Introduction to Web Application

Element types

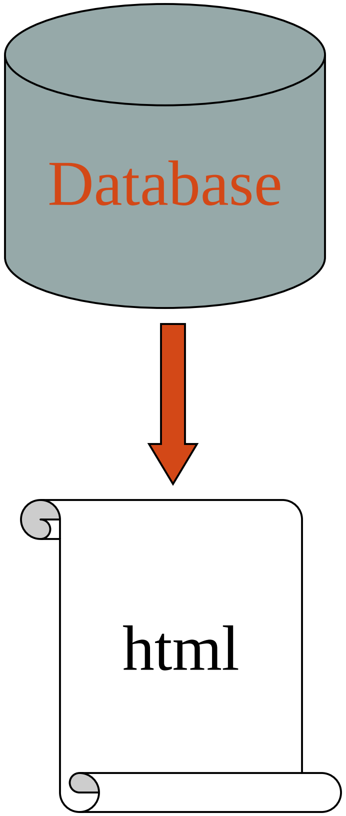
Directive

Life cycle

Data-Driven Websites

- Websites that provide access to:
  - Lots of data
  - Dynamic data
  - Customized views of data
  - E.g. ebay.com

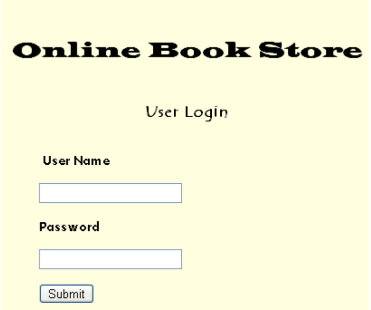
- Scripts map data to html



GET/Home.html HTTP/ 1.1
Host: www.onebookstore.com
User-Agent: Mozilla/5.0(Windows..
…..

http://www.onebookstore.com/Home.html

USER



Online Book Store

User Login

User Name

Password

Submit

HTTP/1.1 200 OK
Set-Cookie:…
…
<html><body>
<h1 align=….

Finds the page and generates Http response

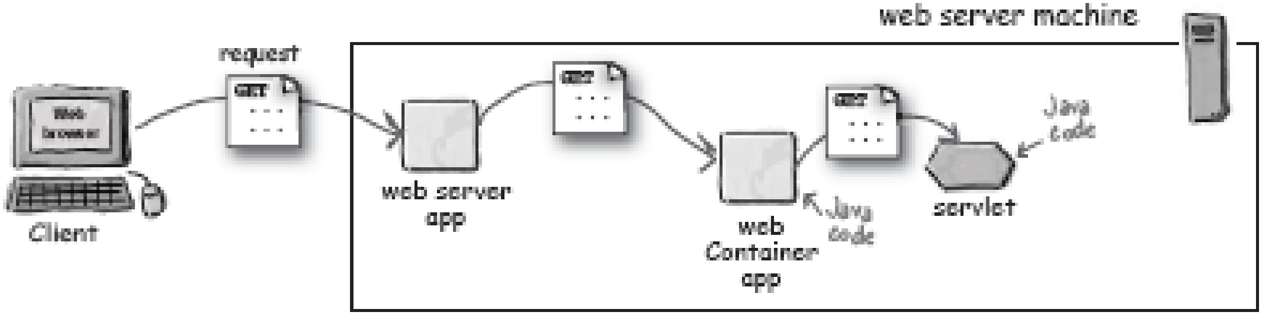
Http Request

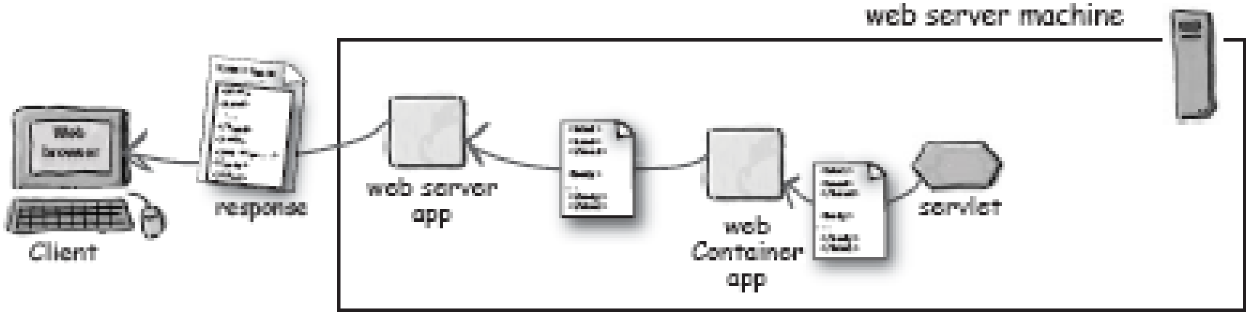
- ***HTTP GET***
  - The total amount of characters in a GET is really limited (depending on the server)
  - The data you send with the GET is appended to the URL up in the browser bar, so whatever you send is exposed
  - Because of this, the user can bookmark a form submission if you use GET
- HTTP POST
  - The data in included in the request body
  - More data can be sent

Servlets

- The purpose of a servlet is to create a Web page in response to a client request
- Servlets are written in Java, with a little HTML mixed in
  - The HTML is enclosed in `out.println( )` statements

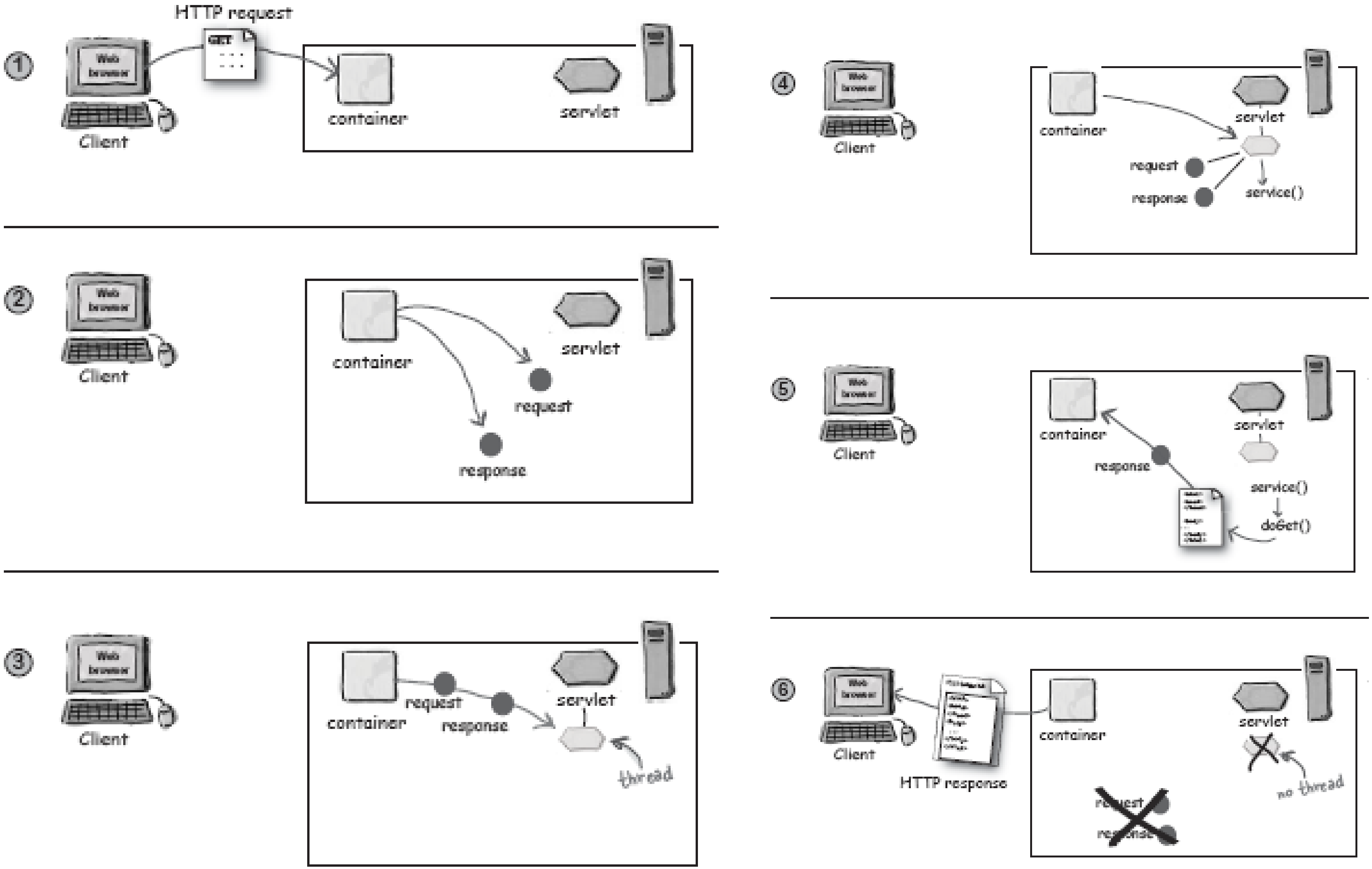
Web Server vs Web Container





Web Container

- Communications support
  - The container provides an easy way for your servlets to talk to web server. You don't have to build a ServerSocket, listen on a port, create streams, etc.
  - The Container knows the protocol between the web server and itself,
- Lifecycle Management
  - It takes care of loading the classes, instantiating and initializing the servlets, invoking the servlet methods, and making servlet instances eligible for garbage collection
- Multithreading Support
  - The Container automatically creates a new Java thread for every servlet request it receives
- Declarative Security
  - With a Container, you get to use an XML deployment descriptor to configure (and modify) security without having to hard-code it into your servlet (or any other) class code
  - You can manage and change your security without touching and recompiling your Java source files.
- JSP Support
  - The container takes care of translating a jsp file into java code


1
HTTP request
GET
Web browser
Client
container
servlet
2
Web browser
Client
container
servlet
request
response
3
Web browser
Client
container
request
response
servlet
thread
4
Web browser
Client
container
servlet
request
response
service()
5
Web browser
Client
container
servlet
response
service()
doGet()
6
Web browser
Client
HTTP response
container
servlet
no thread
request
response

A “Hello World” servlet

(from the Tomcat installation documentation)

```
public class HelloServlet extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse
response)       throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>\n" +
                "<HEAD><TITLE>Hello</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1>Hello World</H1>\n" +
                "</BODY></HTML>");
  }
}
```

A more meaningful one

- public class HelloServlet extends HttpServlet {
- List<StudyMaterials> stList=new ArrayList<…> stList();
- public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
- response.setContentType("text/html");
- PrintWriter out = response.getWriter();
- for(StudyMaterials st: this.stList)
- out.println(“Title “ + stList.getTitle() + “ URL: “ + stList.getURL());
- }
- }

Deployment Descriptor

```
<?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
http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd”    version="2.5“>

<servlet>
   <servlet-name>Form1</servlet-name>
   <servlet-class>book.HelloServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>Form1</servlet-name>
            <url-pattern>/store/home.do</url-pattern>
</servlet-mapping>
</web-app>
```

The <servlet> element tells the Container which class files belong to a particular web application.

Think of the <servlet-mapping> element as what the Container uses at runtime when a request comes in, to ask, "which servlet should I invoke for this requested URL?"

Resultant URL

–

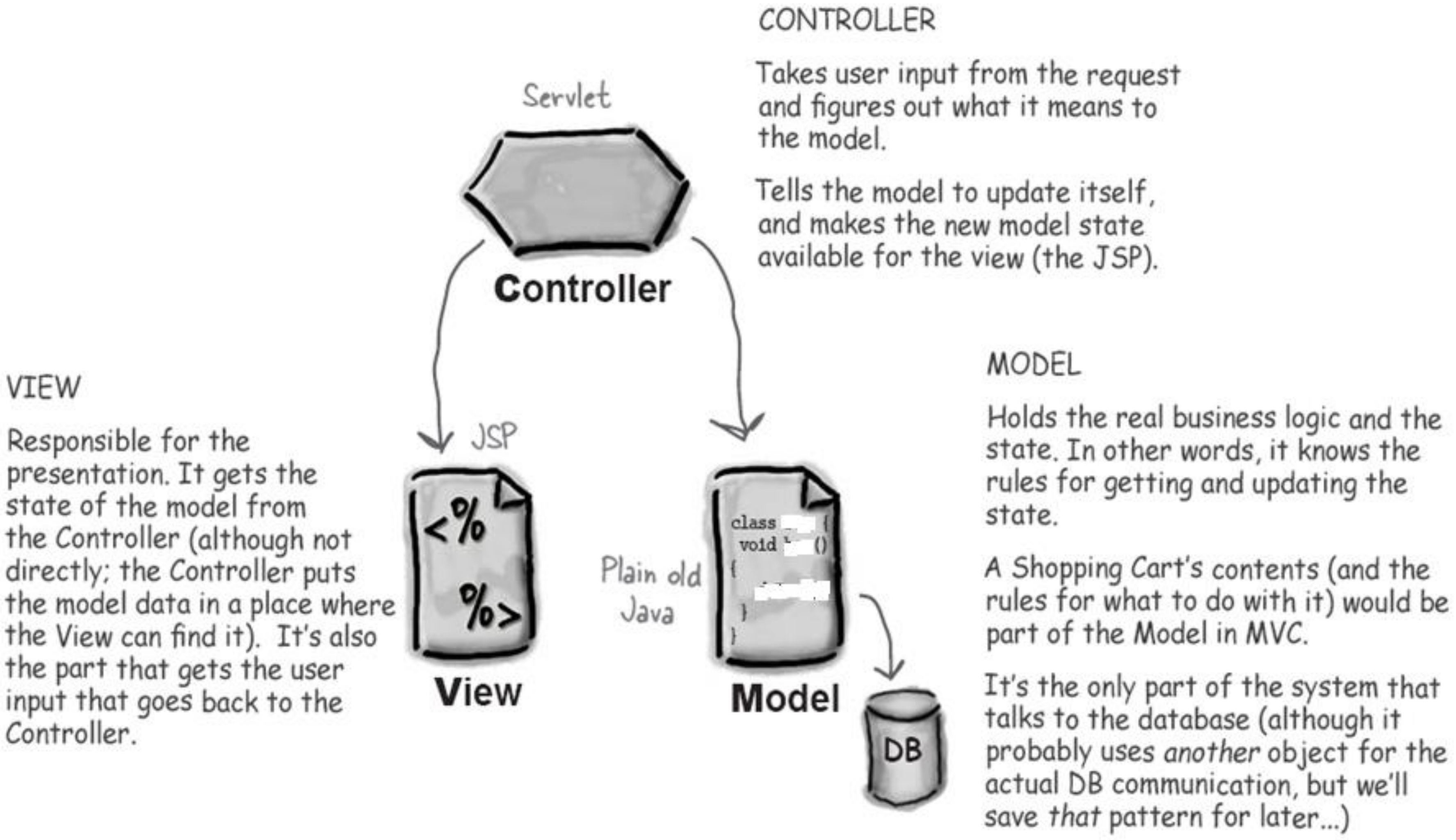
http://*hostname/webappName/MyAddress*

Deployment Descriptor

- **The deployment descriptor (DD), provides a “declarative” mechanism for customizing your web applications without touching source code!**
- **The actual path name for a servlet is not specified**
  - Minimizes touching source code that has already been tested.
  - Lets you fine-tune your app’s capabilities, even if you don’t *have the source code.*
  - Lets you adapt your application to different resources (like databases), without having to recompile and test any code.
  - Makes it easier for you to maintain dynamic security info like access control lists and security roles.
  - Lets non-programmers modify and deploy your web applications

MVC

- **Model*View*Controller (MVC) takes the business logic out of the servlet, and puts it in a "Model"— a reusable plain old Java class.**
  - **The Model is a combination of the business data (like the state of a Shopping Cart) and the methods (rules) that operate on that data.**



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

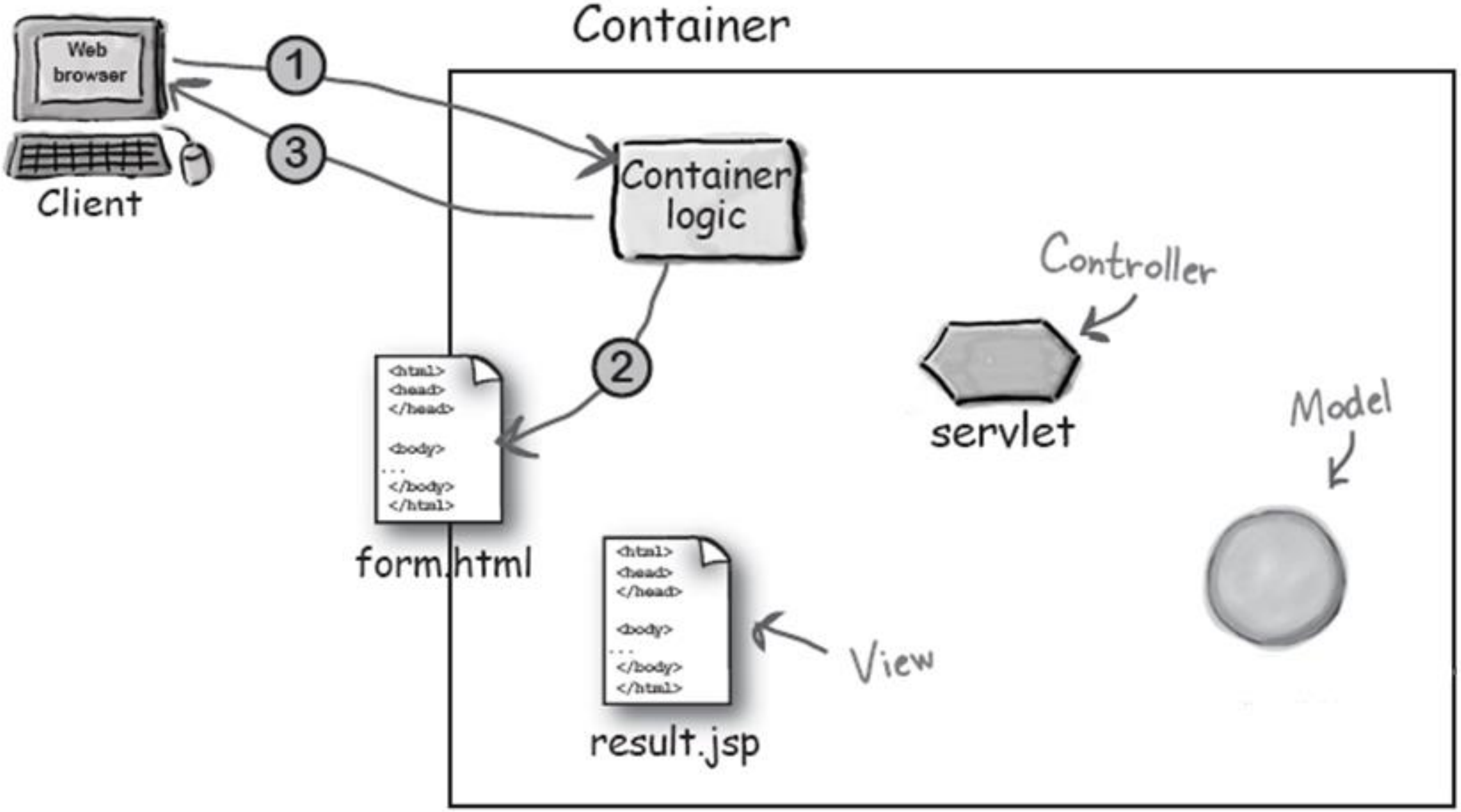
Responsible for the presentation. It gets the state of the model from the Controller (although not directly; the Controller puts the model data in a place where the View can find it). It's also the part that gets the user input that goes back to the Controller.

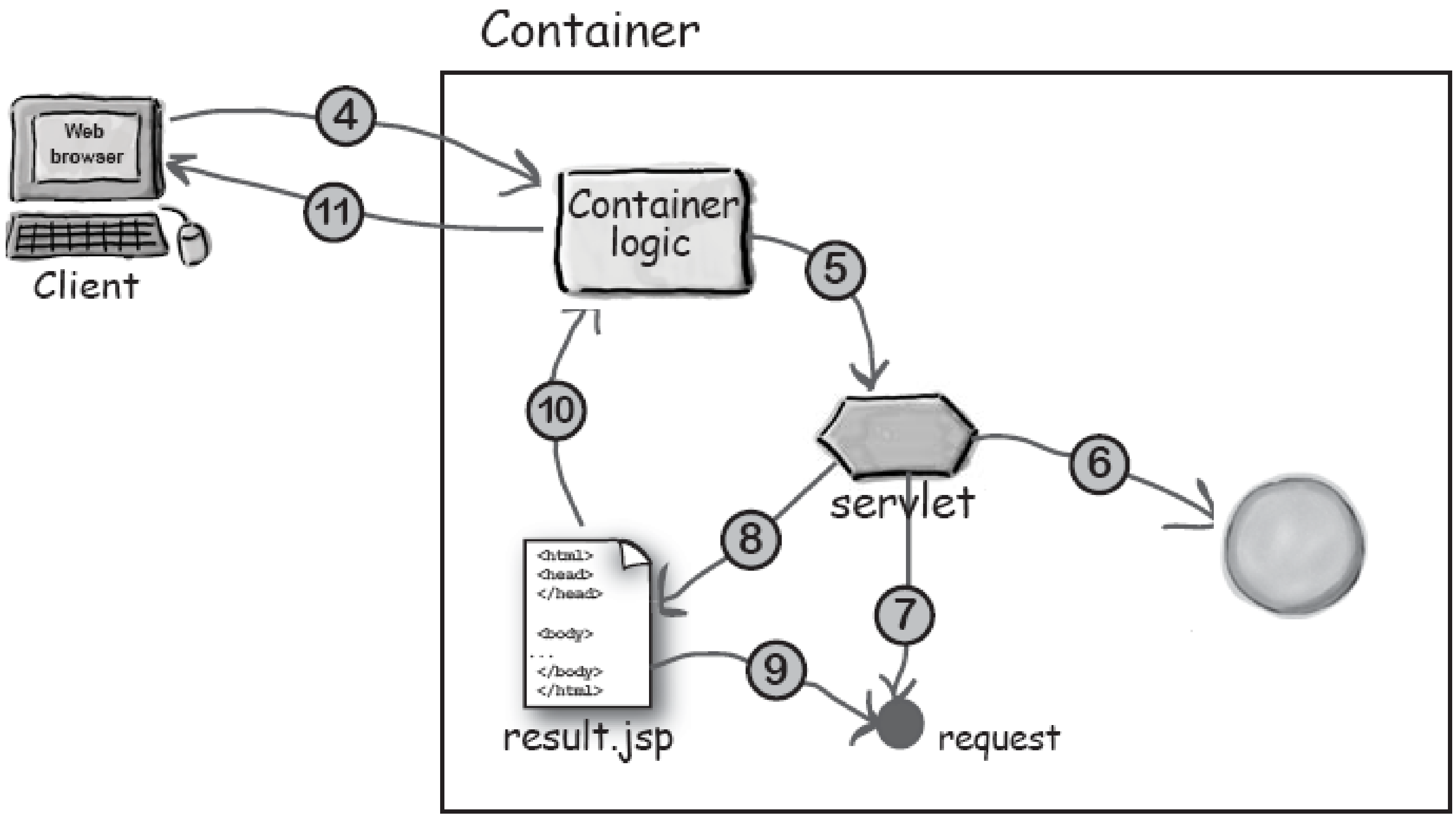
MODEL

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

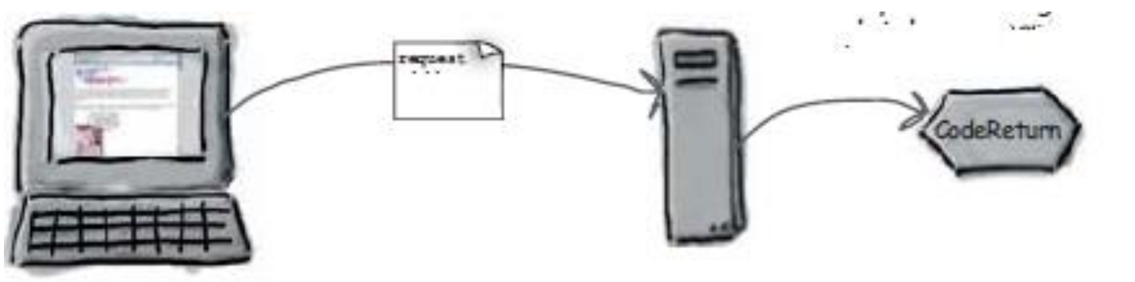
A Shopping Cart's contents (and the rules for what to do with it) would be part of the Model in MVC.

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...)

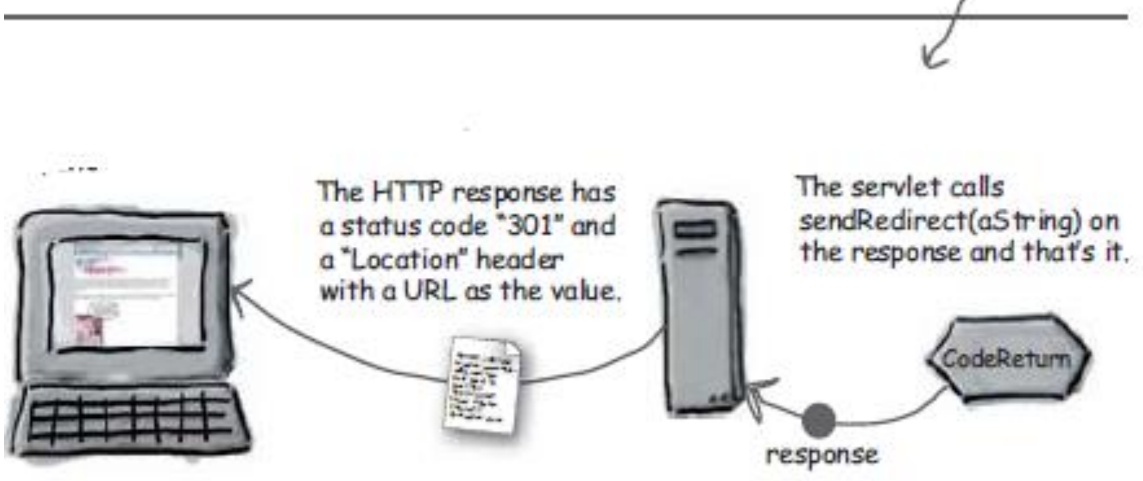

Container
Web browser
Client
1
3
Container logic
2
<html>
<head>
</head>
<body>
...
</body>
</html>
form.html
Controller
servlet
Model
<html>
<head>
</head>
<body>
...
</body>
</html>
result.jsp
View


Container
Web browser
Client
4
11
Container logic
5
servlet
6
7
8
9
10
<html>
<head>
</head>
<body>
...
</body>
</html>
result.jsp
request

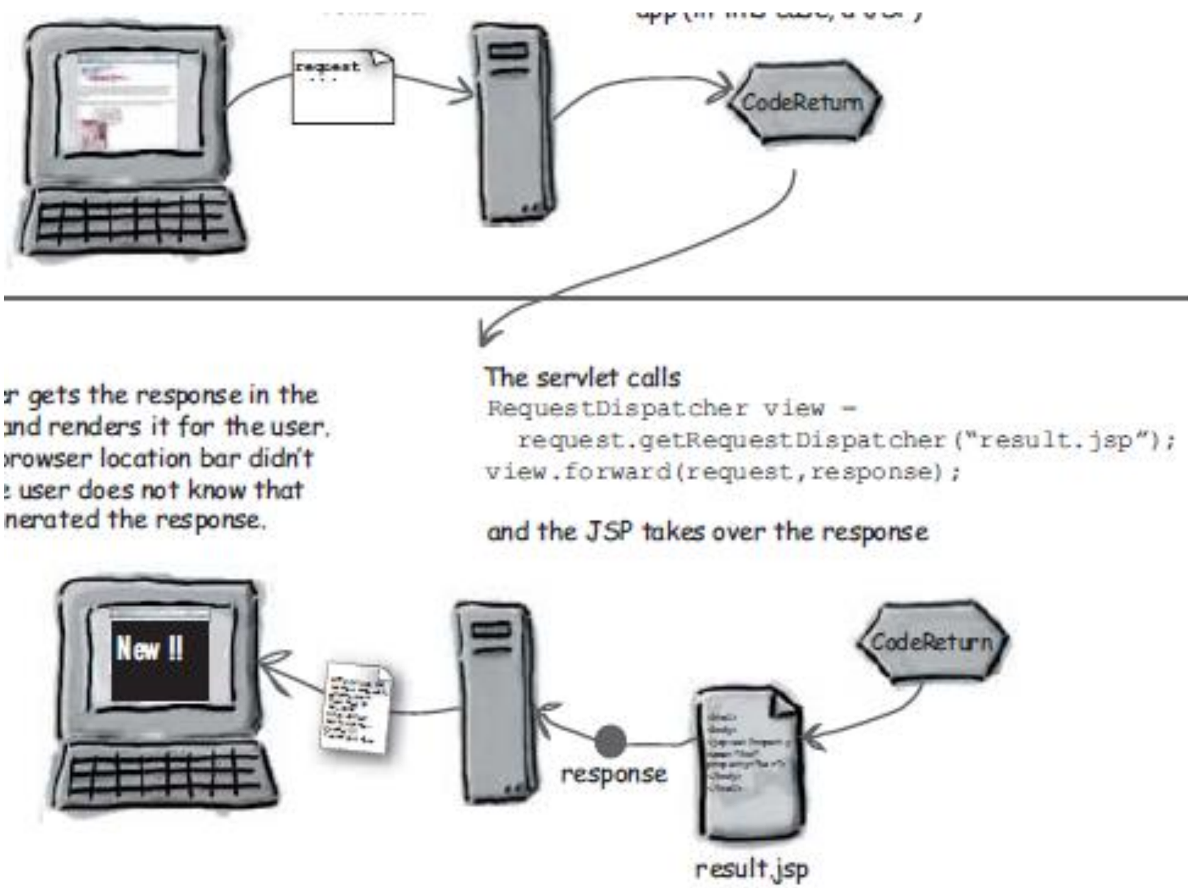
Send Redirect



response.sendRedirect(http://www.abc.in);


The HTTP response has a status code "301" and a "Location" header with a URL as the value.
The servlet calls sendRedirect(aString) on the response and that's it.
CodeReturn
response

Request Dispatcher

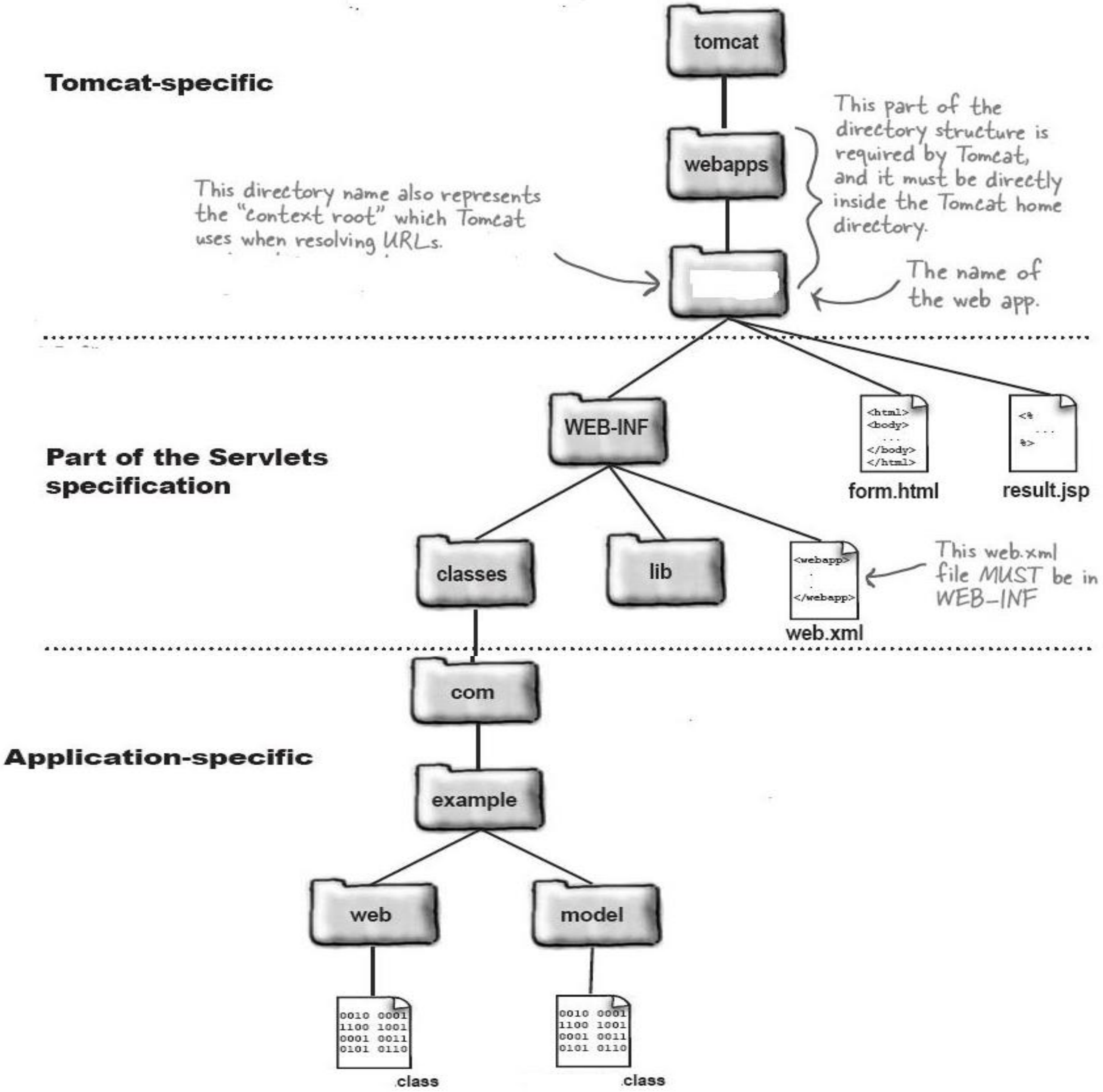


r gets the response in the
ınd renders it for the user.
ɔrowser location bar didn't
: user does not know that
nerated the response.

The servlet calls

```
RequestDispatcher view =
  request.getRequestDispatcher("result.jsp");
view.forward(request,response);
```

and the JSP takes over the response



Tomcat-specific

Part of the Servlets specification

Application-specific

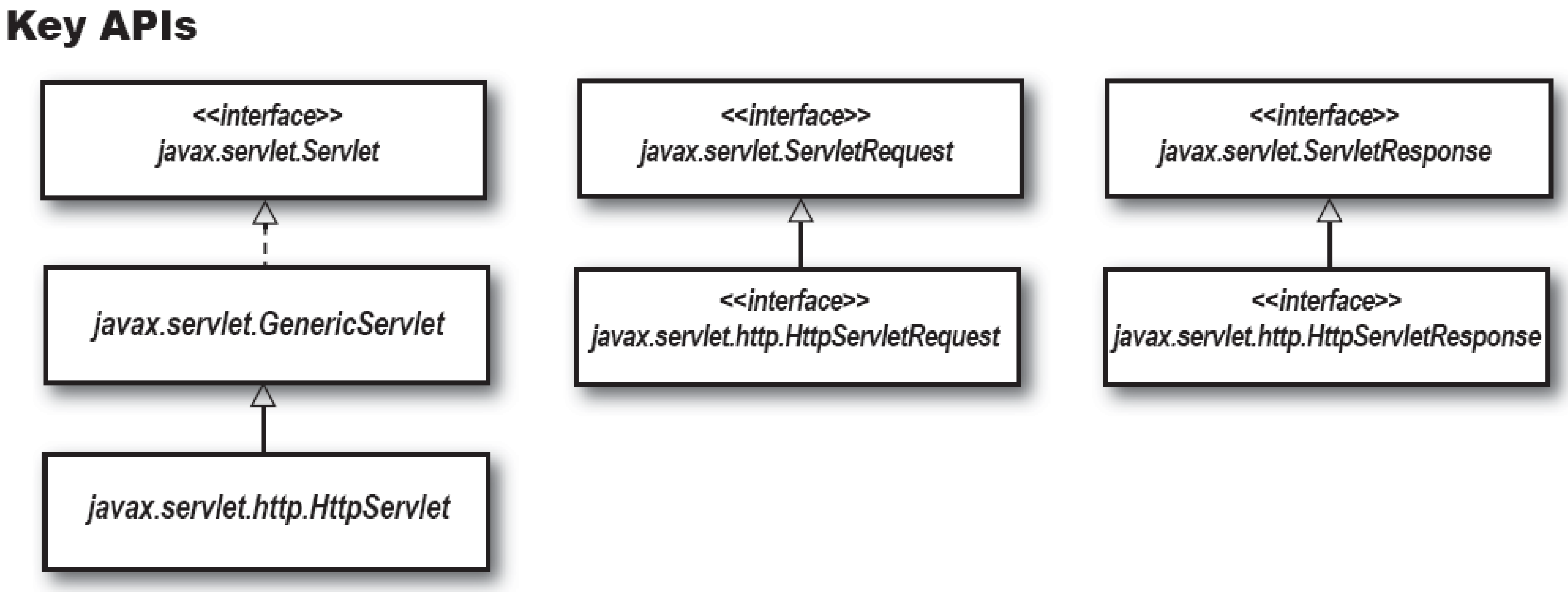
This part of the directory structure is required by Tomcat, and it must be directly inside the Tomcat home directory.

This directory name also represents the "context root" which Tomcat uses when resolving URLs.

The name of the web app.

This web.xml file MUST be in WEB-INF

APIs



Key APIs

<<interface>>
javax.servlet.Servlet

javax.servlet.GenericServlet

javax.servlet.http.HttpServlet

<<interface>>
javax.servlet.ServletRequest

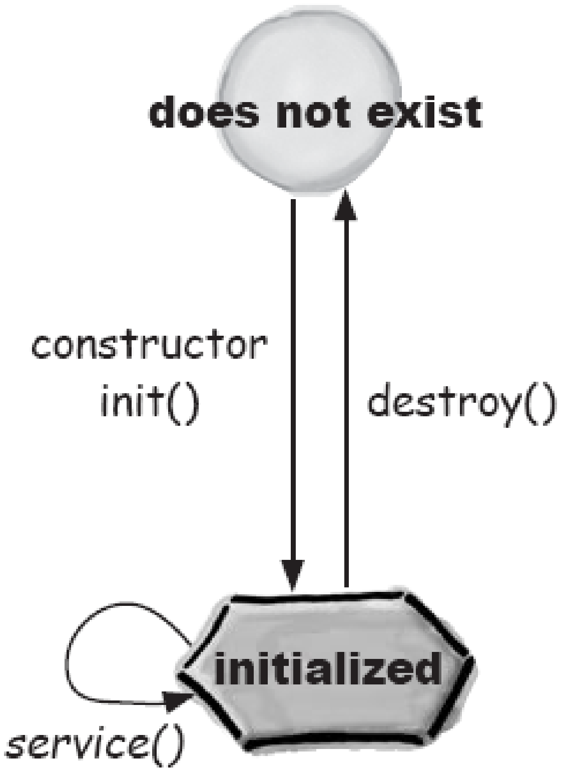
<<interface>>
javax.servlet.http.HttpServletRequest

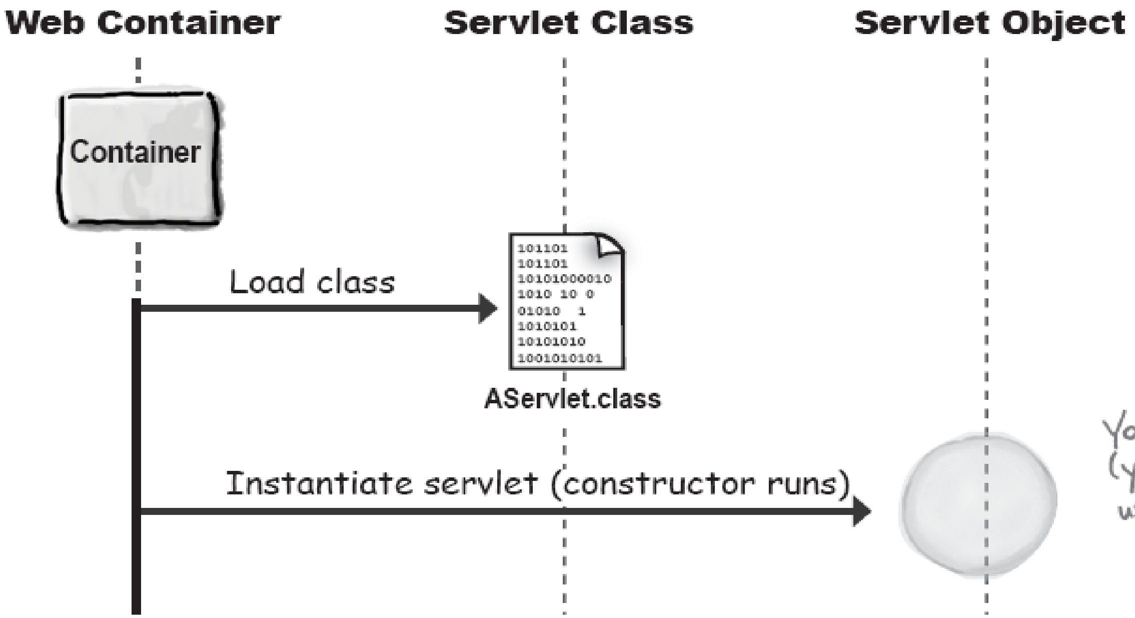
<<interface>>
javax.servlet.ServletResponse

<<interface>>
javax.servlet.http.HttpServletResponse

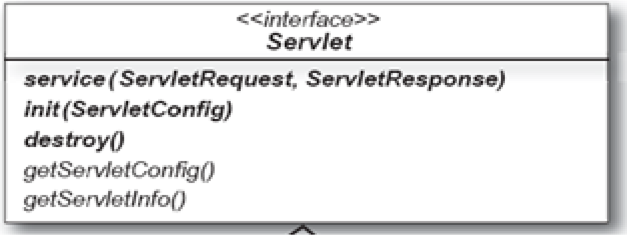
- Compiling your servlet
  - Javac –classpath /…./tomcat/common/lib/servlet-api.jar; <servlet name>

Servlet Lifecycle





Your servlet class no-arg constructor runs (you should NOT write a constructor; just use the compiler-supplied default).



<<interface>>
Servlet

service(ServletRequest, ServletResponse)
init(ServletConfig)
destroy()
getServletConfig()
getServletInfo()

Servlet interface
(javax.servlet.Servlet)

The Servlet interface says that all servlets have these five methods (the three in bold are lifecycle methods).

Three Big Lifecycle Moments

- Init()
  - If you have initialization code (like getting a database connection or registering yourself with other objects), then you'll override the init() method in your servlet class
- Service()
  - You should NOT override the service() method. Your job is to override the doGet() and/or doPost() methods and let the service() implementation from HTTPServlet worry about calling the right one.
- doGet() or doPost()
  - Whichever one(s) you override tells the Container what you support

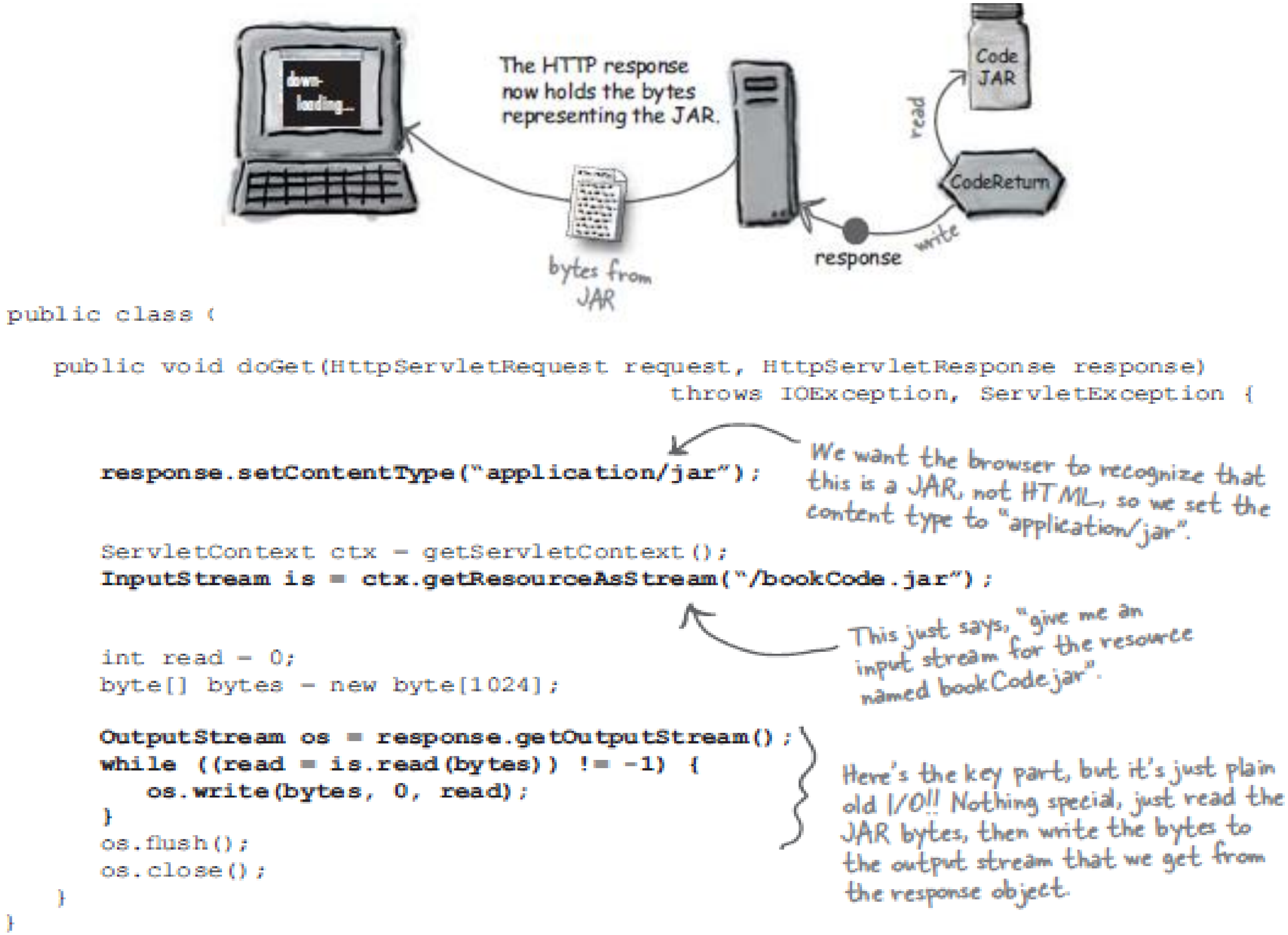
What makes an object a servlet

- ServletConfig Object
  - One ServletConfig object per servlet
  - Use it to pass deploy-time information to the servlet (a database for example) that you don't want to hard-code into the servlet (servlet init parameters)
  - Use it to access the ServletContext.
  - Parameters are configured in the Deployment Descriptor.

What makes an object a servlet

- One ServletContext per web app
- Use it to access web app *parameters* (also configured in the Deployment Descriptor).
- Use it as a kind of *application bulletin-board,* where you can put up messages (called *attributes*) that other parts of the application can access.
- Use it to get *server info,* including the name and version of the Container, and the version of the API that's supported.

Download a JAR



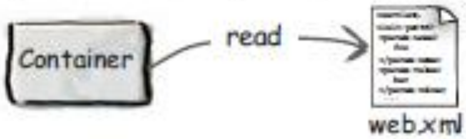
```
public class {

   public void doGet(HttpServletRequest request, HttpServletResponse response)
                                          throws IOException, ServletException {

      response.setContentType("application/jar");

      ServletContext ctx = getServletContext();
      InputStream is = ctx.getResourceAsStream("/bookCode.jar");

      int read = 0;
      byte[] bytes = new byte[1024];

      OutputStream os = response.getOutputStream();
      while ((read = is.read(bytes)) != -1) {
         os.write(bytes, 0, read);
      }
      os.flush();
      os.close();
   }
}
```

We want the browser to recognize that this is a JAR, not HTML, so we set the content type to "application/jar".

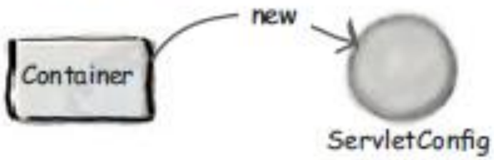
This just says, "give me an input stream for the resource named bookCode.jar".

Here's the key part, but it's just plain old I/O!! Nothing special, just read the JAR bytes, then write the bytes to the output stream that we get from the response object.

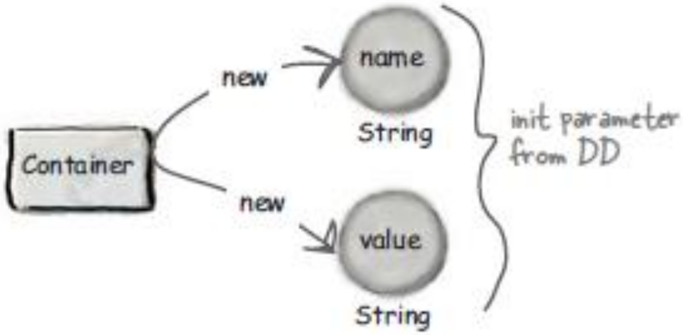
1. Container reads the Deployment Descriptor for this servlet, including the servlet init parameters (<init-param>).



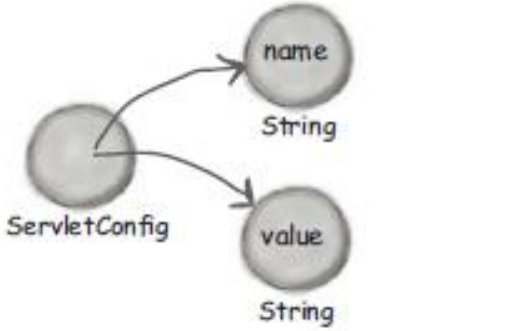
2. Container creates a new ServletConfig instance for this servlet.



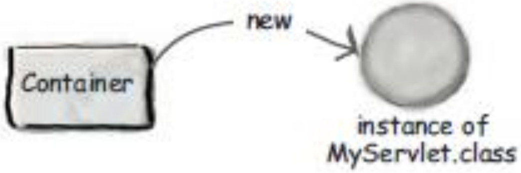
3. Container creates a name/value pair of Strings for each servlet init parameter. Assume we have only one.



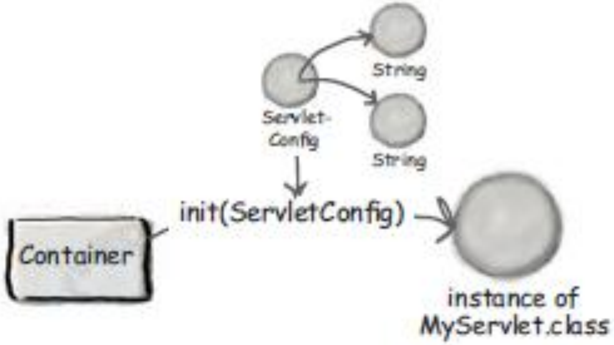
4. Container gives the ServletConfig references to the name/value init parameters.



5. Container creates a new instance of the servlet class.



6. Container calls the servlet's init() method, passing in the reference to the ServletConfig.



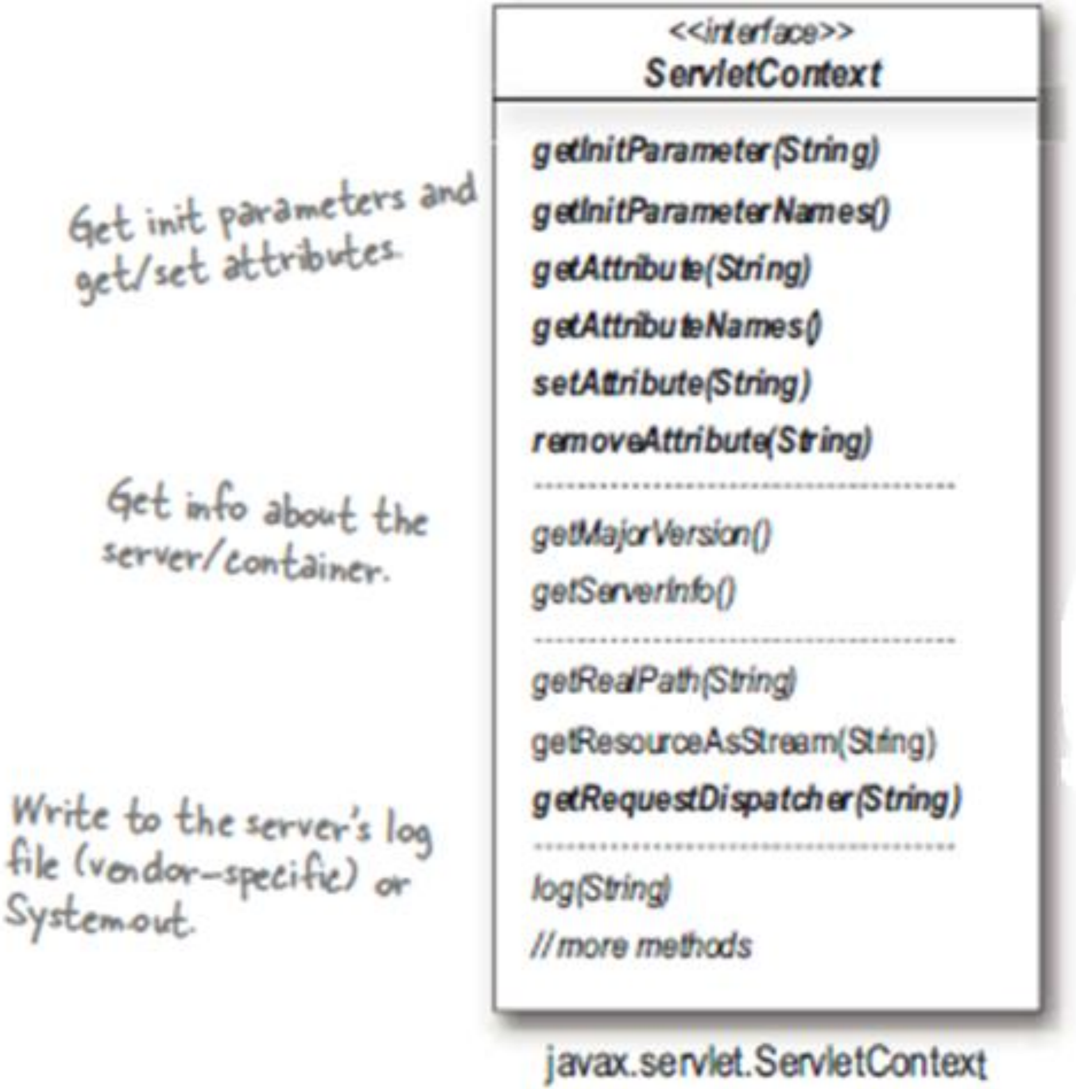
ServletConfig parameters

- <servlet>
  - <servlet-name>InitTest</servlet-name>
  - <servlet-class>moreservlets.InitServlet</servlet-class>
  - <init-param>
    - <param-name>firstName</param-name>
    - <param-value>BCSE4</param-value>
  - </init-param>
  - <init-param>
    - <param-name>emailAddress</param-name>
    - <param-value>abc@jdvu.ac.in</param-value>
  - </init-param>
- </servlet>
- <servlet-mapping>
  - <servlet-name>InitTest</servlet-name>
  - <url-pattern>/showInitValues</url-pattern>
- </servlet-mapping>

Read Config Parameter

```
public class InitServlet extends HttpServlet {
    private String firstName, emailAddress;
    public void init() {
        ServletConfig config = getServletConfig();
        firstName =config.getInitParameter("firstName");
        if (firstName == null) {
            firstName = "Missing first name";
        }
        emailAddress =config.getInitParameter("emailAddress");
        if (emailAddress == null) {
            emailAddress = "Missing email address";
        }
}
```

ServletContext Parameter



<<interface>>

ServletContext

getInitParameter(String)

getInitParameterNames()

getAttribute(String)

getAttributeNames()

setAttribute(String)

removeAttribute(String)

getMajorVersion()

getServerInfo()

getRealPath(String)

getResourceAsStream(String)

getRequestDispatcher(String)

log(String)

//more methods

javax.servlet.ServletContext

Get init parameters and get/set attributes.

Get info about the server/container.

Write to the server's log file (vendor-specific) or Systemout.

getServletConfig().getServletContext().getinitParameter()

ServletContext Parameter

- <?xml version="1.0" encoding="ISO-8859-1" ?>
- <web-app …>
- <context-param>
- <param-name>**email**</param-name>
- <param-value>**abc@gmail.com**</param-value>
- </context-param>
- </web-app>

  - Use getServletContext().getInitParameter() to read context params from servlet

Attributes

- An attribute is a name/value pair in a map instance variable
- An attribute is either bound to a ServletContext, HttpServletRequest or an HttpSession object
- There is no servlet specific attribute
- Return type of an attribute is an object