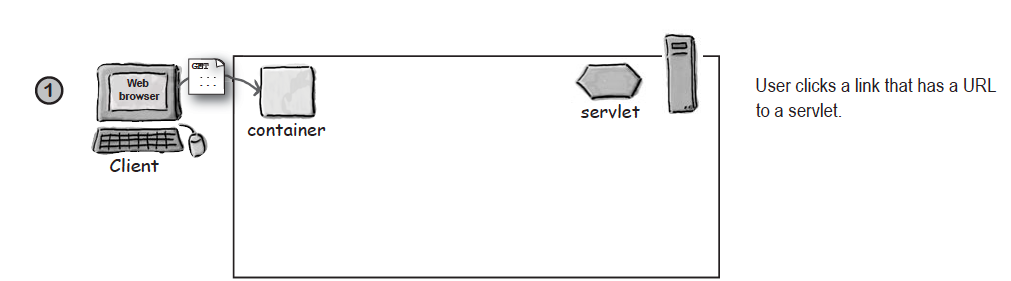
**Chapter 6**

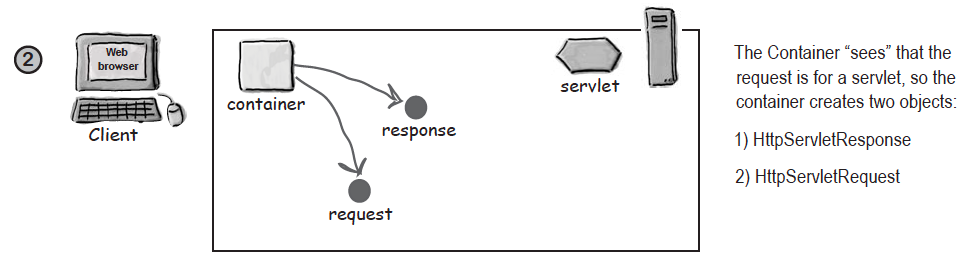
Servlets live to service clients, a servlet's job is to take a client's request and send back a response. The request might be simple: “get me the Welcome Page” or might be complex: “Complete my shopping cart check out”.

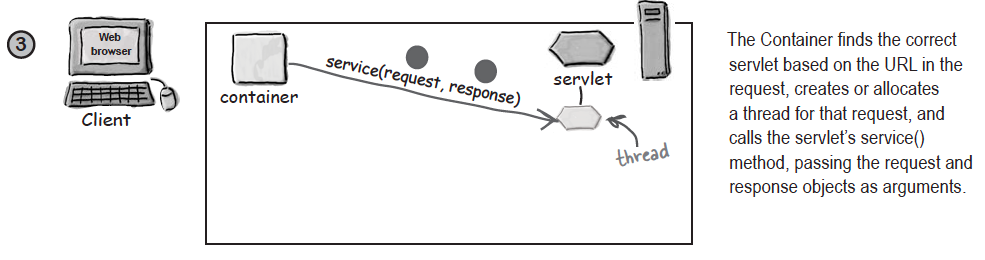
The request carries crucial data, and your servlet code should know how to find it and how to use it. The response carries the info the browser needs to render a page and your servlet code should know how to send it or not your servlet can decide to pass the request to something else(another page, servlet or JSP) instead.

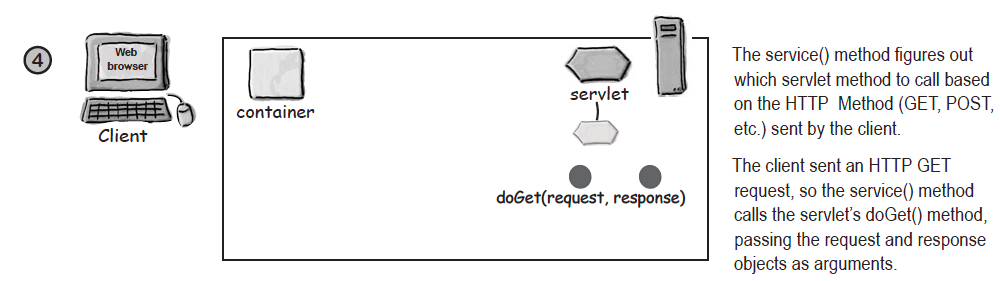
As we know that servlets are controlled by the Container(Apache Tomcat in our case).

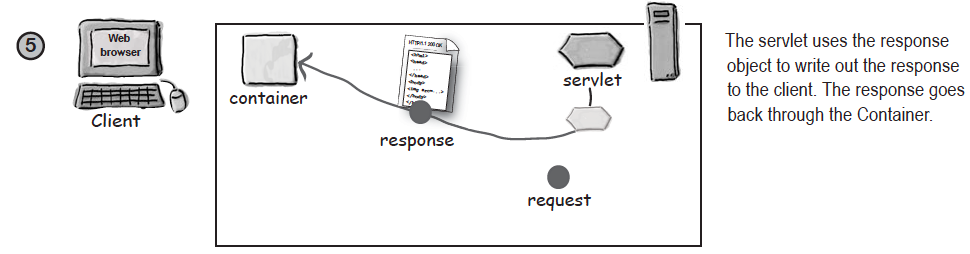
In the previous chapters we looked at the Container's overall role in a servlet's life, it creates the request and response objects, creates or allocates a new thread for the servlet and calls the servlet's service() method passing the request and response references as arguments. Here's a quick review:

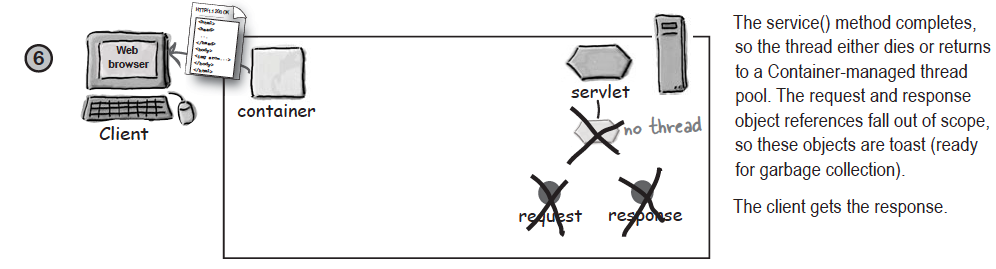






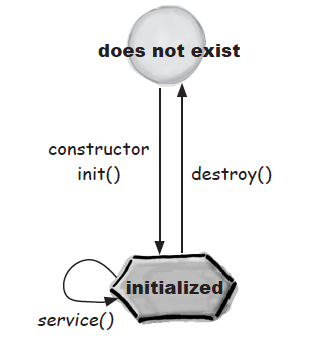




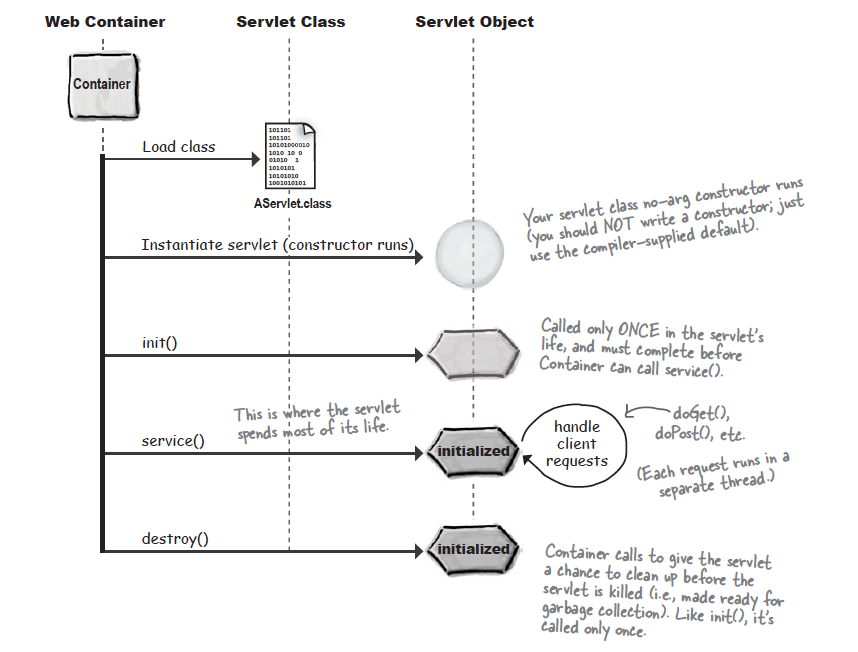


But there's more to a servlet life, we stepped into the middle of the servlet's life but that still leaves questions: when was the servlet class loaded? When did the servlet's constructor run? How long does the servlet object live? When should your servlet initialize resources? When should it clean up its resources?

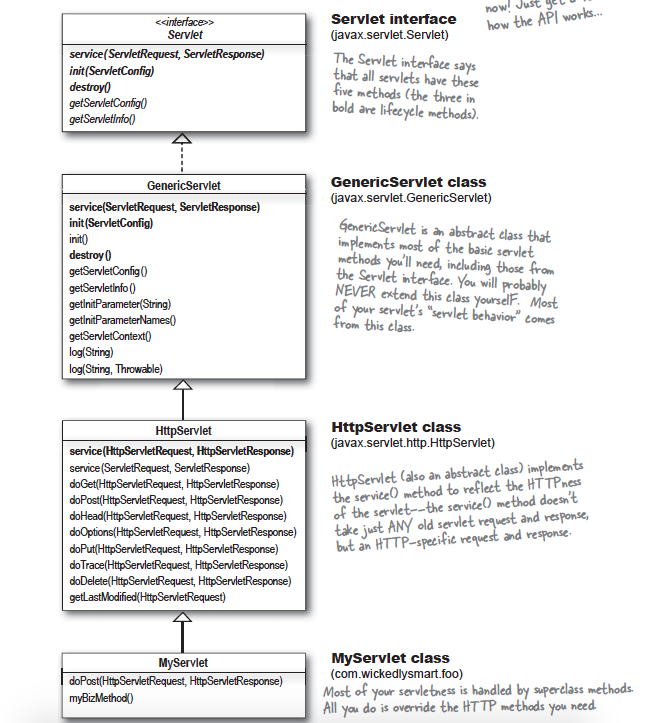
The servlet lifecycle is simple; there’s only one main state—*initialized*. If the servlet isn’t initialized, then it’s either *being initialized* (running its constructor or init()method), *being destroyed* (running its destroy() method), or it simply *does not exist.*



Detailed flow described below:



In the previous chapters, we have discussed about hierarchy of Servlet's Interfaces and Abstract classes, let's revise them with focus on life cycle methods.



**Let's bring some light on the three Big Lifecycle Moments:**

**init()**:

**When it is called?**

The Container calls init() on the servlet instance after the servlet instance is created but before the servlet can service any client requests and it is called only once.

**What it's for?**

Gives you a chance to initialize your servlet before handling any client requests.

**Do you Override it?**

Possibly yes!! 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():**

**When it is called?**

When client comes in, the Container starts a new thread or allocates a thread from the pool.

**What it's for?**

This method looks at the request, determines the HTTP method (GET, POST, etc.) and invokes the matching doGet(), doPost(), etc. on the servlet.

**Do you Override it?**

No very unlikely!! You should NOT override the service() method. Your job is to override the doGet() , doPost() methods and let the service() implementation from HTTPServlet worry about calling the right one.

**doGet(), doPost(), etc.:**

The service method invokes doGet() or doPost() based on the HTTP method(GET, POST, etc.) from the request. We are including only doGet() and doPost() here, because those two are probably the only ones you'll ever use.

**What it's for?**

This is where your code begins! This is the method that's responsible for whatever the heck your web app is supposed to doing.

**Do you override it?**

Always at least one of them, Whichever one(s) you override tells the Container what you support. If you don’t override doPost(), for example, then you’re telling the Container that this servlet does not support HTTP POST requests

**destroy():**

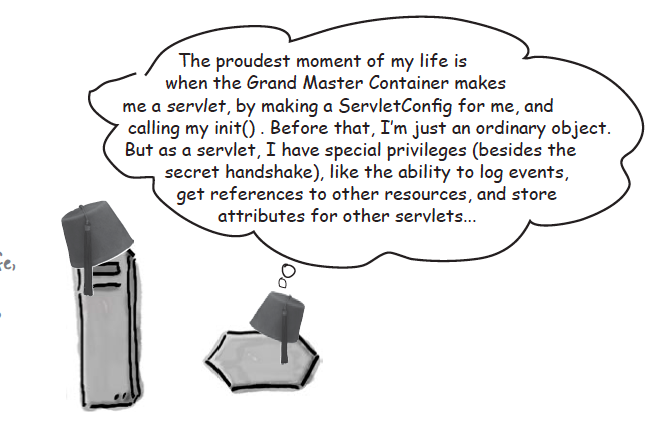
In java servlet, destroy() is not supposed to be called by the programmer. But, if it is invoked, it gets executed. The implicit question is, will the servlet get destroyed? No, it will not. destroy() method is not supposed to and will not destroy a java servlet. Don’t get confused by the name. It should have been better, if it was named onDestroy().

The meaning of destroy() in java servlet is, the content gets executed just before when the container decides to destroy the servlet. But if you invoke the destroy() method yourself, the content just gets executed and then the respective process continues. With respective to this question, the destroy() gets executed and then the servlet initialization gets completed.

**Servlet Initialization, When an object becomes a Servlet**

A Servlet moves from doesn't exist to initialized( which means ready to service client requests), beginning with a constructor. But constructor makes only an object, not a servlet. To be a servlet, the object needs to be granted servletness!!

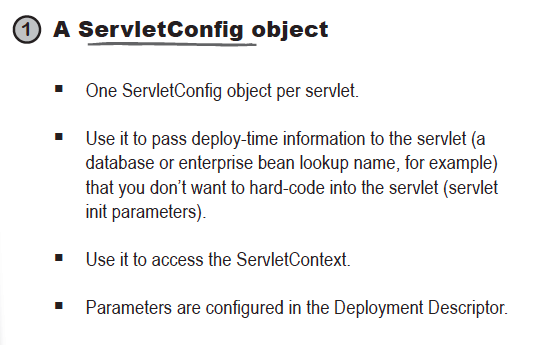
When an object becomes a servlet, it gets all the unique privileges that come with being a servlet, like the ability to use its ServletContext reference to get information from the Container.

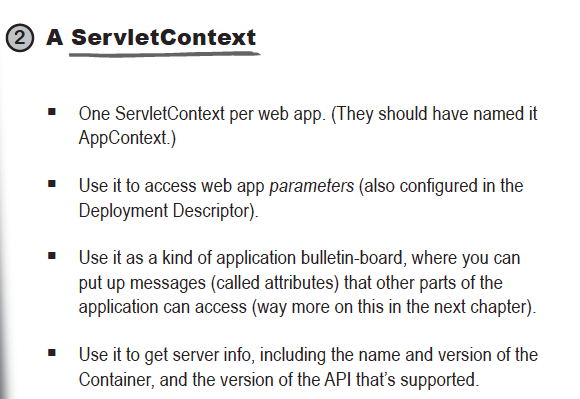
****

One question might be hitting you now, For initializing a servlet can we use constructor in place of init().

Answer: NO, we can not use constructor for initializing a servlet because for initialization we need an object of servletConfig using this object we get all the parameter which are defined in deployment descriptor for initializing a servlet and in servlet class we have only default constructor so if we want to pass a Configobject we don’t have parametrized constructor and apart from this servlet is loaded and initialized by container so its a job of container to call the method according to servlet specification they have lifecycle method so init() method is called firstly.More important Java doesn't allow interfaces to declare constructors.

**So, What does 'being a servlet' buy you?**

****

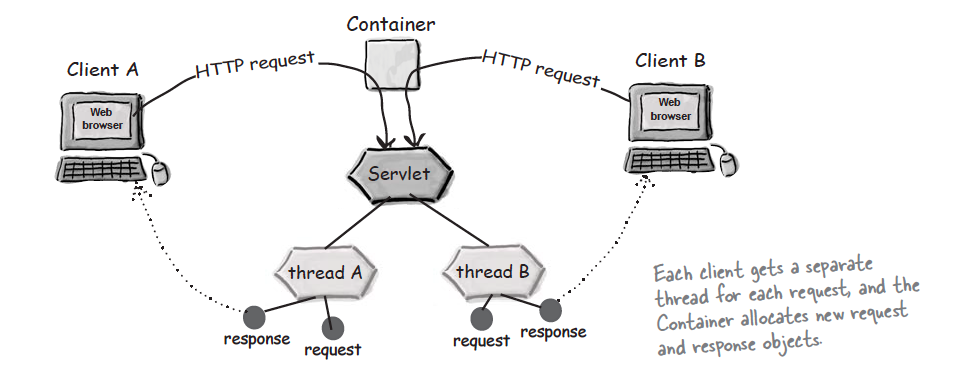
****

We will dig into this topic(ServletContext and ServletConfig) later, let's start with a new topic that's called Servlet and MultiThreading.

**Each request runs in a separate Thread!**

You might hear people say things like, “Each instance of the servlet...” but that’s just wrong. There aren’t multiple instances of any servlet class, except in one special case (called SingleThreadModel, which is inherently evil), but we’re not talking about that special case yet.

The Container runs multiple threads to process multiple requests to a single Servlet and every client request generates a new pair of request and response objects.



So we can say that, servlet's runs inside MULTI THREADING ENVIRONMENT, which brings us to one famous question that Is Servlet's are thread safe???

Before answering it we must know what do we mean by thread safe: When multiple threads/requests execute a single instance of a program and therefore shares memory, multiple threads could possibly be attempting to read and write to the same place in memory, here shared memory can be anything a simple int type instance, some other class instance or even static variables.

The answer to above mentioned question is NO, Servlet's are not thread safe!!.

Let's look at below mentioned sample code which will demonstrate the thread safety issue of Servlet's

package org.nishant.servlets;

import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class UnsafeServlet extends HttpServlet {

private static final long serialVersionUID = 1L;

private int counter = 0;

public UnsafeServlet() {

super();

}

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

System.out.println("Previous value of Counter for " + request

+ " is : " + counter);

counter++;

try {

Thread.sleep(3000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("New value of Counter for " + request + " is : "

+ counter);

}

}

In this code we have included a simple int type instance variable named counter initialized with zero value and in the doGet() method we are printing it's value, then incrementing it by one, then make a sleep of 3 seconds and again prints it's final value.

When we make two requests to this servlet we get the output something like this:

Previous value of Counter for org.apache.catalina.connector.RequestFacade@6e72d873 is : 0

Previous value of Counter for org.apache.catalina.connector.RequestFacade@566798eb is : 1

New value of Counter for org.apache.catalina.connector.RequestFacade@6e72d873 is : 2

New value of Counter for org.apache.catalina.connector.RequestFacade@566798eb is : 2

From the output we can say that here counter value becomes inconsistent, because two different thread tries to read/write same instance variable counter at the same time.

So How do we can make our servlet's thread safe.

We can get over come from this situation by multiple ways mentioned below with their merits and de merits:

**1. Use of SingleThreadModel Interface:**

If we want our servlet can handle only one request at a time, then this can be accomplished by implementing our Servlet with SingleThreadModel Interface. It is a marker interface, means have no methods.

public class UnsafeServlet extends HttpServlet implements ~~SingleThreadModel~~

This interface is currently deprecated since Servlet API 2.4 because it doesn't solves all the thread-safety issues such as static variable and session attributes can be accessed by multiple threads at the same time even if we have implemented the SingleThreadModel interface. So it is recommended to use other means to resolve these thread safety issues.

**2. Avoidance:**

I have always said that the best way to fix problems is to avoid them all together, in our case, this approach is best. When discussing thread safety we are interested only in the variables that we both read and write to pertain a particular WEB conversation. If the variable is for read-only use or it is application wide or if object itself does n't maintains any state( example Dao classes), then no harm results in sharing this memory space across all threads. For all other variables we need to ensure we have our own unique variable instance for each thread, we simple move the declaration of the variable from within the class to within the method using it. The difference is that, for each call to the method, a new variable is created therefore each thread has it's own variable.

**3. Partial synchronization:**

Thread synchronization is an important technique to know. But anytime you synchronize blocks of code, you introduce bottlenecks in your system. When you synchronize a code block, you tell the JVM that only one thread/request may be within this synchronized block of code at a given moment. If we run multithreaded/multiple requests application and a thread runs into a synchronized block being executed by another thread, the second thread must wait until the first thread exits that block.

Last Defence would be to synchronize the whole method, but that is not advisable at all.