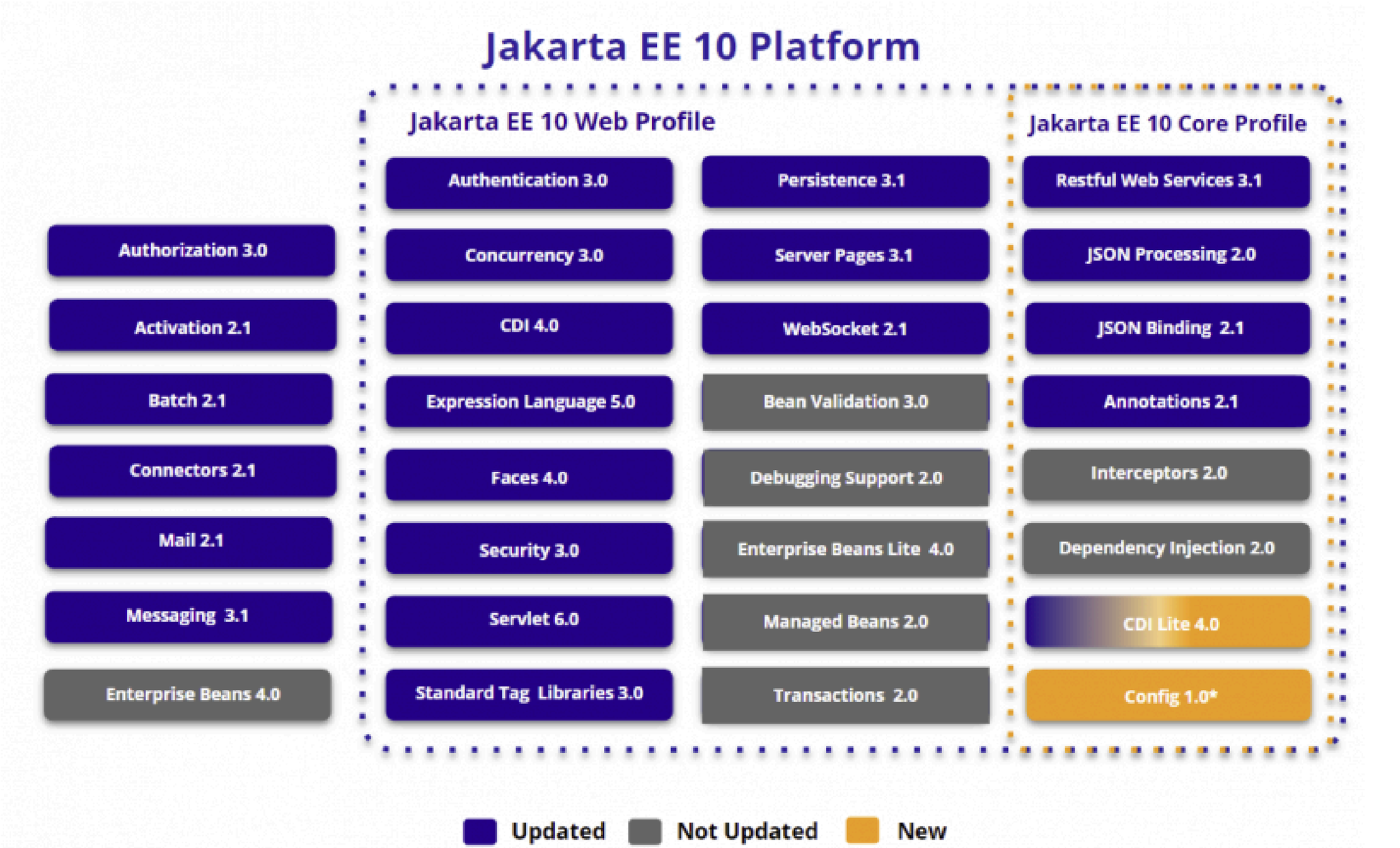
J2EE is group of specification to create dynamic distributed application



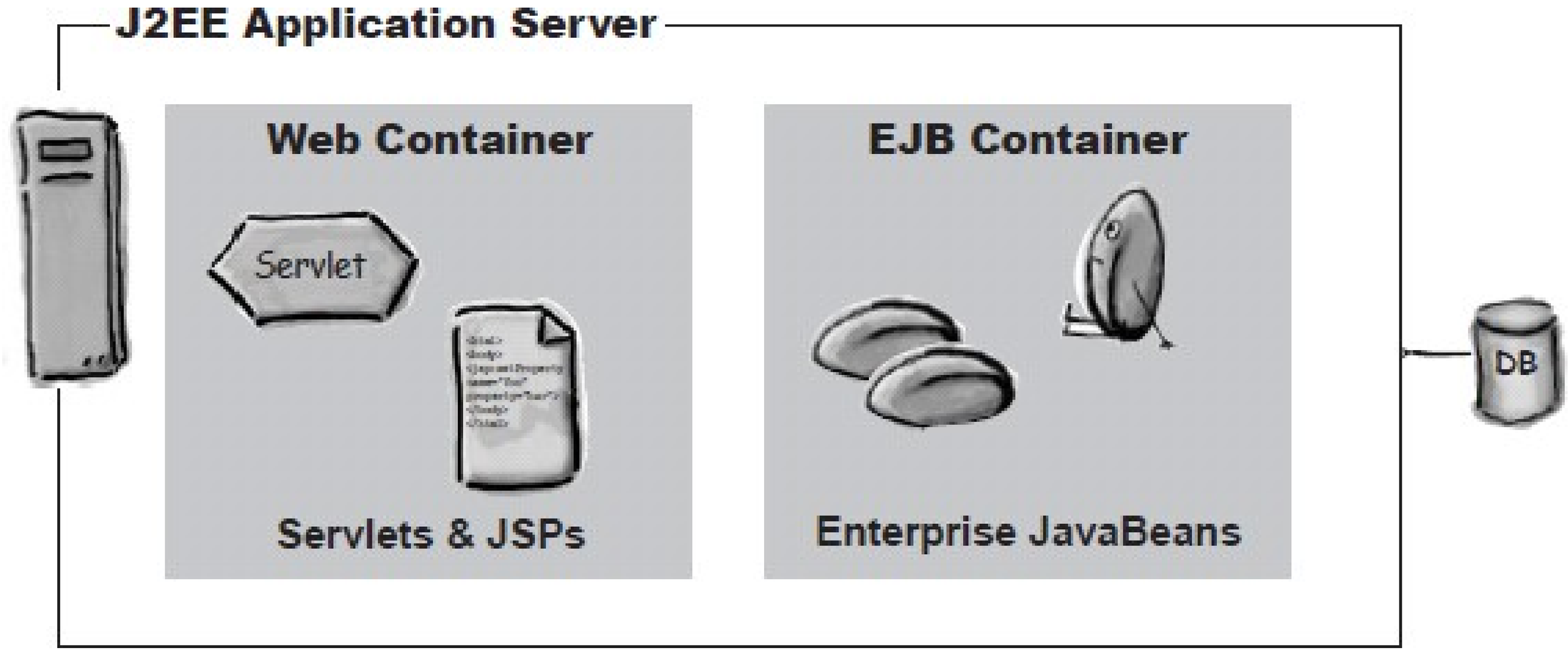
What is J2EE? Introduction



Jakarta EE


Jakarta EE 10 Platform
Jakarta EE 10 Web Profile
Jakarta EE 10 Core Profile
Authorization 3.0
Activation 2.1
Batch 2.1
Connectors 2.1
Mail 2.1
Messaging 3.1
Enterprise Beans 4.0
Authentication 3.0
Concurrency 3.0
CDI 4.0
Expression Language 5.0
Faces 4.0
Security 3.0
Servlet 6.0
Standard Tag Libraries 3.0
Persistence 3.1
Server Pages 3.1
WebSocket 2.1
Bean Validation 3.0
Debugging Support 2.0
Enterprise Beans Lite 4.0
Managed Beans 2.0
Transactions 2.0
Restful Web Services 3.1
JSON Processing 2.0
JSON Binding 2.1
Annotations 2.1
Interceptors 2.0
Dependency Injection 2.0
CDI Lite 4.0
Config 1.0*
Updated
Not Updated
New

J2EE application server

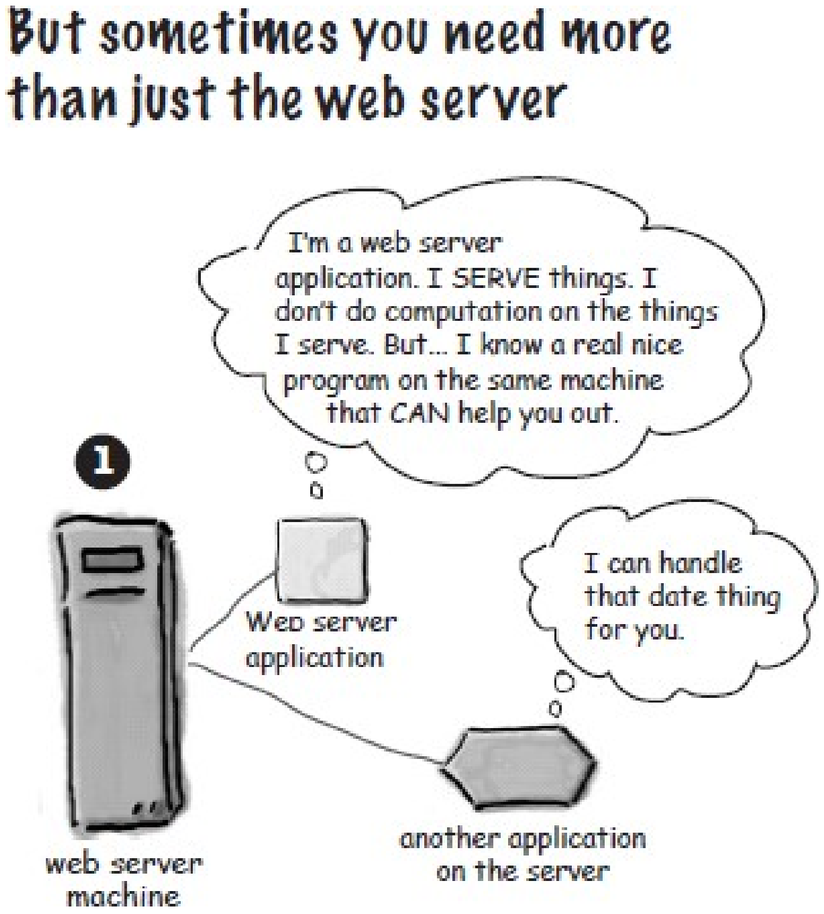


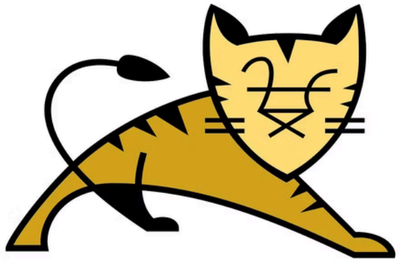
Introduction to Servlet API

Web Server vs Web Container vs Application Server

Apache Tomcat

Why we need tomcat?

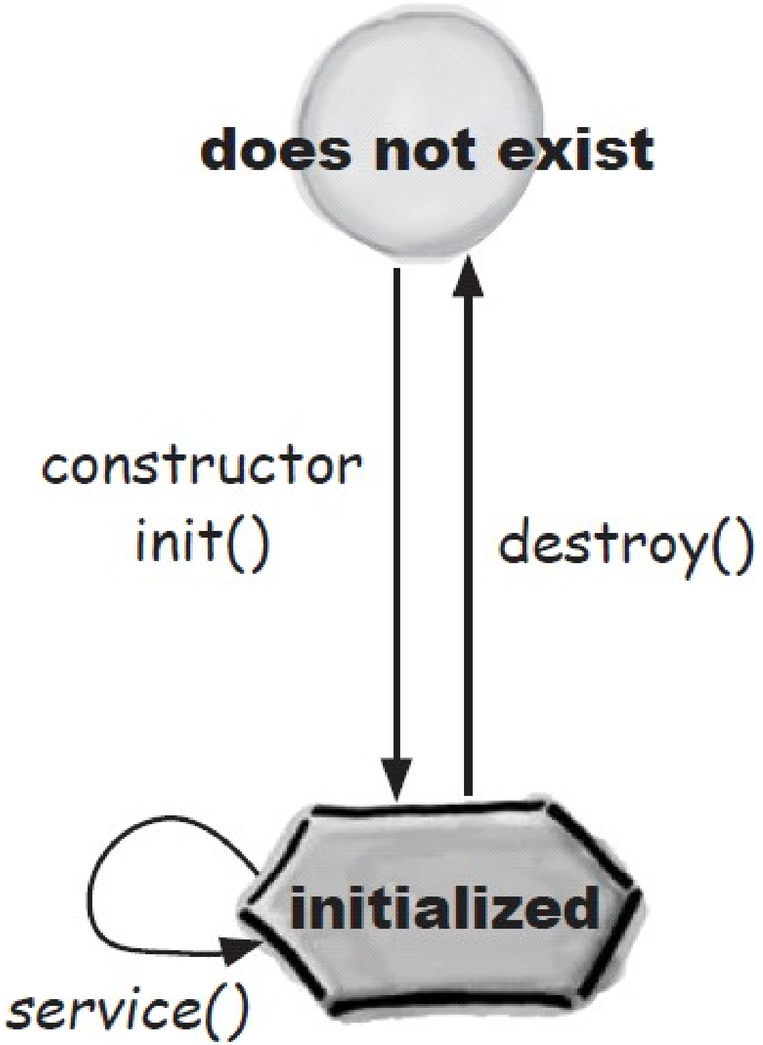


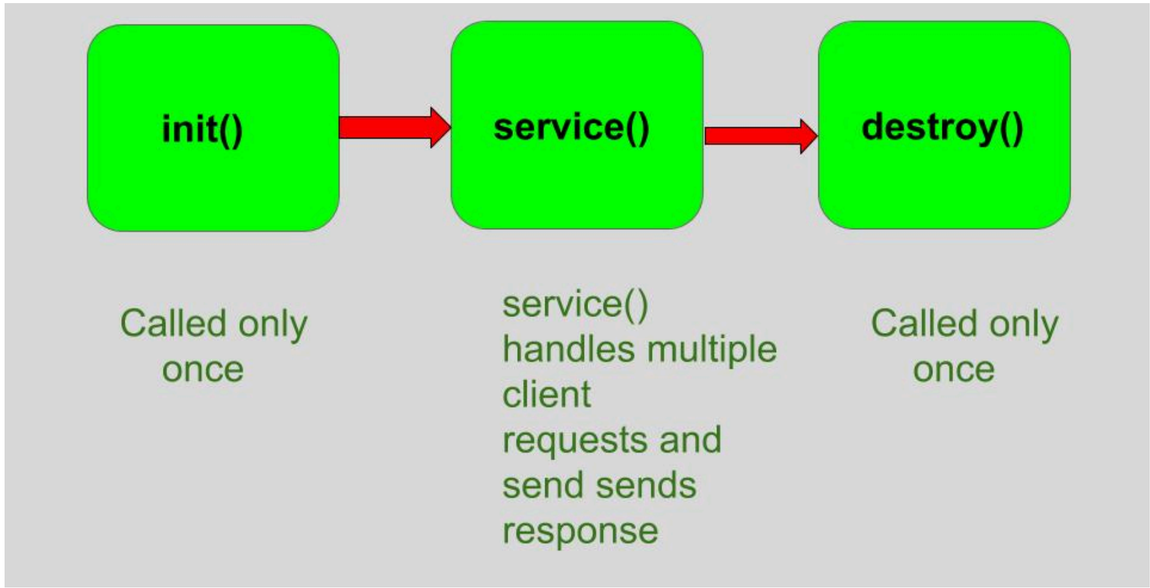


Apache Tomcat

- **Communication support**
- **Lifecycle management**
- **Multithreading support**
- **Declarative security**
- **JSP Support**
- **Dynamic Content**
- **Saving data on the server**

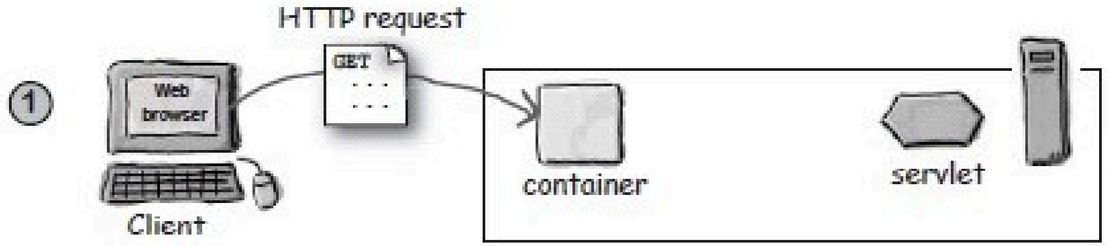
What is Servlet?



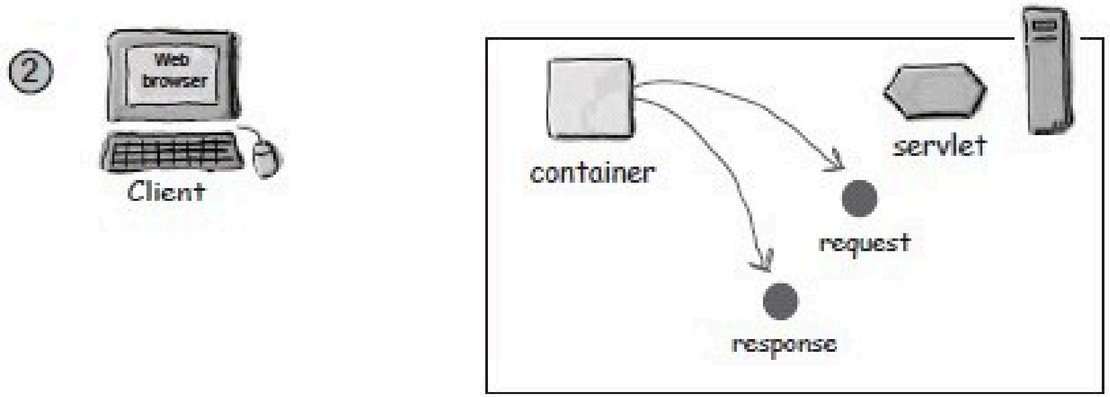


- **What is servlet?**
- **Why we need servlet?**

How a container handle a dynamic request?



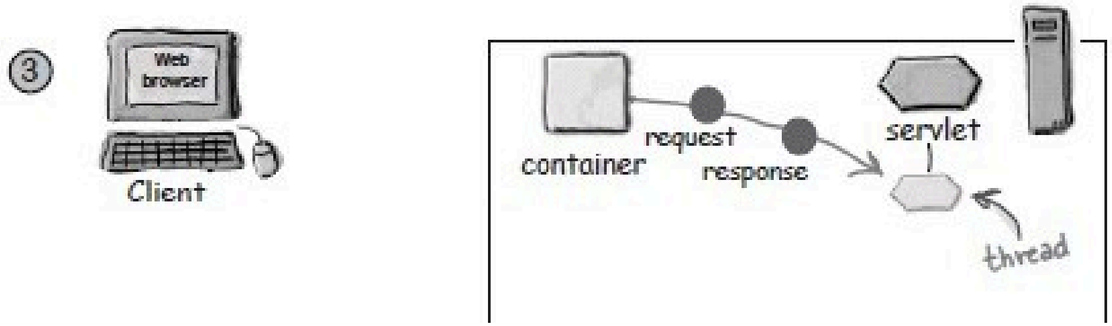
User clicks a link that has a URL to a servlet instead of a static page.



The container "sees" that the request is for a servlet, so the container creates two objects:

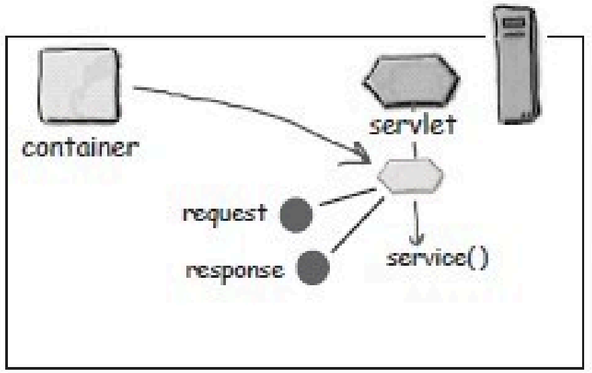
1) HttpServletResponse

2) HttpServletRequest



The container finds the correct servlet based on the URL in the request, creates or allocates a thread for that request, and passes the request and response objects to the servlet thread.

④

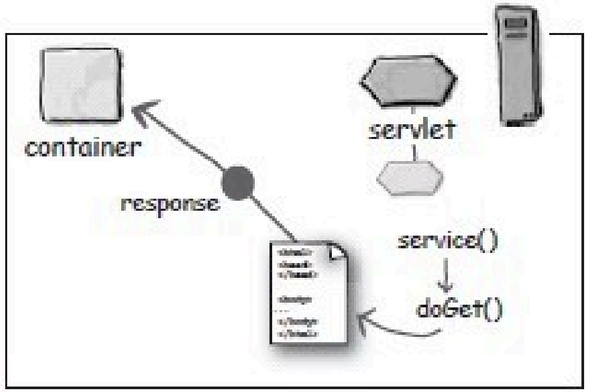


The container calls the servlet's service() method. Depending on the type of request, the service() method calls either the doGet() or doPost() method.

For this example, we'll assume the request was an HTTP GET.

---

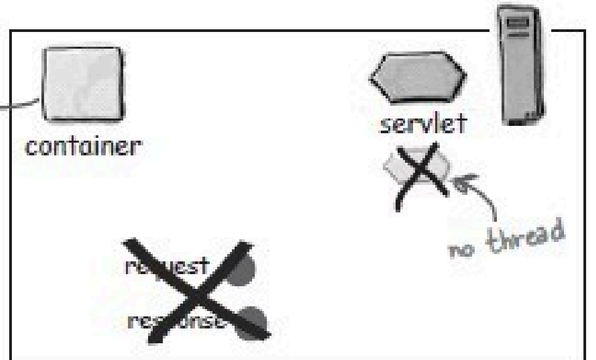
⑤



The doGet() method generates the dynamic page and stuffs the page into the response object. Remember, the container still has a reference to the response object!

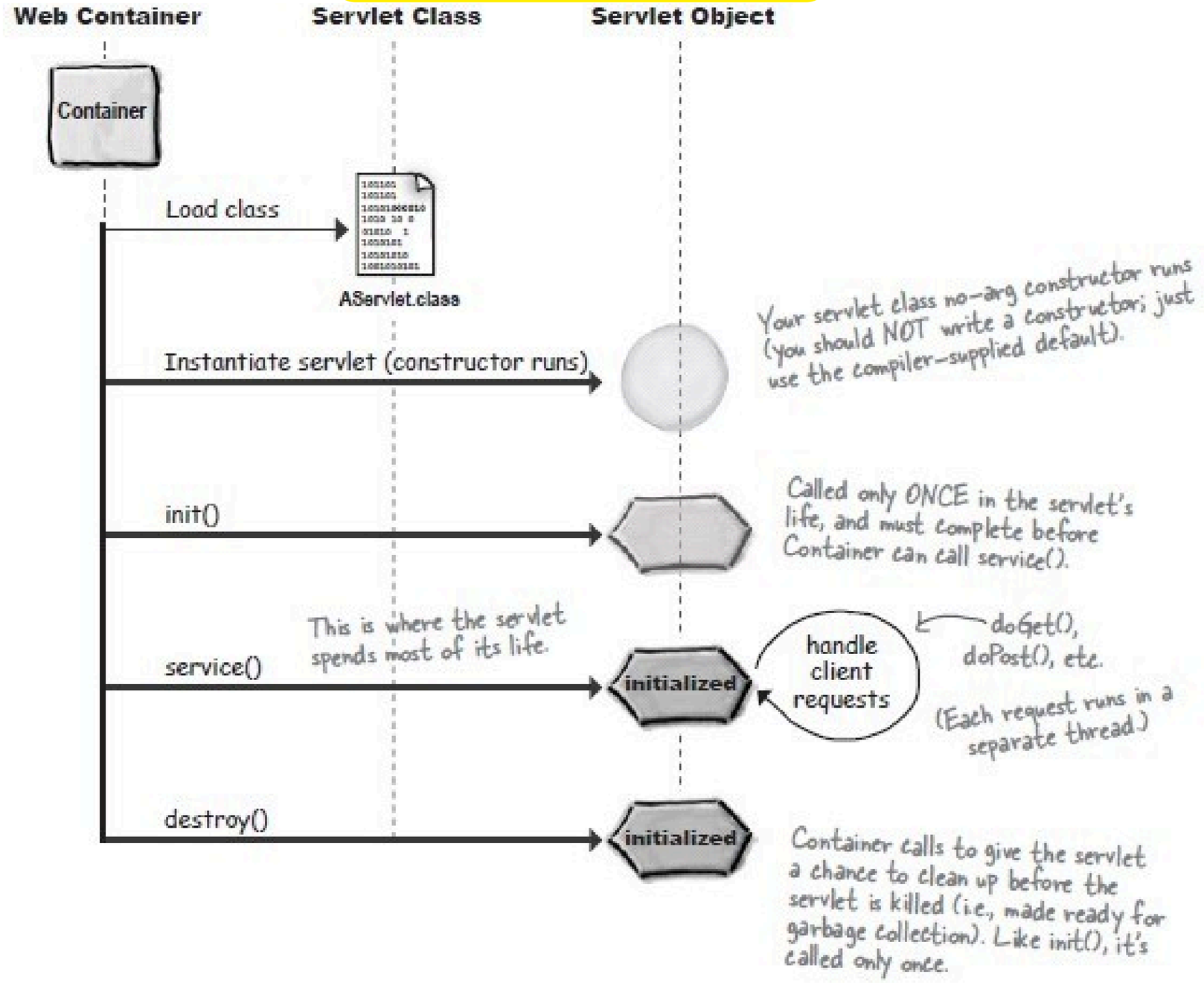
---

⑥

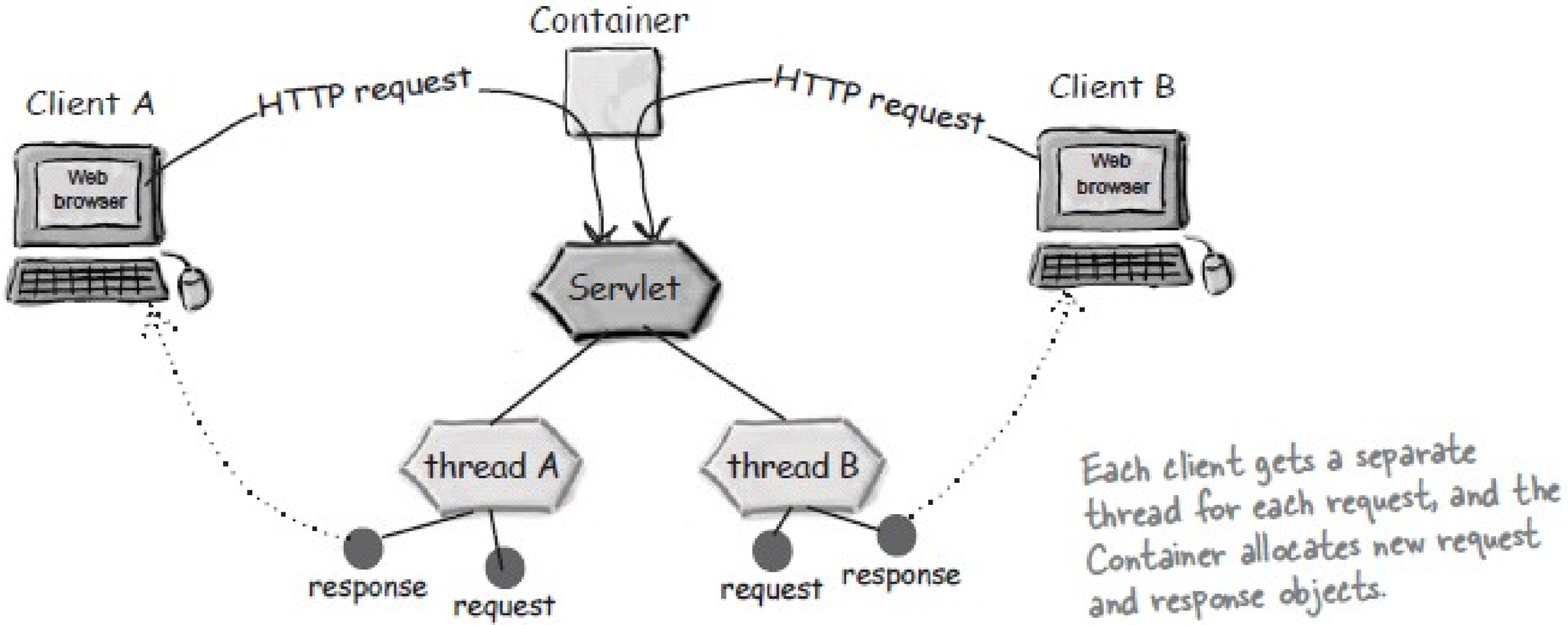


The thread completes, the container converts the response object into an HTTP response, sends it back to the client, then deletes the request and response objects.

Servlet Life Cycle



Each request runs in a separate thread!



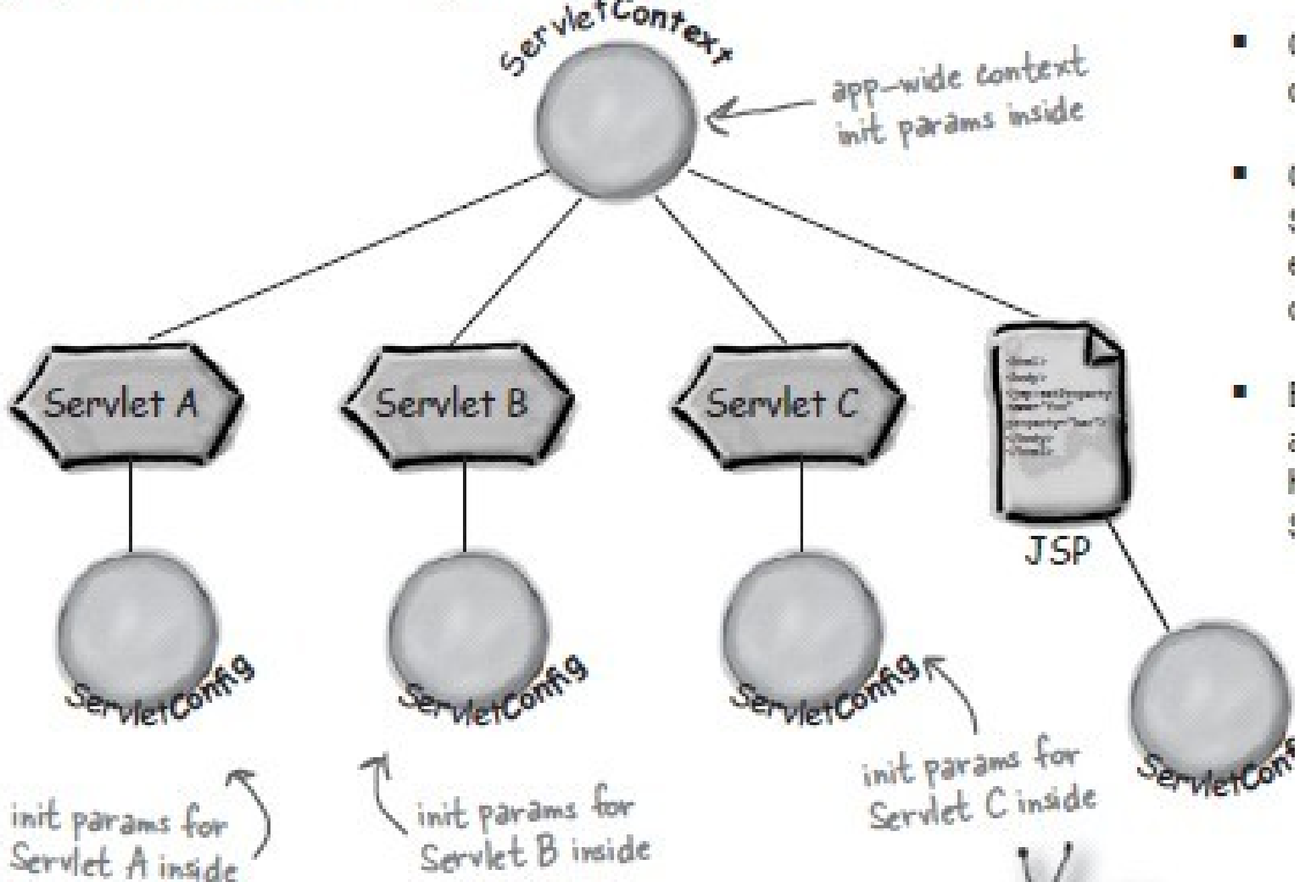
ServletContext vs. ServletConfig

ServletConfig vs ServletContext

ServletConfig is one per servlet

ServletContext is one per web app

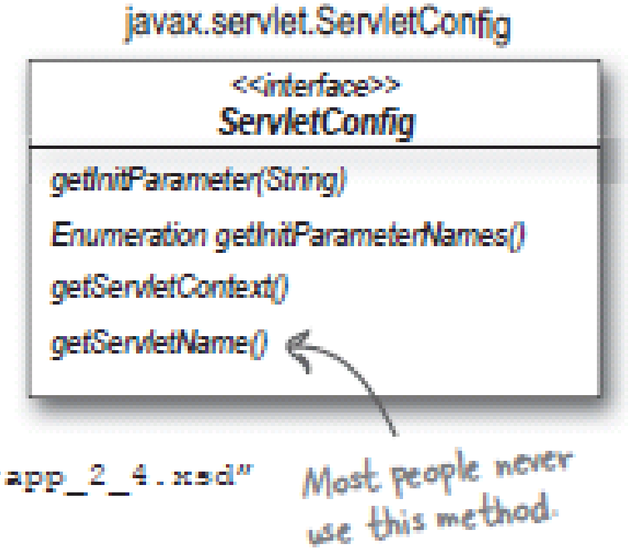
There's only one ServletContext for an entire web app, and all the parts of the web app share it. But each servlet in the app has its own ServletConfig. The Container makes a ServletContext when a web app is deployed, and makes the context available to each Servlet and JSP (which becomes a servlet) in the web app.



Setting ServletConfig

Testing your ServletConfig

ServletConfig's main job is to give you init parameters. It can also give you a ServletContext, but we'll usually get a context in a different way, and the getServletName() method is rarely useful.



In the DD (web.xml) file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
  <servlet>
    <servlet-name>BeerParamTests</servlet-name>
    <servlet-class>com.example.TestInitParams</servlet-class>
    <init-param>
      <param-name>adminEmail</param-name>
      <param-value>likewecare@wickedlysmart.com</param-value>
    </init-param>
    <init-param>
      <param-name>mainEmail</param-name>
      <param-value>blooper@wickedlysmart.com</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>BeerParamTests</servlet-name>
    <url-pattern>/Tester.do</url-pattern>
  </servlet-mapping>
</web-app>
```

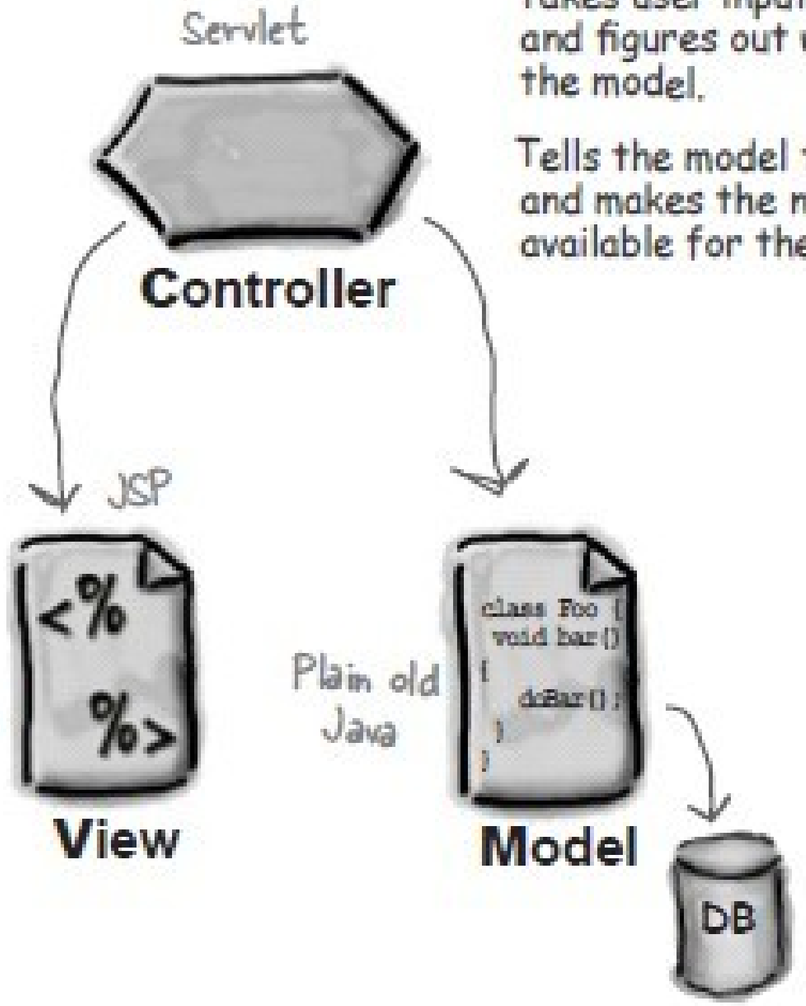
and getting it in Servlet...

In a servlet class:

```
package com.example;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class TestInitParams extends HttpServlet {

  public void doGet(HttpServletRequest request, HttpServletResponse response)
                                                    throws IOException, ServletException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("test init parameters<br>");

        java.util.Enumeration e = getServletConfig().getInitParameterNames();
        while(e.hasMoreElements()) {
           out.println("<br>param name = " + e.nextElement() + "<br>");
        }
        out.println("main email is " + getServletConfig().getInitParameter("mainEmail"));
        out.println("<br>");
        out.println("admin email is " + getServletConfig().getInitParameter("adminEmail"));
    }
  }
```

MVC

MVC with Servlet JSP



CONTROLLER

Takes user input from the request and figures out what it means to the model.

Tells the model to update itself, and makes the new model state available for the view (the JSP).

VIEW

Responsible for the presentation. It gets the state of the model from the Controller (although not directly; the Controller puts the model data in a place where the View can find it). It's also the part that gets the user input that goes back to the Controller.

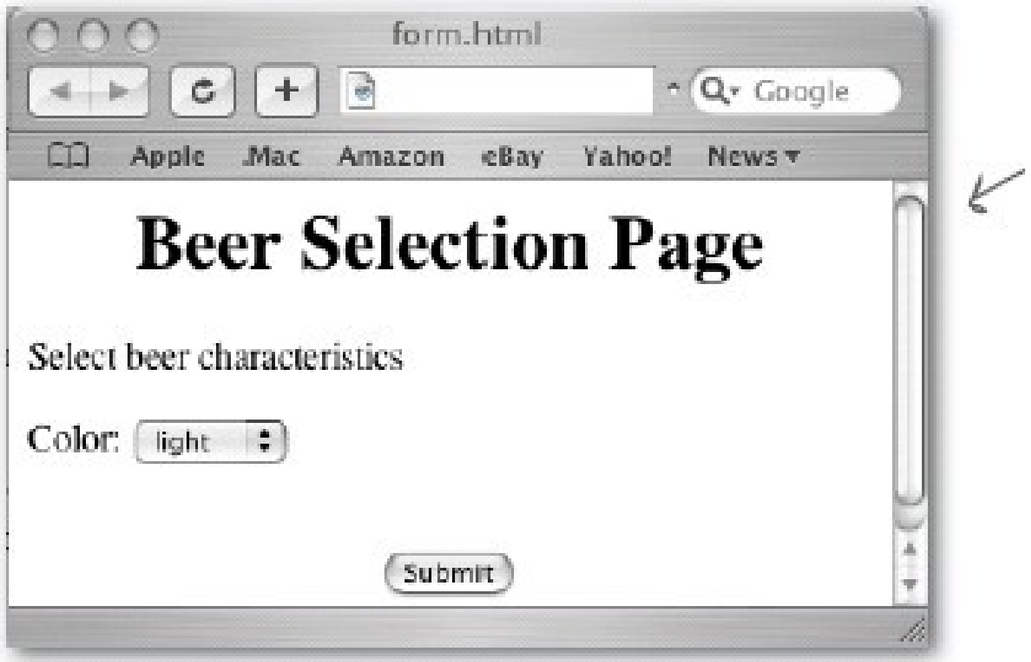
MODEL

Holds the real business logic and the state. In other words, it knows the rules for getting and updating the state.

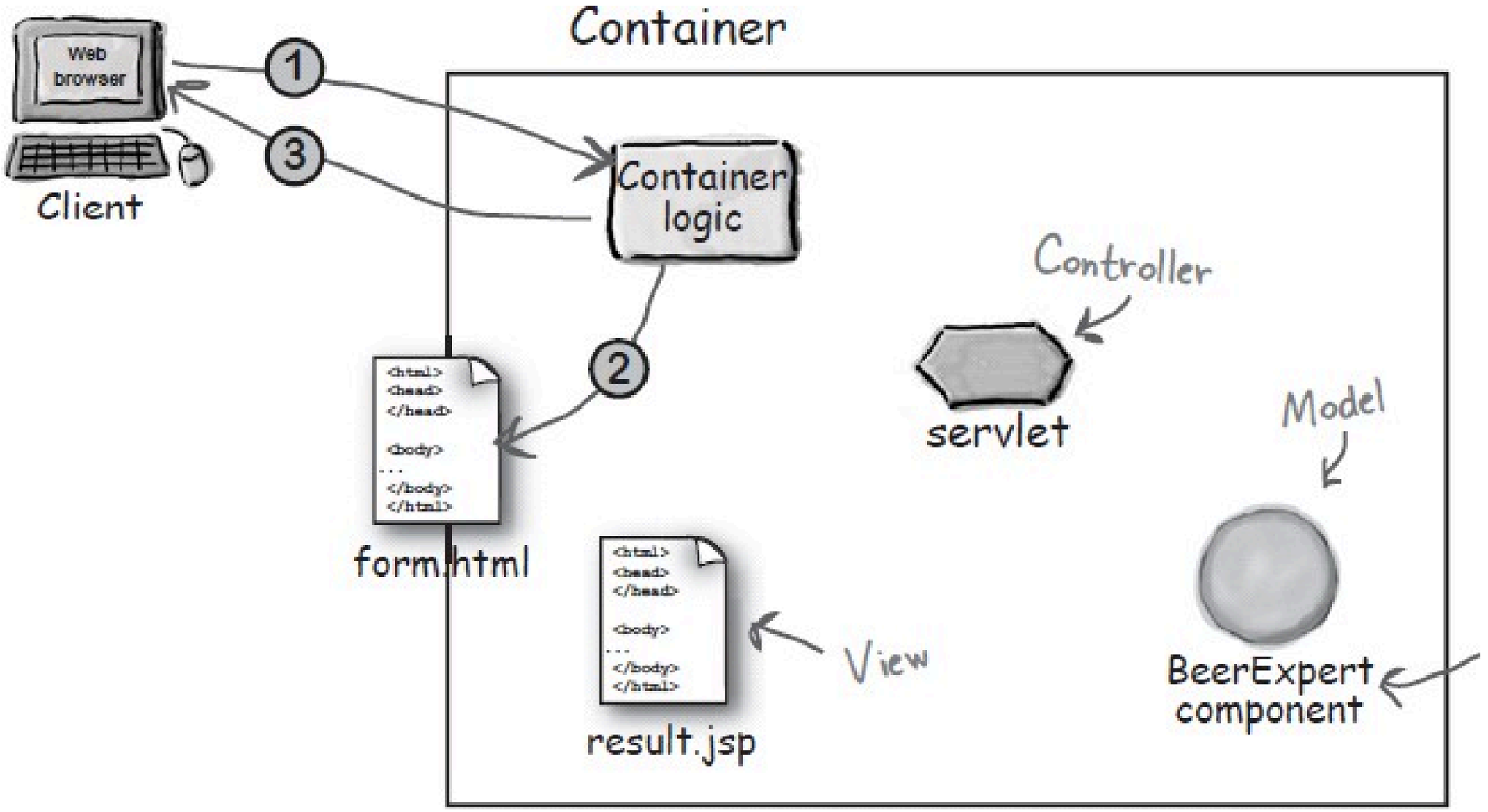
A Shopping Cart's contents (and the rules for what to do with it) would be part of the Model in MVC.

It's the only part of the system that talks to the database (although it probably uses *another* object for the actual DB communication, but we'll save that pattern for later...)

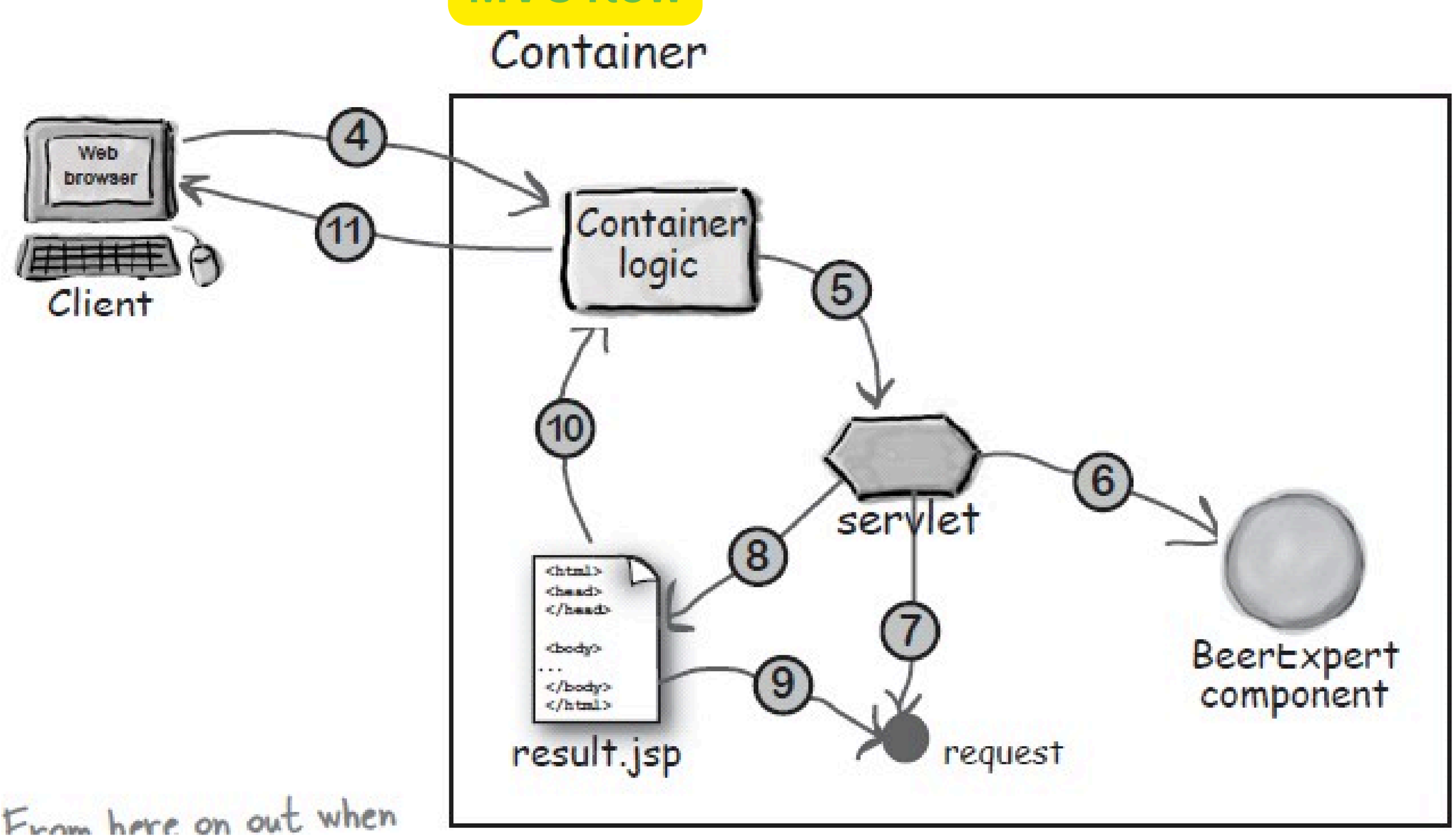
Beer Selection Page



MVC flow


Container
Web browser
1
3
Client
Container logic
2
Controller
servlet
Model
BeerExpert component
<html>
<head>
</head>
<body>
...
</body>
</html>
form.html
<html>
<head>
</head>
<body>
...
</body>
</html>
result.jsp
View

MVC flow

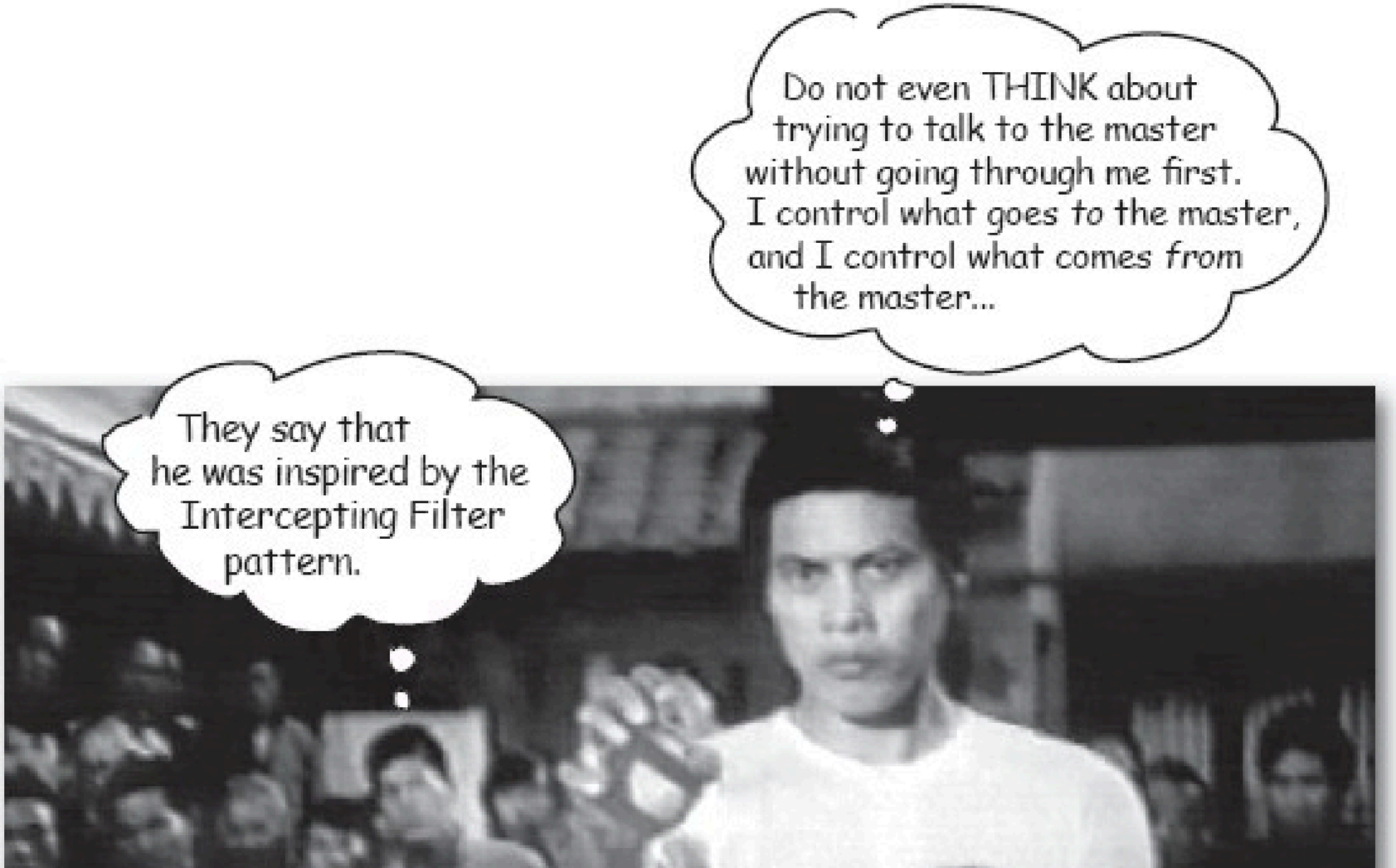


From here on out when

Servlet Filter API

Servlet Filter API

The Power of Filters

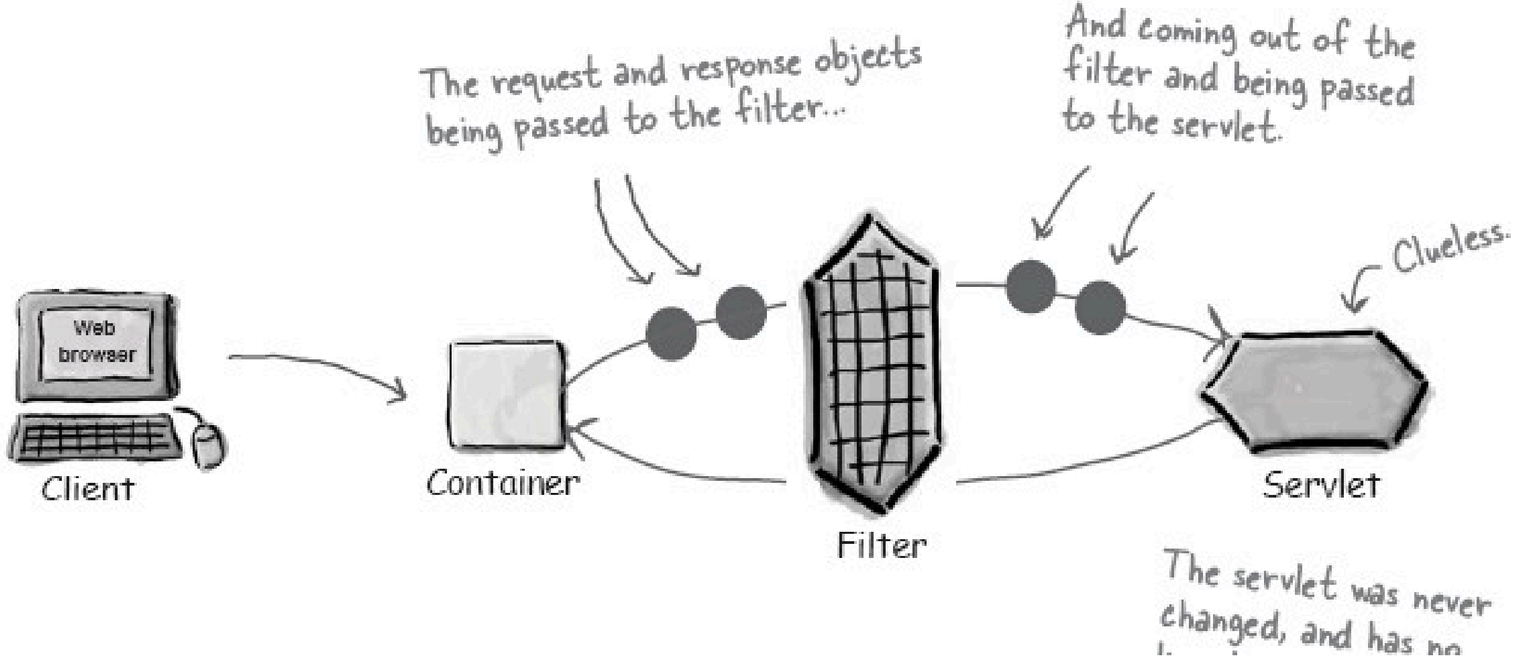


Servlet Filter API

How about some kind of "filter"?

Filters are Java components—very similar to servlets—that you can use to intercept and process requests *before* they are sent to the servlet, or to process responses *after* the servlet has completed, but *before* the response goes back to the client.

The Container decides when to invoke your filters based on declarations in the DD. In the DD, the deployer maps which filters will be called for which request URL patterns. So it's the deployer, not the programmer, who decides which subset of requests or responses should be processed by which filters.



Servlet Filter API

Request filters can:

- perform security checks
- reformat request headers or bodies
- audit or log requests

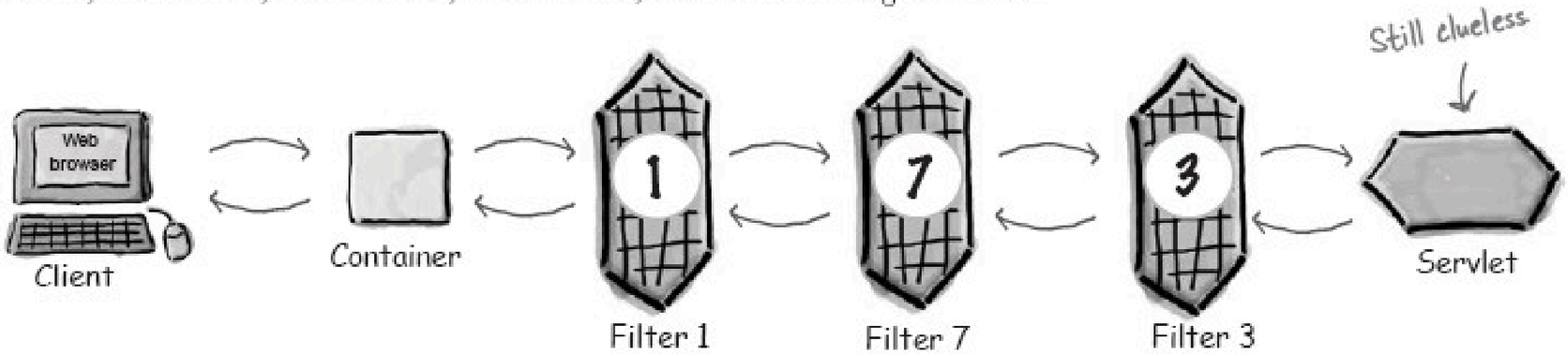
Response filters can:

- compress the response stream
- append or alter the response stream
- create a different response altogether

Filters are modular, and configurable in the DD

DD configuration 1:

Using the DD, you can link them together by telling the Container: "For these URLs, run filter 1, then filter 7, then filter 3, then run the target servlet."



DD configuration 2:

Then, with a quick change to the DD, you can delete and swap them with: "For these URLs, run filter 3, then filter 7, and then the target servlet."