pack:tag

a packed guide to website performance optimization

Abstract

This paper explains the integration, configuration and optimal usage of pack:tag, a Java JSP taglib for website performance optimization.
It also covers a basic practical understanding about website optimization in general.

Author: Daniel Galán y Martins

Intended audience: Developers, Website Designer/Maintainer/Administrators

Version: 1.0
Released: 2008-04-17

Found errors, typos or got some suggestions?
Please mail me to danielgalan at users dot sourceforge dot com, I heartily appreciate contributions.

Table of contents

Introduction

With the upcoming of Ajax and a stronger separation of content and design, many websites grow heavier in terms of kilobytes. A website that loads initially at over 600 kB is no longer an exception. Beside of the size of a website, the number of resources a website needs increases and therefore the number of requests.
Because of this trend, the user has to wait longer for his website to be loaded and rendered in his favorite browser, resulting in a user-experience that is suffering.
Server and bandwidth resources are wasted likewise.

Another problem is if the user returns to a website he visited earlier, and the websites resources has been changed since then (e.g. a new version has been deployed). The website can appear broken, because the browser caches some resources on its own to load the page faster. So even if the resource has been changed on the server, the client uses still the old one.

pack:tag addresses these problems for Java Servlet/JSP based environments, and solves them in a way transparent to the developer and user.

Resources

When a website is requested, several resources are loaded. A brief resource type overview:

- **HTML**
  HTML is usually generated dynamically in a Servlet/JSP environment, and therefore the delivered content is different each time. A possibility to increase download time is ad hoc gzip compression.
- **Images**
  Images reside statically in the file system, their content does not change in most cases. Beside of tweaking the image formats properties (e.g. JPG compression), one widespread method is to combine images which are same in width or height. This can reduce the number of request dramatically and is used mostly on button and icon sets. The name of this technique is know as CSS Sprites (explanation in detail: http://www.alistapart.com/articles/sprites)
- **CSS**
  CSS Files are static and written during development time. You can apply minification, compression and caching of the packed resources to save bandwidth and CPU.
- **JavaScript**
  To a JavaScript file apply the same conditions and rules as to CSS files.
- **Objects** (Flash, JavaFx, Java Applet, ..)
  Embedded Object like Java Applets are usually already compressed. You can not change the inner working from outside, so you have to deploy it as-is.

pack:tag can handle by default JavaScript and CSS.
If you have another custom resource type you want to pack, you can accomplish this by extending the class net.sf.packtag.tag.PackTag.
pack:tag also does offer a GzipFilter, that can be used to compress dynamically generated content like HTML on the fly.

Minification

Minification describes the process of removing unnecessary elements (characters) from a resource, without changing the logical semantics. Of course there are different algorithms out there, but in common they remove white spaces, tabs, line feeds, one-line and multi-line comments. This can shrink a file to about 60 percent of it's original size, depending on the coding style.

Let's look at an example, the following JavaScript file is not minified:

```
var ajax = {
```

```
      /**
       * Returns a XmlHttpRequest Object,that can be used for async. request.
       * Works with mozilla, opera, safari as well as with the ugly IE.
       *
       * @return A XmlHttpRequest Object
       */
      getXmlHttpRequest: function() {
            var xhr = null;
            // Mozilla, Opera, Safari as well as Internet Explorer 7
            if (typeof XMLHttpRequest != "undefined") {
                  xhr = new XMLHttpRequest();
            }
            if (!xhr) {
                  // Internet Explorer 6 and older
                  try {
                        xhr = new ActiveXObject("Msxml2.XMLHTTP");
                  }
                  catch(e) {
                        try {
                              xhr  = new ActiveXObject("Microsoft.XMLHTTP");
                        }
                        catch(e) {
                              xhr  = null;
                        }
                  }
            }
            return xhr;
      }

}
```

This code snippet takes 686 characters, now lets look at the minified version:

```
var ajax={getXmlHttpRequest:function(){var xhr=null;if(typeof XMLHttpRequest!
="undefined"){xhr=new XMLHttpRequest();}if (!xhr) {try {xhr=new
ActiveXObject("Msxml2.XMLHTTP");}catch(e){try{xhr=new
ActiveXObject("Microsoft.XMLHTTP");}catch(e){xhr=null;}}}return xhr;}}
```

After removing the unnecessary elements it only has 265 characters left, that's only 38,6 % of its original size. Of course it still can be interpreted by a browsers.

Compression

Beside of minifying, a resource can always be compressed. The compression happens on a binary level, and the compression ratio reaches between 65 and 85 percent.
Nowadays Browser support gzip, and most web servers have built-in gzip modules that only need to be activated.

Caching

Both minification and compression come at a cost of CPU. Every time a user requests a resource, the server has to minify and compress the same resource over and over again.
To prevent this, once a resource has been packed, it will be cached. This of course costs memory, but today memory is cheaper then CPU cycles.

Combination

Each resource you put on a web site has to be requested by the browser. Each request takes time because of network overhead (lookup, handshake, headers, etc.). You can decrease this by putting resources of the same type together, combining them. So instead of requesting e.g. five JavaScript files, you only need to request only one.

pack:tag

Features

pack:tag offers many advantages to your development and deployment:

- Minification of JavaScript and Cascading Style Sheets
- Caching to memory (default) or file system
- The minified content is delivered gzipped (memory)
- Combination of resources
- Combination of subdirectories (wildcard syntax)
- External resources can be packed and then served locally
- Cached resources are updated automatically on change
- Minification and compression adjustable on single resources
- Relative path support (memory)
- Charset support
- Strict XHTML conformity
- Apache Standard taglib support for evaluations
- Advanced caching techniques (etag configuration, 304 status header, expire header, ..)
- No resource will be written out twice
- Cloaking of resources

Benefits you get when compressing static resources with pack:tag:

- Reduced bandwidth
- Reduced number of requests
- Faster loading time for the client
- Browser caching side-effects of JavaScript files solved
- Obfuscation of the content (depending on the strategy, at least uglyfying) and the resource names, cloaking is also possible

Requirements

You can use pack:tag when the following conditions are fulfilled:

- Java VM 1.4 or higher
- A Java Servlet Webcontainer (e.g. Tomcat or Resin)

pack:tag does not have any dependencies to other libraries.
However, if Apache Log4j or Apache Commons Logging is found (in that order), messages will be send to the appropriate logger system. Elsewise they will be printed directly to System.out/err, so even these libraries are completely optional.

Integration

Download pack:tag and files

You can find the latest version of pack:tag online on sourceforge.net:
http://www.sourceforge.net/projects/packtag
Just click on the download button and get the latest packtag-x.y.zip file (where x.y is the current version number).

If you decompress the archive, you will get the following directories and files:

/files/packtag-x.y.jar	The library needed to run pack:tag, just copy it into the library path of your webapplication (usually WEB-INF/lib)
/files/packtag.properties	This is the configuration file for pack:tag, all settings are set to the default values, so you only need it if you plan to change the default behavior. This file then has to be directly in WEB-INF.

/files/packtag.user.properties	This is the configuration file for user-defined settings, they will override those in packtag.properties. This file comes in handy if you want your site-wide settings in your version control system, but override some settings during development. Your version control system should ignore this file (add this file to a .cvsignore file (CVS) or add a svn:ignore property (SVN)). This file has to be in the same directory as packtag.properties.
/files/web.xml	web.xml serves as template, you just need to copy the <servlet> and <servlet-mapping> for the PackServlet into your own web.xml, which is usually in your WEB-INF directory.
/javadocs	In this directory you find the generated JavaDocs.
/LICENSE.txt	The license file, pack:tag is licensed under the LGPL 2.1.
/README.txt	A short description of the archives content.
packtag - one minute quick start tutorial.pdf	A short introduction to pack:tag
packtag - a packed guide to website performance optimization.pdf	This document

If you wonder why there is no separate tld file any longer, it is bundled in the packtag-x.y.jar, because modern web containers auto detect those files. If you still need it, just extract it from the jar, it is located in the META-INF directory.

Installation

Just copy /files/packtag-x.y.jar into the library directory of your web application (usually WEB-INF/lib). After that you have to copy the <servlet> and <servlet-mapping> from the /files/web.xml into your web.xml descriptor file (usually directly in WEB-INF).

Maven2

Maven2 support is in progress, pom descriptors are available, but pack:tag is not uploaded to a maven repository yet.
Until then you have to install the library by yourself to your local repository:

```
mvn install:install-file -DgroupId=net.sf.packtag -DartifactId=packtag-core
-Dpackaging=jar -Dversion=x.y -Dfile=packtag-x.y.jar
```

Usage

To use the taglib in a JSP, you have to declare it first (like all taglibs):

```
<%@ taglib uri="http://packtag.sf.net" prefix="pack" %>
```

Now you can easily pack JavaScript with the following tag:

```
<pack:script src="/myJavaScriptFile.js"/>
```

Accordingly for Cascading Style Sheets:

```
<pack:style src="/myCascadingStyleSheet.css"/>
```

Sources

You can point to resources in various ways, pack:tag offers following possibilities:

Contextpath

You can point to a file directly from the root of your contextpath, that means starting with a leading

slash:

```
<pack:script src="/js/util/common.js"/>
```

This way it doesn't matter from where you point to that file, because it is an absolute notation.

Relative

Image your JSP is located in "/my/account/login.jsp", and the css "/my/account/login.css" is located in the same subdirectory, so you can use a relative notation in that JSP:

```
<pack:style src="login.css"/>
```

If you want to point to another file relative to the current subdirectory (e.g. "/my/account/logout/logout.css") you can write the following:

```
<pack:style src="../logout/logout.css"/>
```

Wildcard

A new way to define resources in pack:tag is the wildcard notation. With this notation you can pack all the files in a directory with one declaration.
The next example packs all JavaScript files in the directory "/js":

```
<pack:script src="/js/*"/>
```

If you want to include all of the files in a directory and its subdirectories, do the following:

```
<pack:script src="/js/**"/>
```

Of course you can use a relative path like here:

```
<pack:script src="account/**"/>
```

External

You can also point to external resources. pack:tag will download, minify, compress and cache them for you. External resources start with http:// or https://, an example:

```
<pack:style src="http://www.somedomain.com/css/main.css"/>
```

You can override this behavior globally by setting "resources.external" to "false" or on a single resource by setting the attribute "enabled" to "false".

Attributes

Besides the src attribute, pack:tag has some other useful options you can set per tag individually:

Attribute name	Type	Mandatory	Description
src	String	Yes	Path to the resource (see above)
enabled	Boolean	No	When set to true, the resource will be written out directly. No compression or minification takes place.
minify	Boolean	No	Disables the minification when set to true. pack:tag still delivers the resource - compressed and with all caching headers (servlet).
media	String	No	Defines the output media. This attribute is only applicable for pack:style.

Combining resources

With pack:tag, you can combine resources easily. See the following code:

```
<pack:script src="/js/validation.js"/>
<pack:script src="/js/tracking.js"/>
<pack:script src="/js/edges.js"/>
```

This code produces three packed resources, each resulting in a request the browser has to send and receive. We can combine them simply by listing them up, so only a single resource will be delivered:

```
<pack:script>
      <src>/js/validation.js</src>
      <src>/js/tracking.js</src>
      <src>/js/edges.js</src>
</pack:script>
```

Inside a combined resource you can mix all forms of sources as we can see here in this extreme example:

```
<pack:style>
      <src>/main.css</src>
      <src>../logout/logout.css</src>
      <src>/css/**</src>
      <src>http://www.example.com/css/browserfixes.css</src>
      <src>/WEB-INF/css/hidden.css</src>
</pack:style>
```

Configuration

pack:tag offers many configuration possibilities. The easiest and preferred way to change them is to copy the packtag.properties file into your WEB-INF, and then change the appropriate setting like this:

```
cache.type=servlet
```

Another possibility to configure pack:tag is as context parameter in the web.xml file.
Here is an example (note that you have to prefix the setting name with "packtag."):

```
<context-param>
      <param-name>packtag.cache.type</param-name>
      <param-value>servlet</param-value>
</context-param>
```

Notice that settings in the packtag.properties file override context parameters settings.

Thus methods are great for deployment, but what if you have to locally change the setting temporarily? You can create a file called packtag.user.properties in the WEB-INF directory, that overrides the changes in the web.xml and packtag.properties file.
This file is for user-defined settings (e.g. to set the cache.type to disabled during development) and should not be checked into your code-repository. The best way to achieve this is to add the packtag.user.properties file to the .cvsignore file, svn:ignore propset, etc. (your VCS here).

Configuration Options

Overview of the existing settings:

Name	possible values	default
cache.type	servlet, file, disabled	servlet
cache.file.path	valid path	pack
cache.servlet.combined.<resourcetype>.path	valid path	(empty)
resources.checktimestamps	true, false	true
resources.tracking	true, false	true
resources.external	true, false	true
resources.charset	JVM supported charset name	Java >= 5: Platform default Java = 1.4: Latin9

hide.errors	true, false	false
<resourcetype>.strategy	Classname to PackStrategy implementing Class	JavaScript: JsminPackStrategy Css: IBloomCssPackStrategy

The settings in detail:

cache.type

This defines the way how pack:tag caches and delivers the resources for you.
In short: when set to "servlet" the minified data is cached in memory, if "file" the minified data is written to a file.

Here is a more comprehensive overview:

cache.type	**servlet (default)**	**file**	**disabled**
Minification	Yes	Yes	No
Basic browser caching issues	Yes	Yes	No
Additional browser caching issues (header, expiration, ..)	Yes	No	No
Send gzipped	Yes	No	No
Relative Path support	Yes	No	Yes

Note that you lose the benefit of using relatives path declarations when working with cache type "file".

The cache type "disabled" is useful during development, when you debug your JavaScript code.

I suggest to use the cache type "servlet", which is the default, to get the maximum out of pack:tag.

cache.file.path

When you set your cache.type to file,the generated files will be saved in a directory with the name "pack". If you want to change this default name to something else, set the parameter "cache.file.path" to the desired directory name. Your server process must have write access to this directory.

cache.servlet.combined.<resourcetype>.path

This defines the path of the combined packed resources (per resource type), and is only possible when "cache.type" is set to "servlet".
Example: Setting the combined path for JavaScript files to "/js", write the following:

```
cache.servlet.combined.js.path=js
```

Example: Same applies to CSS files, for the directory "/css/combined" write the following:

```
cache.servlet.combined.style.path=css/combined
```

resources.checktimestamps

pack:tag checks by default for each request if the underlying resource has been changed (via timestamp). If it has changed, the resource will be reloaded and packed again.
If a file from a combined resource has been changed, all files from the combined resource will be reloaded and packed.
To disable the timestamp checks, set this option to "false".
Note that you have to set this option to "false" if you work with unexploded wars.

resources.tracking

Each time you include a resource in a JSP, it is also remembered in the request. This way, a resource is only written out once, even it is included multiple times.
You can disable this by setting "resources.tracking" to "false".

resources.external

As explained earlier, pack:tag downloads files from external locations (those starting with http:// or https://), minifies, compresses and serves them locally. You can, however, disable this behavior and let pack:tag write out the source directly. This works for single resources as well as for combined ones.

resources.charset

This defines the character set used for encoding, when loading the resources from disk.
If not set, the default charset of the JVM is used (Java >= 1.5). If you are still using Java 1.4, Latin9 will be assumed.
A list with the supported charsets on your system can be found by calling the Charset.availableCharsets() method.

hide.errors

If an error occurs (e.g. file not found), the system will throw an exception for you, so you can see the error during development immediately.
You can hide those errors by setting this option to "true", so an exception will not be shown to the user. The resource will not be written to the rendered page, but a stack trace will be written in a HTML comment.

<resourcetype>.strategy

Each resource type needs an algorithm how it can be minified. You can set the algorithm by defining a class that implements it.
To set the PackStrategy for JavaScript files, set "script.strategy" to a full qualified classname, e.g.:

```
script.strategy=net.sf.packtag.implementation.yui.YuiCompressorPackStrategy
```

To set the PackStrategy for CSS files, set "style.strategy" to a full qualified classname, e.g.:

```
style.strategy=net.sf.packtag.implementation.yui.CssCompressorPackStrategy
```

See the next chapter for a detailed explanation of the integrated PackStrategies.

PackStrategies

PackStrategies encapsulate existing minification algorithms, so they could be plugged into pack:tag.

Available PackStrategies

pack:tag comes with some build-in PackStrategy implementations, as well as strategies that wrap algorithms from external libraries.
The default implementation for JavaScript is "JSMin", for CSS "iBloom CSS Compressor." Let's take a look at the minification algorithms in detail:

JSMin

Resource type: JavaScript
Class: net.sf.packtag.implementation.JsminPackStrategy
Author: Douglas Crockford
Website: http://www.crockford.com/javascript/jsmin.html
Licence: BSD like

iBloom CSS Compressor

Resource type: CSS
Class: net.sf.packtag.implementation.IBloomCssPackStrategy
Author: Nicholas Gagne
Website: http://www.ibloomstudios.com/articles/php_css_compressor
Licence: None

YuiCompressor

Resource type: JavaScript
Class: net.sf.packtag.implementation.yui.YuiCompressorPackStrategy
Author: Julien Lecomte
Website: http://www.julienlecomte.net/yuicompressor
Licence: BSD

Notice that the YuiCompressor depends on Mozilla Rhino (http://www.mozilla.org/rhino), which is dual licensed under MPL 1.1/GPL 2.0.
So you have to download the YuiCompressor jar by yourself (which bundles Mozilla Rhino), in order to use the YuiCompressor PackStrategy.

Foohack CSS Compressor

Resource type: CSS
Class: net.sf.packtag.implementation.yui.CssCompressorPackStrategy
Author: Isaac Schlueter
Website: http://www.julienlecomte.net/blog/2007/08/14 (announcement), http://foohack.com
Licence: BSD

You have to download the YuiCompressor jar by yourself, in order to use this PackStrategy.

Jawr Css Minifier

Resource type: CSS
Class: net.sf.packtag.implementation.jawr.CssMinifierPackStrategy
Author: Jordi Hernández Sellés
Website: https://jawr.dev.java.net
Licence Apache Licence 2.0

Notice that you need to download the jawr library to use this PackStrategy.

Setting the PackStrategy

You can change the PackStrategy for a resource type by the "<resourcetype>.strategy" setting. Take a look in the previous configuration options chapter for detailed information.

A short example, to set the JavaScript PackStrategy to the YuiCompressor, write the following in your packtag.properties file:

```
script.strategy=net.sf.packtag.implementation.yui.YuiCompressorPackStrategy
```

To set the style PackStrategy from to the Foohack CSS Compressor:

```
style.strategy=net.sf.packtag.implementation.yui.CssCompressorPackStrategy
```

Developing your own PackStrategies

A PackStrategy is nothing but a very simple Interface:

```
public interface PackStrategy {
      public String pack(InputStream resourceAsStream, Charset charset)
                                                        throws PackException;
}
```

As you can see, a resource is delivered as an InputStream, as well as the used Charset.
For your convenience, an abstract implementation that reads the stream and delivers a String called AbstractPackStrategy exists.
So your minification algorithm can work either directly on the stream or, if you prefer, on a String.

Best practices and FAQ

Cloaking resources

In a typical Java web-application you put all your requestable resources (like JSPs, JavaScript, CSS, images, etc.) under a special directory called "webapp". All resources can be accessed by the browser directly (except those who are intercepted by the webcontainer for execution, like JSP or Servlets).
When a JavaScript file is put under "/webapp/js/app.js", the user can access that file in his browser by simply requesting "http://www.yourdomain.com/js/app.js". In this case he can see the file formatted and with all comments.

This is common practice, but can be avoided. Inside the webapp directory, another special purpose directory exists: "WEB-INF". "WEB-INF" contains meta-information about your web-application and content inside this directory can not be accessed by the user.
So instead putting your JavaScript or CSS file somewhere inside your "webapp" directory, put them in your "WEB-INF". This would usually not work, because the user can not access "WEB-INF", luckily you are using pack:tag. Let's take the example from above and refer to it with pack:tag:

```
<pack:script src="/WEB-INF/js/app.js"/>
```

This will will be rendered as follow:

```
<script type="text/javascript" src="/js/app.js.h1995556352.pack"
charset="utf-8"></script>
```

As you can see, pack:tag maps the file one subdirectory down to a virtual location.

This solution has only one drawback: when you disable pack:tag, no resource can be found.

Gzipping dynamic content like generated HTML (GzipFilter)

As mentioned in the introduction, only static resources can be cached, otherwise they have to be minified and/or compressed at runtime, which costs more CPU but saves bandwidth.
To apply compression to dynamic content, you can use the GzipFilter pack:tag includes. The GzipFilter was originally written by Jayson Falkner.
In order to use the compression for generated HTML, declare the GzipFilter in your web.xml:

```
	<filter>
		<filter-name>GzipFilter</filter-name>
		<filter-class>net.sf.packtag.filter.GzipFilter</filter-class>
	</filter>

	<filter-mapping>
		<filter-name>GzipFilter</filter-name>
		<url-pattern>*.jsp</url-pattern>
	</filter-mapping>
```

This applies compression to all requested JSPs. Don't use an url-pattern like "*", because this can lead to errors when resources are compressed twice.
To enable the GzipFilter e.g. for a Struts-based application add the following filter-mapping:

```
	<filter-mapping>
		<filter-name>GzipFilter</filter-name>
		<url-pattern>*.do</url-pattern>
	</filter-mapping>
```

Combined Resources mistakes

I have seen people using combined resources in the following way: they wanted only the resources

that were needed on the current page. As honorable this seems, it produces larger downloads.
Let's look at an example.
On page 1 three resources were needed:

```
<pack:script>
      <src>/js/app.js</src>
      <src>/js/effects.js</js>
      <src>/js/tools.js</src>
</pack:script>
```

On page 2 the same three resources were needed, plus some validation for forms:

```
<pack:script>
      <src>/js/app.js</src>
      <src>/js/effects.js</js>
      <src>/js/tools.js</src>
      <src>/js/validation.js</src>
</pack:script>
```

On page 3, only two of them were really needed, so lets pack them:

```
<pack:script>
      <src>/js/app.js</src>
      <src>/js/tools.js</src>
</pack:script>
```

Obviously, this results in three different resources, that has to be downloaded on each site again, even if two resources are always the same.
There are some approaches to solve this:

1. Always deliver all resources
   a. By explicit naming:
   ```
   <pack:script>
         <src>/js/app.js</src>
         <src>/js/effects.js</js>
         <src>/js/tools.js</src>
         <src>/js/validation.js</src>
   </pack:script>
   ```
   b. By wildcard naming:
   <pack:script src="/js/*"/>
2. Deliver only the core as combined resource:
   This means for page 2 that two resources will be delivered as single requests:
   ```
   <pack:script>
         <src>/js/app.js</src>
         <src>/js/tools.js</src>
   </pack:script>
   ```
   And the rest as single resource
   ```
   <pack:script src="/js/effects.js"/>
   <pack:script src="/js/validation.js"/>
   ```
3. Don't deliver combined:
   Instead you have to deliver all resources as single requests:
   ```
   <pack:script src="/js/app.js"/>
   <pack:script src="/js/tools.js"/>
   <pack:script src="/js/effects.js"/>
   <pack:script src="/js/validation.js"/>
   ```

You have to decide which approach fits your development, deployment and performance strategy best.

Ending semicolon in JavaScript

JavaScript is a dynamic language, that has also very lax restrictions on its syntax. You don't have to end a statement in a single line with a semicolon.
However, a minified version of a JavaScript file with such a syntax would end in errors, either on

minification, or later on client side. So I advice you to write clean code.
The same rules apply to other elements like curly braces.

Exploded/unexploded war

pack:tag works well with exploded wars. If you restrain the automatic war extraction, you have to set the "resources.checktimestamps" setting to "false".

JavaScript libraries

Some JavaScript libraries can cause headaches, here some of them:

prototype

prototype can not be minified because the syntax isn't very minifier friendly (missing semicolons, etc.). There exists prepacked prototype libraries, but they are not maintained by the originators. Instead you could use the minify="false" attribute, this way prototype is not minified but compressed (24kb is better then 101kb ;)). You might also want to try the YuiCompressor.
Further readings:

- http://dev.rubyonrails.org/ticket/7311
- http://andrewdupont.net/2007/02/26/packing-prototype

script.aculo.us

script.aculo.us with loader options doesn't works. This happens because the scriptaculous.js is just a loader stub. Use the packer on the specific files (e.g. slider.js) instead.

Manual cache flushing

You can clean pack:tag's cache with the following line of code:

```
PackCache.clearCache();
```

Appendix

Thanks

In alphabetically order:
- Dr. Michael Sievers
- frenchyooy@sf.net
- qxo@sf.net
- Ryan Gardner
- Sicke Westerdijk
- Stuart Batty
- Tarun Reddy
- Thomas Duerr

Licenses

pack:tag

This document

Similar projects

There exists some other projects out there, that have similar goals:

Jawr

Jawr is a bundling and compression solution for web application resources.
In contrast to pack:tag, you have to declare all resources you want to use in so called "bundles" first. You do this with a special notation in property files.
Website: https://jawr.dev.java.net

Resource Accelarate

The Resource Accelerate from Xucia is a Filter that provides compression, header directives and JavaScript minification.
Website: http://www.xucia.com/#Resource%20Accelerate

JavaScript Optimizer JSO

As the website says: A project that allow you to manage easily your JavaScript and CSS resources and to reduce the amount of data transfered between the server and the client.
Website: http://js-optimizer.sourceforge.net

Bundle-Foo

This does minification, combination and header directives for Ruby.
Website: http://code.google.com/p/bundle-fu/

Tools

No article to website optimization would be complete without mentioning the most outstanding web development tools:

- FireBug - http://www.getfirebug.com
  THE web development inspection and analysis tool
- YSlow - http://developer.yahoo.com/yslow
  Yahoo's performance analysis tool

With this tools you can analyze your web traffic in detail, and track performance issues.

Further readings and links

- http://stevesouders.com/hpws
  14 Rules for Faster-Loading Web Sites, from the book High Performance Web Sites by Steve Souders. Worth reading (covers the yslow rules).
- http://www.slideshare.net/stoyan/high-performance-web-pages-20-new-best-practices
  High Performance Web Pages Presentation, 20 new rules by Yahoo
- http://www.alistapart.com/articles/sprites
  Explaination of image combination in CSS, called sprites.
- http://dean.edwards.name/packer
  Another minification algorithm, maybe you can create a Java version that can be plugged into pack:tag? :)

Additional Sources

Articles about pack:tag

- http://blog.augmentedfragments.com/2008/01/compressing-and-obfuscating-javascript.html
- http://www.selfcontained.us/2008/02/29/combine-your-js-and-css-files-with-packtag/

Miscellaneous:

- pack:tag on ohloh - http://www.ohloh.net/projects/7677

Feedback

If you have suggestions, found bugs, typos or want to give feedback in general, please go to the project page, hosted on sourceforge, and leave a message:
http://www.sourceforge.net/projects/packtag