The Deployment Descriptor: web.xml

Java web applications use a deployment descriptor file to determine how URLs map to servlets, which URLs require authentication, and other information. This file is named **`web.xml`**, and resides in the app's WAR under the **`WEB-INF/`** directory. **`web.xml`** is part of the servlet standard for web applications.

For more information about the **`web.xml`** standard, see the Metawerx web.xml reference wiki (http://wiki.metawerx.net/wiki/Web.xml) and the Servlet specification (http://jcp.org/aboutJava/communityprocess/final/jsr154/index.html).

Deployment descriptors

A web application's deployment descriptor describes the classes, resources and configuration of the application and how the web server uses them to serve web requests. When the web server receives a request for the application, it uses the deployment descriptor to map the URL of the request to the code that ought to handle the request.

The deployment descriptor is a file named **`web.xml`**. It resides in the app's WAR under the **`WEB-INF/`** directory. The file is an XML file whose root element is **`<web-app>`**.

Here is a simple **`web.xml`** example that maps all URL paths (**`/*`**) to the servlet class **`mysite.server.ComingSoonServlet`**:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="2.5">
    <servlet>
        <servlet-name>comingsoon</servlet-name>
        <servlet-class>mysite.server.ComingSoonServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>comingsoon</servlet-name>
```

```
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Note: If you factor your application into services, each service has its own configuration parameters.

Servlets and URL paths

`web.xml` defines mappings between URL paths and the servlets that handle requests with those paths. The web server uses this configuration to identify the servlet to handle a given request and call the class method that corresponds to the request method. For example: the `doGet()` method for HTTP `GET` requests.

To map a URL to a servlet, you declare the servlet with the `<servlet>` element, then define a mapping from a URL path to a servlet declaration with the `<servlet-mapping>` element.

The `<servlet>` element declares the servlet, including a name used to refer to the servlet by other elements in the file, the class to use for the servlet, and initialization parameters. You can declare multiple servlets using the same class with different initialization parameters. The name for each servlet must be unique across the deployment descriptor.

```
    <servlet>
        <servlet-name>redteam</servlet-name>
        <servlet-class>mysite.server.TeamServlet</servlet-class>
        <init-param>
            <param-name>teamColor</param-name>
            <param-value>red</param-value>
        </init-param>
        <init-param>
            <param-name>bgColor</param-name>
            <param-value>#CC0000</param-value>
        </init-param>
    </servlet>

    <servlet>
        <servlet-name>blueteam</servlet-name>
        <servlet-class>mysite.server.TeamServlet</servlet-class>
        <init-param>
            <param-name>teamColor</param-name>
            <param-value>blue</param-value>
```

```
        </init-param>
        <init-param>
            <param-name>bgColor</param-name>
            <param-value>#0000CC</param-value>
        </init-param>
    </servlet>
```

The **`<servlet-mapping>`** element specifies a URL pattern and the name of a declared servlet to use for requests whose URL matches the pattern. The URL pattern can use an asterisk (`*`) at the beginning or end of the pattern to indicate zero or more of any character. The standard does not support wildcards in the middle of a string, and does not allow multiple wildcards in one pattern. The pattern matches the full path of the URL, starting with and including the forward slash (`/`) following the domain name. The URL path cannot start with a period (`.`).

```
    <servlet-mapping>
        <servlet-name>redteam</servlet-name>
        <url-pattern>/red/*</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>blueteam</servlet-name>
        <url-pattern>/blue/*</url-pattern>
    </servlet-mapping>
```

With this example, a request for the URL **`http://www.example.com/blue/teamProfile`** is handled by the **`TeamServlet`** class, with the **`teamColor`** parameter equal to **`blue`** and the **`bgColor`** parameter equal to **`#0000CC`**. The servlet can get the portion of the URL path matched by the wildcard using the ServletRequest object's **`getPathInfo()`** method.

Note: Static files that are served verbatim to users, such as images, CSS or JavaScript, are handled separately from paths mentioned in the deployment descriptor. A request for a URL path that matches a path to a file in the WAR that's considered a static file will serve the file, regardless of servlet and filter mappings in the deployment descriptor. You can exclude files from those treated as static files using the appengine-web.xml (https://cloud.google.com/appengine/docs/standard/java/config/appref) file.

The servlet can access its initialization parameters by getting its servlet configuration using its own **`getServletConfig()`** method, then calling the **`getInitParameter()`** method on the configuration object using the name of the parameter as an argument.

```
String teamColor = getServletConfig().getInitParameter("teamColor");
```

JSPs

An app can use JavaServer Pages (JSPs) to implement web pages. JSPs are servlets defined using static content, such as HTML, mixed with Java code.

App Engine supports automatic compilation and URL mapping for JSPs. A JSP file in the application's WAR (outside of **`WEB-INF/`**) whose filename ends in **`.jsp`** is compiled into a servlet class automatically, and mapped to the URL path equivalent to the path to the JSP file from the WAR root. For example, if an app has a JSP file named **`start.jsp`** in a subdirectory named **`register/`** in its WAR, App Engine compiles it and maps it to the URL path **`/register/start.jsp`**.

If you want more control over how the JSP is mapped to a URL, you can specify the mapping explicitly by declaring it with a **`<servlet>`** element in the deployment descriptor. Instead of a **`<servlet-class>`** element, you specify a **`<jsp-file>`** element with the path to the JSP file from the WAR root. The **`<servlet>`** element for the JSP can contain initialization parameters.

```
<servlet>
    <servlet-name>register</servlet-name>
    <jsp-file>/register/start.jsp</jsp-file>
</servlet>

<servlet-mapping>
    <servlet-name>register</servlet-name>
    <url-pattern>/register/*</url-pattern>
</servlet-mapping>
```

Note: The **`<jsp-file>`** must start with a forward slash (**`/`**) if the JSP is in the application's root directory.

You can install JSP tag libraries with the **`<taglib>`** element. A tag library has a path to the JSP Tag Library Descriptor (TLD) file (**`<taglib-location>`**) and a URI that JSPs use to select the library for loading (**`<taglib-uri>`**). Note that App Engine provides the JavaServer Pages Standard Tag Library (http://java.sun.com/products/jsp/jstl/) (JSTL), and you do not need to install it.

```
<taglib>
    <taglib-uri>/escape</taglib-uri>
```

```
    <taglib-location>/WEB-INF/escape-tags.tld</taglib-location>
</taglib>
```

Security and authentication

An App Engine application can use Google Accounts for user authentication. The app can use the Google Accounts API (https://cloud.google.com/appengine/docs/standard/java/users) to detect whether the user is signed in, get the currently signed-in user's email address, and generate sign-in and sign-out URLs. An app can also specify access restrictions for URL paths based on Google Accounts, using the deployment descriptor.

The **`<security-constraint>`** element defines a security constraint for URLs that match a pattern. If a user accesses a URL whose path has a security constraint and the user is not signed in, App Engine redirects the user to the Google Accounts sign-in page. Google Accounts redirects the user back to the application URL after successfully signing in or registering a new account. The app does not need to do anything else to ensure that only signed-in users can access the URL.

A security constraint includes an authorization constraint that specifies which Google Accounts users can access the path. If the authorization constraint specifies a user role of `*`, then any users signed in with a Google Account can access the URL. If the constraint specifies a user role of **`admin`**, then only registered developers of the application can access the URL. The **`admin`** role makes it easy to build administrator-only sections of your site.

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>profile</web-resource-name>
        <url-pattern>/profile/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>*</role-name>
    </auth-constraint>
</security-constraint>

<security-constraint>
    <web-resource-collection>
        <web-resource-name>admin</web-resource-name>
        <url-pattern>/admin/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
```

```
        <role-name>admin</role-name>
    </auth-constraint>
</security-constraint>
```

Note: G Suite domain administrators and App Engine domain administrators are not included in the **`admin`** role in this context. Only the application developers, such as those in **`Viewer`**, **`Owner`**, or **`Developer`** roles, can access these portions of the application.

App Engine does not support custom security roles (**`<security-role>`**) or alternate authentication mechanisms (**`<login-config>`**) in the deployment descriptor.

Security constraints apply to static files as well as servlets.

Secure URLs

Google App Engine supports secure connections via HTTPS for URLs using the **`*.appspot.com`** domain. When a request accesses a URL using HTTPS, and that URL is configured to use HTTPS in the **`web.xml`** file, both the request data and the response data are encrypted by the sender before they are transmitted, and decrypted by the recipient after they are received. Secure connections are useful for protecting customer data, such as contact information, passwords, and private messages.

Note: To use G Suite domains with HTTPS, you must first activate and configure SSL for App Engine with your domain (https://cloud.google.com/appengine/docs/standard/java/securing-custom-domains-with-ssl). Otherwise, users attempting to view pages via HTTPS on your domain will likely see timeouts, errors, or warnings.

To declare that HTTPS should be used for a URL, you set up a security constraint in the deployment descriptor (as described in Security and authentication (#Security_and_Authentication)) with a **`<user-data-constraint>`** whose **`<transport-guarantee>`** is **`CONFIDENTIAL`**. For example:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>profile</web-resource-name>
        <url-pattern>/profile/*</url-pattern>
    </web-resource-collection>
```

```
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

Requests using HTTP (non-secure) for URLs whose transport guarantee is **`CONFIDENTIAL`** are automatically redirected to the same URL using HTTPS.

Any URL can use the **`CONFIDENTIAL`** transport guarantee, including JSPs and static files.

The development web server does not support HTTPS connections. It ignores the transport guarantee, so paths intended for use with HTTPS can be tested using regular HTTP connections to the development web server.

When you test your app's HTTPS handlers using the versioned appspot.com URL, such as **`https://1.latest.<i>your_app_id</i>.appspot.com/`**, your browser warns you that the HTTPS certificate was not signed for that specific domain path. If you accept the certificate for that domain, pages will load successfully. Users will not see the certificate warning when accessing **`https://<i>your_app_id</i>.appspot.com/`**.

You can also use an alternate form of the versioned appspot.com URL designed to avoid this problem by replacing the periods separating the subdomain components with the string "**`-dot-`**". For instance, the previous example could be accessed without a certificate warning at **`https://1-dot-latest-dot-`*`your_app_id`*`.appspot.com/`**.

Google Accounts sign-in and sign-out are always performed using a secure connection and is unrelated to how the application's URLs are configured.

As mentioned above, security constraints apply to static files as well as servlets. This includes the transport guarantee.

Note: Google recommends (http://googleonlinesecurity.blogspot.com/2014/08/https-as-ranking-signal_6.html) using the HTTPS protocol to send requests to your app. Google does not issue SSL certificates for double-wildcard domains hosted at **`appspot.com`**. Therefore with HTTPS you must use the string "-dot-" instead of "." to separate subdomains, as shown in the examples below. You can use a simple "." with your own custom domain or with HTTP addresses.

The welcome file list

When the URLs for your site represent paths to static files or JSPs in your WAR, it is often a good idea for paths to directories to do something useful as well. A user visiting the URL path `/help/accounts/password.jsp` for information on account passwords might try to visit `/help/accounts/` to find a page introducing the account system documentation. The deployment descriptor can specify a list of filenames that the server should try when the user accesses a path that represents a WAR subdirectory that is not already explicitly mapped to a servlet. The servlet standard calls this the "welcome file list."

For example, if the user accesses the URL path `/help/accounts/`, the following `<welcome-file-list>` element in the deployment descriptor tells the server to check for `help/accounts/index.jsp` and `help/accounts/index.html` before reporting that the URL does not exist:

```
<welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>index.html</welcome-file>
</welcome-file-list>
```

Filters

A *filter* is a class that acts on a request like a servlet, but can allow the handling of the request to continue with other filters or servlets. A filter may perform an auxiliary task, such as logging, performing specialized authentication checks, or annotating the request or response objects before calling the servlet. Filters allow you to compose request processing tasks from the deployment descriptor.

A filter class implements the `javax.servlet.Filter` interface, including the `doFilter()` method. Here is an simple filter implementation that logs a message, and passes control down the chain, which may include other filters or a servlet, as described by the deployment descriptor:

```
package mysite.server;

import java.io.IOException;
import java.util.logging.Logger;
import javax.servlet.Filter;
```

```
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class LogFilterImpl implements Filter {

    private FilterConfig filterConfig;
    private static final Logger log = Logger.getLogger(LogFilterImpl.class.getNa

    public void doFilter(ServletRequest request, ServletResponse response, Filte
        throws IOException, ServletException {
        log.warning("Log filter processed a " + getFilterConfig().getInitParamet
            + " request");

        filterChain.doFilter(request, response);
    }

    public FilterConfig getFilterConfig() {
        return filterConfig;
    }

    public void init(FilterConfig filterConfig) {
        this.filterConfig = filterConfig;
    }

    public void destroy() {}

}
```

Similar to servlets, you configure a filter in the deployment descriptor by declaring the filter with the **`<filter>`** element, then mapping it to a URL pattern with the **`<filter-mapping>`** element. You can also map filters directly to other servlets.

The **`<filter>`** element contains a **`<filter-name>`**, **`<filter-class>`**, and optional **`<init-param>`** elements.

```
    <filter>
        <filter-name>logSpecial</filter-name>
        <filter-class>mysite.server.LogFilterImpl</filter-class>
        <init-param>
            <param-name>logType</param-name>
```

```
        <param-value>special</param-value>
    </init-param>
</filter>
```

The **`<filter-mapping>`** element contains a **`<filter-name>`** that matches the name of a declared filter, and either a **`<url-pattern>`** element for applying the filter to URLs, or a **`<servlet-name>`** element that matches the name of a declared servlet for applying the filter whenever the servlet is called.

```
<!-- Log for all URLs ending in ".special" -->
<filter-mapping>
    <filter-name>logSpecial</filter-name>
    <url-pattern>*.special</url-pattern>
</filter-mapping>

<!-- Log for all URLs that use the "comingsoon" servlet -->
<filter-mapping>
    <filter-name>logSpecial</filter-name>
    <servlet-name>comingsoon</servlet-name>
</filter-mapping>
```

Note: Filters are not invoked on static assets, even if the path matches a **`filter-mapping`** pattern. Static files are served directly to the browser.

Error Handlers

You can customize what the server sends to the user when an error occurs, using the deployment descriptor. The server can display an alternate page location when it's about to send a particular HTTP status code, or when a servlet raises a particular Java exception.

The **`<error-page>`** element contains either an **`<error-code>`** element with an HTTP error code value (such as **`500`**), or an **`<exception-type>`** element with the class name of the expected exception (such as **`java.io.IOException`**). It also contains a **`<location>`** element containing the URL path of the resource to show when the error occurs.

```
<error-page>
    <error-code>500</error-code>
    <location>/errors/servererror.jsp</location>
</error-page>
```

Note: At present, you cannot configure custom error handlers for some error conditions. Specifically, you cannot customize the HTTP `404` response page when no servlet mapping is defined for a URL, the `403` quota error page, or the `500` server error page that appears after an App Engine internal error.

Unsupported web.xml features

The following web.xml features are not supported by App Engine:

- App Engine supports the `<load-on-startup>` element for servlet declarations. However, the load actually occurs during the first request handled by the web server instance, not prior to it.
- Some deployment descriptor elements can take a human readable display name, description and icon for use in IDEs. App Engine doesn't use these, and ignores them.
- App Engine doesn't support JNDI environment variables (`<env-entry>`).
- App Engine doesn't support EJB resources (`<resource-ref>`).
- Notification of the destruction of servlets, servlet context, or filters is not supported.
- The `<distributable>` element is ignored.
- Servlet scheduling with `<run-at>` is not supported.

Última actualización: Diciembre 13, 2017.