**Servlet Attributes And Listeners**

In today's modern web application, many components work together to accomplish a goal. We have models, controllers and views(as per MVC). We have parameters and attributes. We have helper classes. But how do we tie the pieces together? How do we let components share information? How do we hide information? How do we make information thread-safe?

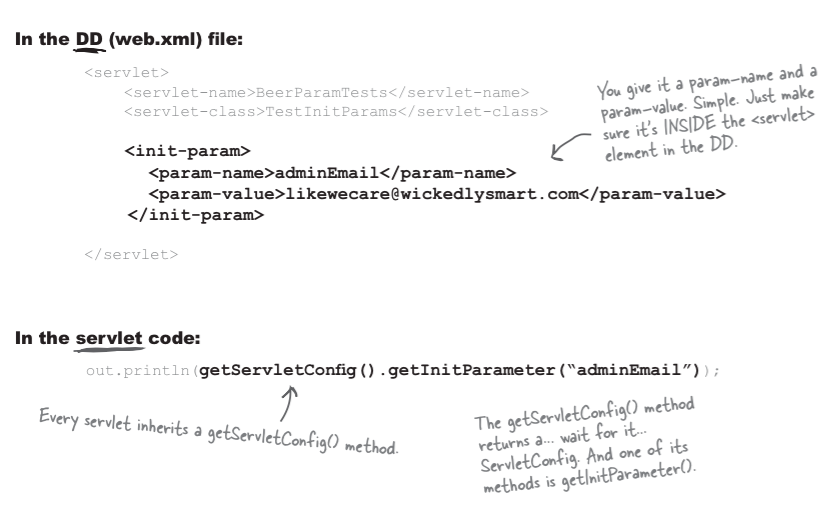
Let's start this with an example, suppose I have a dynamic web application and I want to put my email address. Now according to the requirement we don't want this hard coded email address in our servlet code :

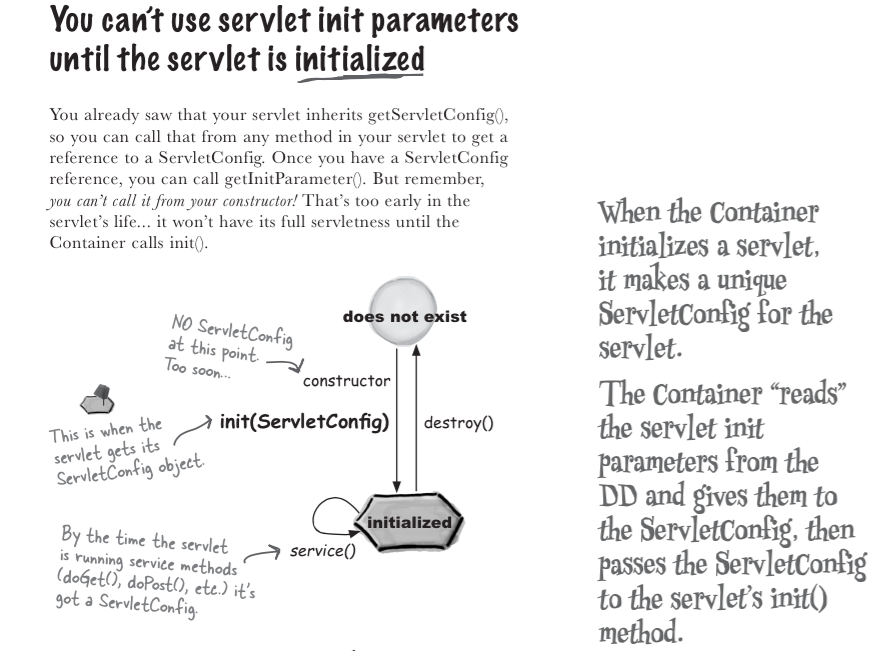
PrintWriter pw = response.getWriter();

pw.println(“[abc@email.com](mailto:abc@email.com)”);

Hard coding of the email is BAD!!! what happens when this email changes? We will have to re compile, we would rather put this email address in the Deployment Descriptor(web.xml), so that when we deploy our web app, our servlet can somehow read this email address from the DD. That way, we won't have to hard code this email in the servlet class and to change this email, we modify only the web.xml file, without having to touch the servlet source code.

We have already seen the request parameters that can come over in a doGet() or doPost(), but servlets can have initialization parameters as well.





When the container makes a servlet, it reads the DD and creates the name/value pair for the ServletConfig. The container never reads the init parameters again! Once the parameters are in the ServletConfig they won't be read again until/unless you redeploy the servlet.

But How can a JSP get Servlet init parameters??

A ServletConfig is for servlet configuration( it does n't say JSPConfig). So if we want other parts of our application to use same info we put in the servlet's init parameters in the DD, we need something more.

Example Option :

String emailAddress = getServletConfig().getInitParameter(“email”);

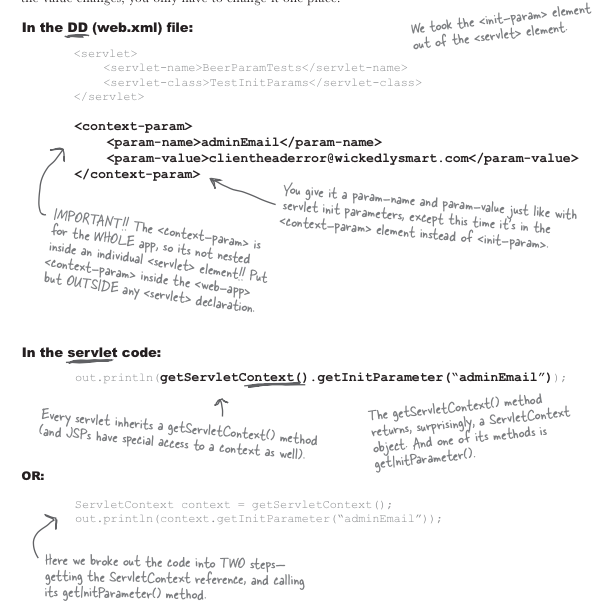
request.setAttribute(“email”,emailAddress);

We *could* do it this way. The request object lets you set *attributes* (think of them as a name/value pair where the value can be any object) that any other servlet or JSP that gets the request can use. That means any servlet or JSP to which the request is forwarded using a *RequestDispatcher*. We’ll look at RequestDispatcher in detail later, but for now all we care about is getting the data (in this case the email address) to the pieces of the web app that need it, rather than just one servlet.

But that doesn’t help us with the email address, because we might need to use it from all over the application! There *is* a way to have a servlet read the init parameters and then store them in a place other parts of the app could use, but then we’d have to know *which* servlet would always run first when the app is deployed, and any changes to the web app could break the whole thing. No, that won’t do either.

**Context Init Parameters :**

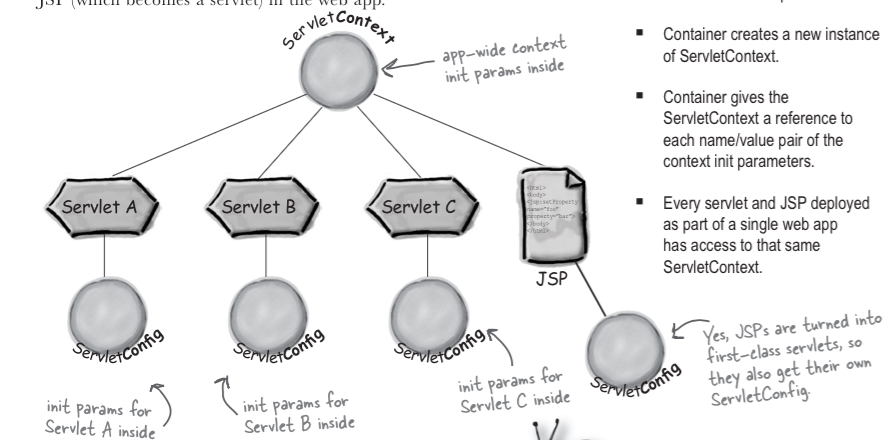
*Context* init parameters work just like *servlet* init parameters, except context parameters are available to the entire webapp, not just a single servlet. So that means any servlet and JSP in the app automatically has access to the context init parameters, so we don’t have to worry about configuring the DD for every servlet, and when the value changes, you only have to change it one place!

****

Remember the difference between servlet init parameters and context init parameters

|  |  |
| --- | --- |
| Context Init Parameters | Servlet Init Parameters |
| 1. Declared with in <web-app> element but not within a specific <servlet> element.  <web-app ...>  **<context-param>**  **<param-name>foo</param-name>**  **<param-value>bar</param-value>**  **</context-param>**  </web-app> | 1. Declared with in the <servlet> element for each specific servlet  <web-app ..>  <servlet>  <servlet-name>fooservlet</servlet-name>  <servlet-class>com.fooservlet</servlet-class>  **<init-param>**  **<param-name>foo</param-name>**  **<param-value>bar</param-value>**  **</init-param>**  </servlet>  </web-app> |
| 2. Servlet Code :  **getServletContext().getInitParameter(“foo”);** | 2. Servlet Code :  **getServletConfig().getInitParameter(“foo”);** |
| 3. Availability :  To all the servlets and JSPs that are part of the web  app. | 3. Availability :  To only the servlet for which the <init-param> was configured. (Although the servlet can choose to make it more widely available by storing it in an attribute.) |

To above all, we must remember the basic difference between ServletConfig and ServletContext, that ServletConfig is one per servlet and ServletContext is one per web app. 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.

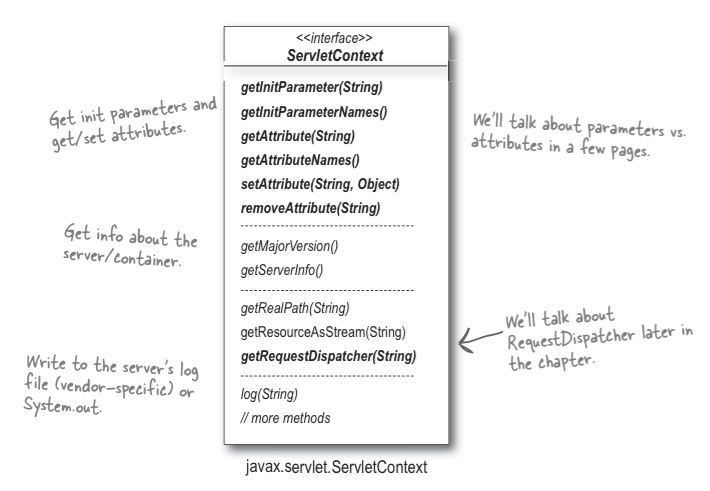


We have said that ServletContext is one per web app, but if your application is distributed across multiple servers ( probably in a clustered environment ), our web app really could have more than one ServletContext. A ServletContext is one per app, but only if the app is in a single JVM!. In a distributed enviroment, we will have one ServletContext per JVM. Now, chances are this won't create problems, but if we have a distributed web app, we better consider the consequences of having different contexts for each JVM.

Another important concept : Think of init parameters as deploy time constants!, we can get them at runtime, but we can't set them. There's no setInitParameter().

**ServletContext Concepts :**

A ServletContext is a JSP or servlet’s connection to both the Container and the other parts of the web app. Here are some of the ServletContext methods.



We can get a ServletContext in two ways, first one we have already discussed that is “getServletContext()” method and second is via ServletConfig object as it always holds a reference to the ServletContext for that servlet that is getServletConfig().getServletContext().

So, which one to use and when? In a servlet, the only time we would need to go through ServletConfig to get ServletContext is if we are in a Servlet class that doesn't extend HttpServlet or GenericServlet. But the chance of anyone using a non HTTP servlet is almost zero or even if we can check the implementation of getServletcontext() method in GenericServlet abstract class, that method itself uses ServletConfig object to return ServletContext reference.

GenericServlet.java :

public ServletContext getServletContext(){

return getServletConfig().getServletContext();

}

As of now, we have seen that we can get init parameters at Application level(ServletContext) and at Servlet level(ServletConfig), but these init parameters are of only String type data, what if we want an application init parameter that is a database Datasource?

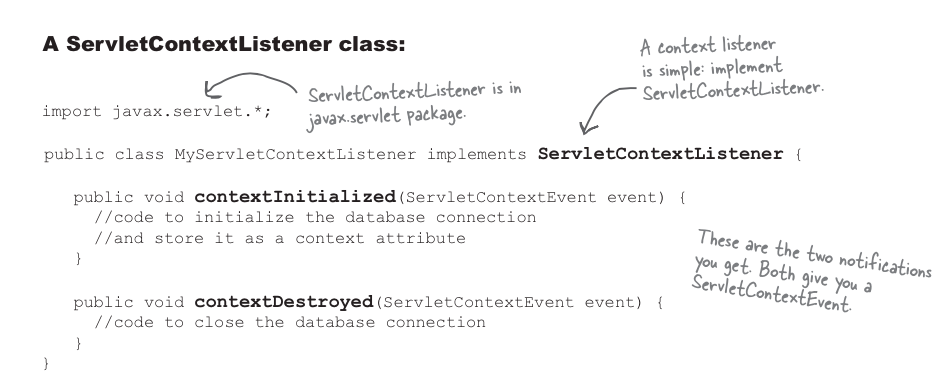
One certain solution is to store tha database related information in the form of init parameters(from ServletContext) and initialize tha datasource or connection object in the servlet class itself. But we can't really put that code in a servlet, because which servlet would we choose to be The One To Lookup The Datasource And Store It In An Attribute? Do we really want to try to guarantee that one servlet in particular will always run first?

For this scenario we want something like a “main” method for our whole web application. So that some code will always runs before any servlet or JSPs or at the time of application startup.

What we really want a kind of application initializer listener, which can get the context init parameters and run some code before the rest of the application can service a client.

We want a seperate class not a servlet or JSP, that can listen for the two key events in a ServletContext's life that is initialization and destruction. We want ServletContextListener, in this scenario we need to create a seperate class which implement this ServletContextListener interface.





Ok, we have a listener class. Now what should we do? Where should we put the class? Who instantiates it? How should I register for events? How does the listener set the attribute in the right ServletContext?

In order to answer this questions, we will proceed with an example, in this example we will create a Database connection object by reading context parameters in the contextInitialized method and store that connection object as an attribute in the ServletContext, because of this now we won't be dependent on any particular servlet or JSP, any servlet or any JSP can now acess this connection object from ServletContext, for all this to happen we need to register this listener in deployement descriptor. Let's start this steps one by one:

**Step 1: Create a DaoConnection class which holds the database connection object and also initialize it.**

public class DaoConnection {

private final Connection databaseConnection;

public DaoConnection(String connectionUrl, String username,String password, String jdbcDriver) throws ClassNotFoundException,SQLException {

this.databaseConnection = createConnection(connectionUrl, username,

password, jdbcDriver);

}

private Connection createConnection(String connectionUrl, String username,

String password, String jdbcDriver) throws ClassNotFoundException,

SQLException {

Connection databaseConnection = null;

Class.forName(jdbcDriver);

System.out.println("JDBC Driver " + jdbcDriver + " sucessfully loaded");

databaseConnection = DriverManager.getConnection(connectionUrl,

username, password);

System.out.println("Sucessfully created database connection");

return databaseConnection;

}

public Connection getDatabaseConnection() {

return databaseConnection;

}

}

**Step 2: Create a MysqlInitializerListener which creates database connection on context start up and stores the database connection object into ServletContext and destroys the database connection on context destruction, here database is mysql.**

public class MysqlInitializerListener implements ServletContextListener {

private static final String mysqlJdbcDriver = "com.mysql.jdbc.Driver";

@Override

public void contextInitialized(ServletContextEvent event) {

ServletContext context = event.getServletContext();

String databaseName = context.getInitParameter("database\_name");

String username = context.getInitParameter("database\_user\_name");

String password = context.getInitParameter("database\_passwd");

String connectionUrl = createCoonectionUrl(databaseName);

DaoConnection connection = null;

try {

connection = new DaoConnection(connectionUrl, username, password,

mysqlJdbcDriver);

} catch (ClassNotFoundException e) {

e.printStackTrace();

} catch (SQLException e) {

e.printStackTrace();

}

context.setAttribute("databaseConnection", connection);

}

@Override

public void contextDestroyed(ServletContextEvent event) {

ServletContext context = event.getServletContext();

DaoConnection connection = (DaoConnection) context

.getAttribute("databaseConnection");

try {

connection.getDatabaseConnection().close();

} catch (SQLException e) {

e.printStackTrace();

}

context.removeAttribute("databaseConnection");

System.out.println("Sucessfully closed database connection");

}

private String createCoonectionUrl(String databaseName) {

StringBuilder connectionUrl = new StringBuilder(

"jdbc:mysql://localhost:3306/");

connectionUrl.append(databaseName);

return connectionUrl.toString();

}

}

**Step 3: Register MysqlInitializerListener in the deployement descriptor, in order to make our listener come into action.**

<listener>

<listener-class>org.nishant.MysqlInitializerListener</listener-class>

</listener>

**NOTE : We read somewhere that servlet attributes had to be Serializable...**

There are several different attribute types, and whether the attribute should be Serializable only matters with Session attributes. And the scenario in which it matters is *only* if the application is distributed across more than one JVM. We’ll talk all about that in the Sessions chapter.

There’s no technical *need* to have any attributes (including Session attributes) be Serializable, although you might consider making all of your attributes Serializable by default, unless you have a really good reason NOT to.

Think about it—are you really certain that nobody will ever want to use objects of that type as arguments or return values as part of a remote method call? Can you really guarantee that anyone who uses this class (Dog, in this case) will never run in a distributed environment?

So, although you aren’t *required* to make any attributes Serializable, you probably *should* if you can.

**NOTE : How are we telling the Container that this is a listener for ServletContext events?**

There doesn’t seem to be an XML element for <listener-type> or anything that says what type of events this listener is for No. There’s no naming convention The Container figures it out simply by inspecting the class and noticing the listener interface (or inter- faces; a listener can implement more than one listener interface). Does that mean there are other types of listen- ers in the servlet API? Yes, there are several other types of listeners that we’ll continue to talk about all of them in the next chapter.