

Easy Java Websites (EJW)

Configuration Free

An Extremely Easy To Use
Web Application Framework
for Java

Copyright © 2005 - Present, EasierJava All Rights Reserved.

This Easier Java Persistence manual is a copyrighted work and is not transferable. If you received this manual somewhere other than EasierJava.com, please report it.

EasierJava, EJP and EJW are trademarks or registered trademarks of EasierJava, Inc. in the U.S. and other countries. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All other marks are trademarks or registered trademarks of their respective holders in the U.S. and other countries.

EASY JAVA WEBSITES (EJW) QUICK START.....	3
CLASS MAPPING.....	4
METHOD MAPPING.....	4
REST.....	4
CONFIGURATION MODE (OPTIONAL).....	5
REQUEST HANDLER.....	5
SETTING UP AN EJW PROJECT.....	7
INSTALLING A WEB SERVER.....	7
A TYPICAL EJW PROJECT LAYOUT.....	7
PAGE AUTHORIZING.....	8
APPLICATION MESSAGING (MESSAGES TO THE USER).....	9
EJW MODEL/BUSINESS INTERFACING.....	9
SERVERINTERFACE.....	10
CONFIGURATION FREE REQUESTS.....	10
CONFIGURING THE EJW SERVLET	11
SERVLET 3.X ANNOTATIONS.....	11
SERVLET DEPLOYMENT DESCRIPTOR (WEB.XML)	11
<i>Servlet (required)</i>	12
<i>initDestroyHandler (optional)</i>	12
<i>classPrefix (optional)</i>	13
<i>urlRewriting (optional)</i>	13
<i>handleMimeTypes (optional)</i>	14
<i>Servlet Mapping (required)</i>	14
SPECIAL TOPICS	15
LOGGING.....	15
REST (RESTFUL WEB SERVICE) REQUESTS.....	15
AJAX REQUESTS.....	17
DEFAULT REQUEST.....	17
ERROR HANDLING.....	17
SECURITY/USER AUTHENTICATION.....	18
SECURE CONNECTIONS.....	18

Easy Java Websites (EJW) Quick Start

EJW is a Java servlet-based model-view-controller web application development framework specifically designed to satisfy the need for extremely simple web development. In fact, EJW is so simple it can actually be learned in few minutes.

EJW is as easy as:

```
public class HelloWorld extends RequestHandler
{
    public String hello()
    {
        getServerInterface().addViewObject("helloWorld", "Hello World");

        return "/WEB-INF/helloWorld.jsp";
    }
}
```

Just plain HTML and your favorite template markup:

```
<div align="center">
    <h3>Hello, This Is A Simple "Hello World" Example.</h3>
    <h2>${helloWorld}</h2>
</div>
```

Calling the above is simple:

```
http://host/context/helloWorld/hello
```

The following servlet configuration is all that is needed (outside our control, it's a servlet thing):

```
<servlet>
    <servlet-name>ejwRequestServlet</servlet-name>
    <servlet-class>ejw.RequestServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>ejwRequestServlet</servlet-name>
    <url-pattern>/helloWorld/hello</url-pattern>
</servlet-mapping>
```

That's it! No other XML, annotation, or configuration is needed!

EJW provides support for all of the features you expect, but does not need additional tag libraries or extensive APIs to implement them, what you already know is all you will need. With EJW, you can use any model/business layer technologies, any database-access technologies, any web-authoring technologies, and plain old HTML and HTML forms.

Class Mapping

With EJW you have a configuration free mode where you simply drive the web application with URIs. EJW automatically finds the class instance required to satisfy a request.

For example, with the following URL:

`http://www.website.com/application/HelloWorld`

EJW would, by default, find a class named `[webapp.][application.]HelloWorld.class`.

EJW will automatically load the class and, by default, call the method “`validateRequest`” if it exists. EJW then calls the method “`request`” which must exist. If the method is not found an HTTP not found error is returned.

The URI can be anything as long as all or part of it identifies a locatable class. All of EJW’s functionality is available in code and the methods only need to return a forwarding URI, which can be another request handler or a final destination such as a JSP or Velocity page. The methods can also return `NO_FORWARDING` if there is no need for forwarding (no data is returned to the client).

Method Mapping

EJW also provides for mapping web requests to methods within a class.

As another example, with the following URL:

`http://www.website.com/application/manageUsers/newUser`

EJW would, by default, find a class named `[webapp.][application.]ManageUsers.class`. EJW loads an instance of the class and optionally calls the method “`validateNewUser`” if it exists. EJW then calls the method “`newUser`” which must exist (or an HTTP not found error is returned).

REST

EJW provides another configuration free mode that provides server side support for REST (RESTful web services). REST is based on HTTP concepts, and is simply the use of the 6 most

common HTTP request methods (POST, GET, PUT, DELETE, HEAD and OPTIONS). Which, as defined in both HTTP and REST, correspond to CRUD type operations:

- POST (create - also insert, update, delete, append)
- GET (read - also select, get)
- PUT (update - also insert, update, replace)
- DELETE (delete - also remove, cut)
- HEAD (same as get without data/body - just headers)
- OPTIONS (what HTTP methods are supported)

Regular EJW request handling can also be RESTful, as the server interface will provide any information required to figure out what type of request is made, as well as, the path arguments. Requests are also RESTful if they have query strings. By definition, REST does not allow for session tracking or cookies (they are still available with EJW), and authentication (also available with EJW) is handled as defined for HTTP (via request headers).

Configuration Mode (Optional)

EJW also provides a completely optional, but simple, XML configuration mode which consists of:

```
<request id="" match="" matchRegExp="" secureConnection=""
    handlerClass="" handlerMethod="">
  <security role="" />
  <parameter name="" value="" />
  <forward id="" key="" protocol="" url="" rewrite=""/>
  <redirect id="" key="" protocol="" url="" rewrite=""/>
  <validation id="" parameterName="" type="" arguments="" errorMessage=""/>
</request>

<url id="" protocol="" url="" rewrite=""/>
<message id="" value="" />
```

This mode provides the ability for class and method reuse, passing in static parameters, regular expression matching of URIs, soft coded URIs, and soft coded messages. EJW configuration even allows overriding configuration free and other configurations. The configuration mode also provides for XML defined security, forwarding, redirects and validation.

See the configuration manual for more details.

Request Handler

In both configuration free and configuration based cases, the request handler is a class that extends RequestHandler (RestfulHandler for REST), for example:

```

public class MyRequestHandler extends RequestHandler
{
    // default handler
    public String processRequest(ServerInterface serverInt) { ... }

    // default handler's optional validator
    public String validateRequest(ServerInterface serverInt) { ... }

    public String processNewUser(ServerInterface serverInt) { ... }
    public String validateNewUser(ServerInterface serverInt) { ... }
}

```

For RESTful web services:

```

public class MyRestfulHandler extends RestfulHandler
{
    // Handles HTTP GET request.
    public String get(ServerInterface serverInterface, String[] args) { ... }

    // Handles HTTP HEAD request.
    public String head(ServerInterface serverInterface, String[] args) { ... }

    // Handles HTTP PUT request.
    public String put(ServerInterface serverInterface, String[] args) { ... }

    // Handles HTTP POST request.
    public String post(ServerInterface serverInterface, String[] args) { ... }

    // Handles HTTP DELETE request.
    public String delete(ServerInterface serverInterface, String[] args) { ... }

    // Handles HTTP OPTIONS request.
    public String options(ServerInterface serverInterface, String[] args) { ... }
}

```

All parameters are optional and you can also use `HttpServletRequest` and/or `HttpServletResponse` as parameters.

In a configuration free mode, the only need is for the class to be in the package “webapp” (default), which is optionally definable in the `web.xml` file via the `init` parameter “classPrefix”:

```

<init-param>
  <param-name>classPrefix</param-name>
  <param-value>webapp</param-value>
</init-param>

```

And of course, the class/JAR files need to be locatable (usually in the WEB-INF/classes and/or WEB-INF/lib directory).

That's it, that's all there is to EJWs configuration free automatic URI mapping!

Setting Up an EJW Project

Installing an EJW project is simply a matter of copying the "EJW_XX.war" file of choice to your web server's auto-deploy directory "webapps". Your web server will automatically unpack and install the web application. You can also simply unzip the archive (WARs are zip files). It's also possible to rename the ".war" file to ".zip" and unzip the archive.

Once installed simply start development of your webapp.

Installing a Web Server

If you need a web server, simply download and install Apache's Tomcat, Coucho's Resin or Jetty. They are free and easy to install and run. There are many J2EE web servers that you can also use.

Resin is extremely easy on Windows' platforms. Simply unzip Resin in the directory of your choice and run 'httpd.exe'. Copy the 'EJW_XX.war' file to the webapps directory and you're done.

A Typical EJW Project Layout

EJW is easy to set up and use with your projects, and follows the typical setup for Java web technology applications:

```
/
/images/
/WEB-INF/
/WEB-INF/classes/
/WEB-INF/content/
/WEB-INF/include/
/WEB-INF/lib/
/WEB-INF/web.xml
/WEB-INF/webapp.xml
/WEB-INF/mainpage.jsp
index.jsp
```

where ‘/’ is any directory you like. This is our typical setup, but you are free to use any workable setup you like. In a configuration free mode the webapp.xml file is not needed. But even in a configuration free mode it’s still quite useful for defining non-hardcoded URIs and messages.

The JAR files in the “WEB-INF/lib” directory can also be located in your web server’s “common” or “shared” lib directory (container dependent), where they will be available to all applications. However, if you’re working with multiple versions of a jar, it may be better to keep them in the project’s lib directory to keep them isolated from other applications.

Working with EJW is no different than working with any other Java web technology, only much simpler.

Page Authoring

Page authoring can be handled with any view technology you like. JSP/JSTL is a good choice, and so is Apache’s Velocity.

All attributes set with `getServerInterface().addViewObject()` and `getServerInterface().setAttribute()`, which simply calls the servlet request’s `setAttribute()`, are available to the page author via JSP or Velocity expressions:

`${attributeName}`

Servlets provide several implicit objects (such as `${param}`) that are always available to the page author. EJW also provides several implicit objects, summarized by the following table:

<code>\${ejwUrl.id}</code>	provides URLs as defined in the url, forward, and redirect definitions in your configuration file
<code>\${ejwMessage.id}</code>	Access messages defined in the XML configuration file
<code>\${ejwContextPath}</code>	The servlet context
<code>\${ejwServletPath}</code>	The servlet path (exact or path oriented paths only)
<code>\${ejwPathInfo}</code>	The request extra path information (follows servlet path)
<code>\${ejwQueryString}</code>	The request query string
<code>\${ ejwUri}</code>	The request URI
<code>\${ ejwUrl}</code>	The request URL
<code>\${ ejwScheme}</code>	The request scheme (http, https, etc.)
<code>\${ ejwServerName}</code>	The server name the request called
<code>\${ ejwServerPort}</code>	The port the request called
<code>\${ ejwLocalName}</code>	The local server name
<code>\${ ejwLocalPort}</code>	The local port name
<code>\${ejwRemoteUser}</code>	Contains the remote user id

#{ejwRemoteUserInformation}	provides the following properties for accessing user information: <pre> #{ejwRemoteUserInformation.userId}, #{ejwRemoteUserInformation.salutation}, #{ejwRemoteUserInformation.firstName}, #{ejwRemoteUserInformation.lastName}, #{ejwRemoteUserInformation.roleMap} </pre>
------------------------------------	--

Application Messaging (messages to the user)

You can notify users of any condition (errors, warnings, notices, etc.) via application messaging. Application messaging can be accessed via ServerInterface with the following methods:

```

public String getApplicationMessage()
public void setApplicationMessage(String application_message)

```

The methods have no effect other than to set and retrieve messages.

To access them in your web page, use:

```

#{ejwApplicationMessage}

```

EJW Model/Business Interfacing

All Model interfacing (accessing business objects) is done by extending the RequestHandler or RestfulHandler classes:

```

public abstract class RequestHandler
{
    public String validateRequest(ServerInterface serverInt) { return SUCCESS; }
    public abstract String request(ServerInterface serverInt);
}

```

or:

```

public class RestfulHandler
{
    public String get(ServerInterface serverInterface, String[] args) { ... }
    public String head(ServerInterface serverInterface, String[] args) { ... }
    public String put(ServerInterface serverInterface, String[] args) { ... }
}

```

```
public String post(ServerInterface serverInterface, String[] args) { ... }
public String delete(ServerInterface serverInterface, String[] args) { ... }
public String options(ServerInterface serverInterface, String[] args) { ... }
}
```

and implementing or overriding the methods of interest. All parameters are optional and you can also use `HttpServletRequest` and/or `HttpServletResponse` as parameters.

If you need custom validation, then “`validateRequest`” is the method to override. The default implementation simply returns the constant `SUCCESS`. Your validation method implementation should return the constant `SUCCESS` for further processing via `request()`, or a forwarding URI.

The `request()` method is where you want to process any data submitted by forms or data being processed for the view. Your implementation should return a forwarding URI. A request can be forwarded any number of times to any number of other request handlers, but it should ultimately end in a view-handling request (ex. `page.jsp` or `page.vm`, depending on your view technology).

Any of the above methods can also return `NO_FORWARDING`, at which point request processing is terminated; no further processing will be performed. This is how you might handle a web event, such as a payment notification from PayPal. Or if you are handling the output directly with `response.getOutputStream()` or `response.getWriter()`.

ServerInterface

`ServerInterface` is the class that provides access to the servlet environment, and allows access to servlet-based objects such as the servlet request and response objects. An instance of `ServerInterface` is passed to the `RequestHandler` methods or obtained at anytime with `getServerInterface()`. You can also access request parameters (your web data), set attributes (outgoing view data), access session attributes (user-based in-between request state information), etc. There is only the one class, and it provides everything you need from a servlet/request handling viewpoint. For more information, view the Javadocs for `ServerInterface`.

Configuration Free Requests

In configuration free mode the request handler is found by matching all or part of the URI, minus the extension (if there is one), to a class found in the classpath.

For example, with a web request like the following:

```
http://www.website.com/anything/className/methodName/extraPathInfo
```

All path parts are case sensitive. Once the class is found, the class is loaded, and a new instance of the class is created and cached for all future requests for that URI.

Note, configuration based requests can be used and mixed freely with configuration free requests. Furthermore, even if requests are configuration free, configuration items such as:

```
<url id="" protocol="" url="" rewriteUrl="" />
<message id="" value="" />
```

are still very useful for soft coding URIs and text messages.

See the configuration manual for more details.

Configuring the EJW Servlet

EJW supports both servlet 3.x annotations, as well as, typical deployment discriptor (web.xml) definitions.

Servlet 3.x Annotations

NOTE: As you well know, annotations are equal to hard coding, so proceed with that knowledge.

By implementing the following:

```
@WebServlet(urlPatterns={"/"},
    initParams={
        @WebInitParam(name="param-name", value="param-value")
    })

public class SimpleController extends RequestServlet { }
```

you can create your own custom controller (EJW handles controller functionality automatically) to take advantage of Servlet 3.x annotations. This allows you to do away with the web.xml deployment descriptor. You can use any of the Servlet 3.x annotations.

You can also use the `@MultipartConfig` annotation for file uploads. However, EJW also supports file uploads and both the Servlet 3.x implementation and EJW use Apache's commons file upload package. The difference is EJW handles multipart form data requests dynamically, where as, the servlet spec 3.x requires the request to be multi-part form data.

Servlet Deployment Descriptor (web.xml)

The following Tomcat example is based on the standard Servlet specification 2.4, which you can download from Sun Microsystems' Java website at <http://java.sun.com>.

You can define the servlet with "RequestServlet" or any other name you like, which you will map requests to a little later. The servlet supports several initialization parameters, which are defined below.

```
<servlet>
  <servlet-name>RequestServlet</servlet-name>
  <servlet-class>ejw.RequestServlet</servlet-class>
  <init-param>
    <param-name>initDestroyHandler</param-name>
    <param-value>business.AppUtils</param-value>
  </init-param>
  <init-param>
    <param-name>classPrefix</param-name>
    <param-value>webapp</param-value>
  </init-param>
  <init-param>
    <param-name>handleMimeTypes</param-name>
    <param-value>jpg, swf</param-value>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>RequestServlet</servlet-name>
  <url-pattern>*.ejw</url-pattern>
</servlet-mapping>
```

Servlet (required)

The servlet class is always "EJW.RequestServlet" and is defined with:

```
<servlet>
  <servlet-name>RequestServlet</servlet-name>
  <servlet-class>EJW.RequestServlet</servlet-class>
</servlet>
```

where "servlet-name" is anything you like, and will be referenced later in the servlet-mapping definition.

initDestroyHandler (optional)

In the case where you have startup and/or cleanup needs with your web application, EJW allows you to define a class to handle servlet startup and shutdown events. This class is defined to EJW with:

```
<init-param>
  <param-name>initDestroyHandler</param-name>
  <param-value>java_class_without_extension</param-value>
</init-param>
```

The class must extend the class `InitDestroyHandler`:

```
public class InitDestroyHandler
{
  public void init(ServletContext context) throws Exception {}
  public void destroy(ServletContext context) throws Exception {}
}
```

The “init” method will be called with the servlets first use and the “destroy” method will be called when the servlet is shut down.

Example:

```
<init-param>
  <param-name>initDestroyHandler</param-name>
  <param-value>business.AppUtils</param-value>
</init-param>
```

classPrefix (optional)

The class prefix is used for configuration free requests and is simply prepended to the URI. This provides a secure method for locating classes dynamically.

Example:

```
<init-param>
  <param-name>classPrefix</param-name>
  <param-value>webapp</param-value>
</init-param>
```

urlRewriting (optional)

By default, EJW performs URL rewriting for `#{ejwUrl.key}`. If you do not want URL rewriting, you can turn it off with:

```
<init-param>
  <param-name>urlRewriting</param-name>
  <param-value>>false</param-value>
</init-param>
```

handleMimeTypes (optional)

By default EJW ignores mime typed requests (i.e. gif, jpg, swf, etc.) and simply passes them on to be handled as normal. If you want to handle these requests, you'll need to define the mime types with the following:

```
<init-param>
  <param-name>handleMimeTypes</param-name>
  <param-value>gif, jpg</param-value>
</init-param>
```

For example, you might want to do this when implementing an image server.

Servlet Mapping (required)

Servlet mapping is part of the servlet specification 2.4, which you can download from Sun Microsystems' Java website at <http://java.sun.com>. Servlet mapping goes hand-in-hand with the servlet definition above. You can define any number of mappings, and you can map any request pattern to the servlet.

Example:

```
<servlet-mapping>
  <servlet-name>RequestServlet</servlet-name>
  <url-pattern>/webapp/*</url-pattern>
</servlet-mapping>
```

Usual mappings are path-oriented mappings:

```
/path/*
/path/exact
```

or extension-oriented mappings:

```
*.ext
```

Special Topics

Logging

NOTE: With logging set to DEBUG, you can see diagnostic information including details of the web requests.

EJW uses slf4j (www.slf4j.org). Slf4j is a simple logger facade 4 Java. It supports the major logging frameworks such as log4j, Java logging, logback, commons logging, etc.

slf4j (required):

Download from: www.slf4j.org
Must have slf4j-api.jar in your classpath.

slf4j simple (defaults to info):

Download from: www.slf4j.org
Copy slf4j-api.jar and slf4j-simple.jar to your classpath.

logback (defaults to debug):

Download from: logback.qos.ch
Copy slf4j-api.jar, logback-classic.jar and logback-core.jar to your classpath.

log4j (requires configuration):

Download from: <http://logging.apache.org/log4j>
Copy slf4j-api.jar, slf4j-log4j.jar and log4j.jar to your classpath.

REST (RESTful Web Service) Requests

NOTE: Regular EJW request handling can also be RESTful, as the server interface will provide any information required to figure out what type of request is made, as well as, the path arguments sent.

RESTful requests are handled in a configuration free manner. The request is matched to a class:

```
public class MyRestfulHandler extends RestfulHandler
{
    public String get(ServerInterface serverInterface, String[] args) { ... }
    public String post(ServerInterface serverInterface, String[] args) { ... }
    ...
}
```

```
}
```

with an exact match of what is defined in the servlet mapping (the servlet path):

```
<servlet-mapping>
  <servlet-name>ejwRequestServlet</servlet-name>
  <url-pattern>/myRestfulHandler</url-pattern>
  <url-pattern>/myRestfulHandler/*</url-pattern>
</servlet-mapping>
```

Any additional path information is parsed and passed in to the request handler as a String array (String[] args).

With this definition, EJW can find any request handler for any supported URI and any supported HTTP method.

Within your request handler methods you can handle input with request.getInputStream() or request.getReader() and output with response.getOutputStream() or response.getWriter(). In this case you will want to return NO_FORWARDING or super.method(...).

However, it would be a huge waste to limit your self to this mode of input and output. You can still handle URL encoded data sent to the server and access it with getParameter().

More importantly you can return a forwarding URI to a template rendering engine such as JSP/JSTL or Velocity.

For example, if your RESTful GET request returns plain text with variable data, you can use:

```
<%@ page contentType="text/plain" %>

FirstName = ${contact.firstName}
LastName = ${contact.lastName}
Address = ${ contact.address}
...
```

or return a JSON formatted document for your JavaScript:

```
<%@ page contentType="text/plain" %>

// JSON formatted date

{"dayOfWeek" : "${date.dayOfWeek}",
"dayOfMonth" : "${date.dayOfMonth}",
"month" : "${date.month}",
"year" : "${date.year}",
"hour" : "${date.hour}",
```

```
"minute" : "${date.minute} ",
"second" : "${date.second} ",
"message" : "(Click for update)" }
```

and, you can just as easily handle XML, HTML, etc. based responses.

AJAX Requests

Handling AJAX requests is completely trivial. Your client can access any HTTP oriented request/response (with any HTTP method) that you set up with EJW in any mode (configuration free, RESTful, and/or configuration based).

And as described above, you can return a forwarding URI to a template rendering engine such as JSP/JSTL or Velocity:

```
<%@ page contentType="text/plain" %>

// JSON formatted date

{"dayOfWeek" : "${date.dayOfWeek}",
"dayOfMonth" : "${date.dayOfMonth}",
"month" : "${date.month}",
"year" : "${date.year}",
"hour" : "${date.hour}",
"minute" : "${date.minute} ",
"second" : "${date.second} ",
"message" : "(Click for update)" }
```

and you can just as easily handle XML, HTML, etc. based responses.

Default Request

You can define a default request that performs certain actions when no request is found to handle a requestURI. The default request is simply named default and can be defined in the same way:

```
webapp.Default.class
```

Usually the default request handler will simply forward all requests to the website's home page.

Error Handling

Any exceptions thrown back to the EJW servlet will be sent to /WEB-INF/error.jsp.

An all situations, an exception that is thrown back to the EJW servlet will be available to page rendering in `{ejwException}`, and a normalized error message will be available in `{errorMessage}`.

Security/User Authentication

EJW security is all about restricting requests based on security roles. EJW has built-in support for both container-based and application-based security.

If using container-based security, EJW automatically handles it and nothing further is required. EJW will use the `getRemoteUser()` and `isUserInRole()` methods defined in the `ServletRequest` class.

However, application-based security can flow more smoothly when the user is logging in and out. It is best done over a secure connection, but this is not required. If using application-based security, you will need to handle the login and call the `ServerInterface` method:

```
setRemoteUserInformation()
```

with the following arguments:

```
String user_id,  
String salutation,  
String first_name,  
String last_name,  
HashSet roles
```

“salutation”, “first_name” and “last_name” can be null and “roles” is just a `HashSet` of strings. From that point, EJW handles security the same way it would for container-based security.

Logging the user out can be done with:

```
ServerInterface.invalidateRemoteUser()
```

Secure Connections

Most secure connection functionality is handled outside the EJW application environment, as SSL (https) is set up at the container level, and a secure connection is then achieved simply by using “https” (HTTP over SSL) instead of “http”.

You can use EJW to restrict incoming requests via:

```
serverInterface.getServletRequest().isSecure();
```