

Using multiple Struts configuration files

Skill Level: Introductory

[Brett McLaughlin, Sr. \(brett@newInstance.com\)](mailto:brett@newInstance.com)

Author/Editor

O'Reilly Media, Inc.

22 Nov 2005

Breaking a large configuration file into smaller, more manageable parts makes Struts applications easier to organize and maintain. In this tutorial, Brett McLaughlin shows how to set up Apache Struts to use multiple configuration files. The tutorial reviews Struts configuration, takes you step-by-step through execution of a divide-and-conquer configuration strategy, and guides you through some additional configuration cleanup options.

Section 1. Before you start

About this tutorial

This tutorial shows Java™ Web developers how to set up Apache Struts to use multiple configuration files. You'll learn about the purpose and structure of the various Struts configuration files and the rationale for working with smaller files. You'll follow steps to split a large, complex struts-config.xml file for an existing Struts application into multiple configuration files that are organized by function. Having smaller, more manageable parts to work with makes a Struts application easier to organize and reconfigure, and it can help you more easily identify points of failure when problems occur. You'll also learn about some other types of cleanup you can do to improve your Struts configurations.

Who should take this tutorial?

This tutorial is written for Java Web developers who have at least some basic experience with Apache Struts and the Apache Tomcat servlet engine. You should know how to start and stop Tomcat, install Struts, and deploy a Struts application.

You can use a servlet engine other than Tomcat (Struts plays quite well with other servlet engines), provided you know how to set it up and configure it. The tutorial assumes you're using Tomcat and provides no extra detail for non-Tomcat configurations. See [Resources](#) for a link to Struts documentation on installing Struts on non-Tomcat servlet containers.

Struts uses XML configuration files, so you should be comfortable with XML. But you don't need to be a configuration guru because I'll provide some basic information on how Struts uses these files before I go into the details of breaking them up.

Prerequisites

You need a machine -- or ISP -- that has a servlet engine installed. I highly recommend you run through this tutorial either locally, on a development machine, or on a nonproduction ISP account. Don't use a machine serving thousands of users because you'll need to make changes to your servlet container (and if your servlet engine doesn't automatically reload the changes, you'll also need to restart the container manually).

Although neither is required for this tutorial, I'm using Tomcat 5.5.9 and Java 5.0 (Release 1) on Mac OS X. Java 1.4 works just as well, but you might see some compiler warnings in the tutorial examples that you won't see with earlier JVMs. I urge you to use at least version 5.0 of Tomcat; 5.5.x is even better. You also need a text editor of your choice for working with XML.

Obviously, you need Struts installed; if you're unclear on how to install Struts, check out the appendix of [Using the Struts Validator](#), which walks you through the entire process step by step. You can also refer to [Resources](#) in this tutorial for several more good links to help you out.

Finally, in this tutorial you'll configure the Struts Online Banking application, described in *Programming Jakarta Struts* by Chuck Cavaness (see [Resources](#)), so you need to obtain and install it as well:

- Go to <http://examples.oreilly.com/0596006519/>.
- Download jakarta2ed-source.zip.
- Expand the ZIP file.
- Drop banking.war into your servlet engine's Web application directory.

Section 2. Configuration matters

I still remember when an application's *configuration* meant -- at best -- one or two text files that stored a username and password for connecting to a database. The application would read these files in (using hand-written code) and then largely forget that the files ever existed. That's what I like to think of as Really Stupid Configuration. Don't laugh -- there are developers the world over who store the data their application needs in a proprietary format, read these values in using their own proprietary code, and never organize (or reorganize) these files. Maybe you're even thinking sheepishly about an application you recently wrote that does the same sort of thing. It's okay, though; by the end of this section, you'll know more about configuration, why it's important, and how it can help you.

Applications, configuration files, and sleeping at night

When this tutorial uses the terms *application configuration data*, or just *configuration*, I'm referring to a specific type of data that is common to every application. This data is used in setting up the application; it governs how the application sets certain options or features. In some cases, this data might be a simple username and password combination, perhaps for connecting to a database or directory server. In other cases, it might indicate whether the application should run in debug mode, or log user requests, or let managers as well as executives log in. In more sophisticated applications, configuration data might indicate which modules the application should start up, and even how those modules themselves should behave.

In all of these cases, though, this isn't data that is changing all the time. It's fairly static (ever changed the username and password your application uses to talk to a database more than once in a blue moon?), and it's typically not data a user or potential customer enters. More important, it's not data that the application's business logic produces or consumes, such as an order detail or charge amount. Instead it's used only by the application itself, behind the scenes.

This *type* of data is common to all applications; the data itself is not. An application's configuration data might differ across countries, companies, and even instances of the same application within the same company. However, to set itself up correctly, the application always uses *some* set of data that doesn't need to change often, if at all. If you can isolate this data and pull it out of your code into a static resource -- configuration files, for example -- then you're way ahead of the game. When that blue moon comes along and your database connection *does* need to change, you

can simply update the username and password, possibly restart the application, and go back to bed. Because, in the real world, those sorts of changes almost always happen at 4:00 a.m., right? Who wants to be slogging through code updating hardcoded strings and recompiling in the middle of the night? Good configuration usually means changing one or two values in a text file and rolling back over to sleep.

Declarative programming

One special type of configuration you should be aware of is *declarative programming*. Declarative programming is the idea of declaring certain aspects of a program and then letting some engine turn those declarations into actual code. You're programming, but you're not actually writing code -- a strange, but important, paradigm. As a brief example, take a look at Listing 1, which is a portion of a simple struts-config.xml file:

Listing 1. Declarative programming in struts-config.xml

```
<struts-config>
  <action-mappings>
    <action path="/login"
            type="pizza.delivery.security.LoginAction"
            name="loginForm"
            scope="request"
            input="/loginForm.jsp">
      <exception key="error.failedLogin"
                type="pizza.exception.InvalidPasswordException"
                path="/loginForm.jsp" />
      <exception key="error.failedUsername"
                type="pizza.exception.InvalidUsernameException"
                path="/loginForm.jsp" />
    </action>
  </action-mappings>
</struts-config>
```

In this fragment, two exceptions are defined, correlating to two specific problems that can occur when a user tries to log in. The first deals with a bad password and the second with a bad username. In each case, action is directed to a predefined exception class, coded up somewhere else in the application. What's notable here is that this is *all* that's needed to handle these errors. You don't need any code in your form to check for a bad username and then manually throw a `InvalidUsernameException`; the same is true for bad passwords. Struts handles all this for you, because you've declared the error cases and what to do when they occur. In this way, you can convert certain tasks that generally require programming into configuration data.

Struts provides many more declarative options, from defining errors to defining actual forms, error messages, internationalization, and more. By simply stating what possible things can happen -- or which fields should be on a form, for example --

Struts can create and direct resources to the right place. This is configuration at its best, in many ways. To add a new error case (perhaps you want to lock out accounts with three bad password attempts), you only need to edit some XML; no extensive coding, compiling, and testing is involved. Pretty cool, and *extremely* useful for adding small features to an application without cratering thousands of lines of existing, tested, stable code.

Further discussion of declarative programming is far beyond this tutorial's scope. However, declarative programming has become such an important idea -- in Struts and in programming in general -- that I couldn't leave it out of any extended discussion on configuration. So this bit was for free!

Struts configuration: An overview

There's much more to Struts configuration than the bits I've shown you so far. You'll learn much more in the next section, [Struts configuration 101](#), but here's a quick overview.

First, Struts is a servlet-based Web application, so it needs the standard `web.xml` descriptor common to all Web applications and `MessageResources.properties` for messages and application-wide text properties. Then, there are Struts-specific configuration files:

- **struts-config.xml** handles basic Struts configuration tasks. You define global constants, as well as forms, actions, and just about anything else you can think of.
- **validator-rules.xml** defines rules that can be used for ensuring form data integrity, such as the minimum length of a ZIP code or the format of a phone number.
- **validation.xml** handles linking the validation rules up with specific forms and actions, gluing together your validation rules with the data defined in the `struts-config.xml` file.

Add to that your code itself -- as well as any configuration files you create on your own for storing application-specific data -- and you've got a lot to keep up with. And, because of declarative programming, the configuration files hold more than just a few usernames and passwords. By the time you've defined your forms, your actions, your errors, and basic startup options, you've got a *large* `struts-config.xml`. That, of course, is the subject of this tutorial.

Sharing the load

Programmers are quick to endorse a "divide and conquer" approach to development but often fail to practice what they preach. Writing a program that can be split into multiple modules, functions, and so on takes effort. But it's worth doing. The more you break up code, the more you can involve multiple developers (perhaps delegating easier portions to junior developers and harder pieces to the more experienced ones) and (at least in theory) expedite the project.

"Divide and conquer" is a smart design for configuration data too. Breaking up a large file, such `struts-config.xml`, into lots of smaller, easier-to-manage parts pays off. You can organize the smaller files by function or by module to create mini-applications, each dramatically simpler to manage than a huge monolithic application.

Section 3. Struts configuration 101

This section walks you through the Struts configuration files and dissects their contents, at least at a high level. If you're an experienced Struts developer, much of this might be review. However, it's worth a quick reread, because you need to understand how these files are organized when you start breaking them up into smaller pieces. Newer Struts developers can use this section as a Struts configuration primer. It won't give you everything you need to be a Struts master, but you'll certainly pick up on some areas of configuration that you might not have had occasion to use in your own programming yet.

Finding Struts' configuration files

Once you've downloaded the sample Struts application (see [Prerequisites](#)), navigate to the `WEB-INF/` directory. (If you haven't already done so, you need to expand the sample application WAR file with the `jar` command.) You'll find several files in this directory. You'll be focusing on two of them: `struts-config.xml` and `web.xml`. The `struts-config.xml` file is a Struts-specific configuration file that tells Struts how to configure itself, where application resources are located, what error messages to report, and a lot more. The `web.xml` file is the standard Web application deployment descriptor, used largely to communicate some general servlet information to the servlet engine itself.

You'll spend most of your time in Struts development working with `struts-config.xml`, and that's where you'll spend much of your time in this tutorial. Before diving into how you can break this file up, though, you need to understand thoroughly what these configuration files do and how their functionality is organized. If you

understand the structure of these files -- struts-config.xml in particular -- you'll be able to split the configuration data up in a logical way that won't foul up your application.

Using struts-config.xml

The struts-config.xml file is located (at least for now) in the WEB-INF/ directory of the Struts sample Web application. Open this file and take a look; a portion of it is shown in Listing 2 for reference:

Listing 2. Part of struts-config.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
  "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>
  <form-beans>
    <form-bean name="loginForm"
      type="com.oreilly.struts.banking.form.LoginForm" />
    <form-bean
      name="accountInformationForm"
      dynamic="true"
      type="org.apache.struts.action.DynaActionForm">
      <form-property name="accounts" type="java.util.ArrayList" />
    </form-bean>
    <form-bean
      name="accountDetailForm"
      dynamic="true"
      type="org.apache.struts.action.DynaActionForm">
      <form-property name="view"
        type="com.oreilly.struts.banking.view.AccountDetailView" />
    </form-bean>
  </form-beans>

  <global-exceptions>
    <exception
      key="global.error.invalidlogin"
      path="/login.jsp"
      scope="request"
      type="com.oreilly.struts.banking.service.InvalidLoginException" />
  </global-exceptions>

  <global-forwards>
    <forward name="Login" path="/login.jsp" />
    <forward name="SystemFailure" path="/systemerror.jsp" />
    <forward name="SessionTimeOut" path="/sessiontimeout.jsp" />
  </global-forwards>

  <action-mappings>
    <action
      path="/login"
      type="com.oreilly.struts.banking.action.LoginAction"
      scope="request"
      name="loginForm"
      validate="true"
      input="/login.jsp">
      <forward name="Success"
        path="/action/getaccountinformation" redirect="true" />
      <forward name="Failure"
```

```
        path="/login.jsp" redirect="true"/>
    </action>
<!-- and so on... -->
</action-mappings>

<controller
  contentType="text/html; charset=UTF-8"
  debug="3"
  locale="true"
  nocache="true"/>

  <message-resources parameter="BankingMessageResources" null="false"/>
</struts-config>
```

Each of the several basic sections in Listing 2 has its own purpose:

- **Document header:** This is the XML declaration and the `DOCTYPE` statement. These aren't configuration data as much as they are specific to XML and validation. They'll be the same for virtually every Struts application you ever write.
- **Data sources:** This section is nested within the `struts-config` root element, inside a `data-sources` element. Each data source -- represented by a `data-source` element -- defines database connections that your application can use. Although there aren't any data sources defined within the sample application, this is a pretty common occurrence in most enterprise applications.
- **Form beans:** This next section details form beans used in your application. Form beans let you detail the information that is submitted by a certain type of form and avoid coding up JavaBeans for all of your forms and JavaServer Pages (JSPs). The section is noted with a `form-beans` element, and then each bean is delineated with a `form-bean` element.
- **Global exceptions:** After the form beans come the global exceptions, nested within a `global-exception` element. These are exceptions that any portion of the application can use. For example, if a user times out or enters a correct password, lots of places in an application might need to issue an error. This type of global exception is great for reducing duplicated code and unifying your error messages across large enterprise applications.
- **Global forwards:** Next up are global forwards, within the `global-forwards` element. Each forward -- indicated with a `forward` element -- provides an application-wide means of getting to a particular page. You might have a generic error-handling page, and using a global forward means all resources refer to the same object for that page. Even nicer, if you change the error page, you can change its URL here, and all your resources will suddenly get the advantage of the new path. (This is the beauty of configuration files!)

- **Action mappings:** After you've handled exceptions and forwards, the actions of the application are defined. These are each represented by an `action` element, all nested within an `action-mappings` element. These actions map a specific URI -- such as `/logout` -- to a Struts `Action` class, such as `com.oreilly.struts.banking.action.LoginAction`. Again, you can easily change the implementation of an action simply by making a change in this configuration file.
- **Controller instructions:** In Struts 1.1 and later, you can set many properties of the Struts controller (using the `controller` element). Struts now uses a declarative configuration for its controller, so using this section of the configuration file is a lot more convenient than needing to write your own controller servlet (not a particularly fun task).
- **Message resources:** If you choose to use a resource bundle for your application, you can pass in parameters to configure how the resources are used (for example, what should occur if an invalid key is supplied). This configuration is done via the `message-resources` element.
- **Plug-ins:** You can reference any plug-ins your application might want to use via the `plug-in` element. If you've read the tutorial on using the Struts Validator (see [Resources](#)), you're probably already familiar with this functionality. In fact, the Struts Validator is probably the most common Struts plug-in.

This is hardly an exhaustive look at `struts-config.xml`, but it should give you an idea of the basic sections. These will be important when you go about breaking up the configuration into multiple files, because you'll want to have some method to your madness. For example, you can break configuration up logically and put all the actions, form beans, and errors that have to do with one section of your application into one file, and put the configuration components of another part of your application in a different file. Or, you might prefer to break up your files by component, keeping all actions in one file, and all exceptions in another. Whatever your choice, you need to understand this organization to make good decisions.

Using `web.xml`

You need to let your servlet engine know how to deal with the Struts application. `WEB-INF/web.xml` is the place to do this. Listing 3 shows a portion of the sample Struts application's `web.xml` file:

Listing 3. Part of `web.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
```

```
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>banking</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet
      </servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>

    <load-on-startup>1</load-on-startup>
  </servlet>
  <!-- Other entries -->

</web-app>
```

Most of Listing 3 is concerned with locating the Struts configuration data. The `config` parameter lets Struts know where to find its configuration data. (In this case, the location is simply `WEB-INF/struts-config`, although that will soon change.) Most of the rest of Listing 3 should be pretty common for servlet developers; the Struts `ActionServlet` is given a name -- `banking` -- and then loaded on startup.

The sample application's configuration includes several other entries, most of which deal with tag libraries. While those are interesting in themselves, they're really not the focus of this tutorial, so I'll leave them for you to investigate on your own.

Odds and ends

I'll run through a few other configuration items of note quickly, just to complete your 10-minute tour through Struts configuration land:

- **Document type definitions (DTDs):** DTDs are really not configuration data, but they are all located in the `WEB-INF/` directory, alongside `struts-config.xml` and `web.xml`. They are important files, though, because they are referenced by those configuration files. (In the sample, `struts-config.xml` references `struts-config_1_1.dtd`, and `web.xml` references `web-app_2_3.dtd`.)
- **Tag library descriptors (TLDs):** A TLD defines a tag library, associating tag names with Java classes to handle the tag's functionality. These are also pretty important; you'll find it tedious and tiresome to write any substantially large Struts application without using tag libraries somewhere along the way.
- **Properties files:** In the `WEB-INF/classes/` directory, you'll usually find at least one or two properties files. These are Java property files (name-value pairs, like `"title.login=Struts Online Banking - Account`

Login") that your application uses. The sample application has one for messaging (you might have more if you wanted to provide localized resources) and one for logging.

Again, it's worth saying that you don't need to be an expert in each of these areas; you might have used only a few of these items before, or you might be seeing each of them for the first time. As long as you have a basic understanding of how each file affects Struts -- and specifically, its configuration -- you'll be ready to get on with your major task: breaking up the configuration data into multiple files.

Section 4. From one to many

Now that you understand what's in your configuration files, it's time to look at reorganizing. In this section, you'll take your `struts-config.xml` (typically somewhere between 100 and 1000 lines for a medium-sized application) and split it into much more manageable files.

Organizing your configuration files

First, you need to decide whether you want to break your file up by *functionality* or by *typology*.

Breaking up a file by functionality means that you break the configuration file into multiple files, each of which handles some specific part of your application. For example, you might have one file that defines all data related to creating new accounts, and another that deals with logging users in.

Breaking up a file by typology means that you have a file for each type of data in your `struts-config.xml`. So you would have one file with all your `form-bean` elements, another with your `action-mapping` and `action` elements, and so on, right through the various elements that can appear in `struts-config.xml`.

Take at least a moment and think about which of these might be a better idea. All you Type-A personalities out there probably love the idea of breaking your elements into separate files. You need to add a new `Action`? No question about where it goes -- just drop it into `struts-actions.xml` (or whatever you call your action-related file). You'll never spend another moment wondering about where to put a new `global-forward`. However, that's an extremely bad idea!

To understand the problems with this approach, consider how you add functionality to a Struts application (the very words in this sentence should give you a clue as to

which is the better approach):

- You figure out what it is you're adding.
- You develop a form to take in the data you need (and come up with an appropriate `form-bean`).
- You define any `forwards` (or possibly `global-forwards`) that you need for the new form.
- You write an `action-mapping` to process the form (and add an `action` element or two).
- If needed, you define error messages that you might need in an `exception` element (or two).

These steps make it clear that to add a single piece of functionality, you almost always need to add *at least* two elements to `struts-config.xml` (a `form-bean` and an `action`). So if you did break your `struts-config.xml` into files based on typology -- where a file held only one type of element -- you'd need to crack open at least two files for new functionality. If you need to mess around with two or more configuration files just to add one piece of functionality, you don't gain any advantage. It also increases the possibility of error; you've got to keep up with a common naming schema for this functionality across multiple files, and if you change something in one file, you'll probably need to make similar changes in other files -- *not* a good thing!

A much better approach is to break your files up by functionality. Perhaps you put all authentication-related `actions`, `forwards`, `exceptions`, and so forth in a file you call `struts-authentication.xml`. If you add new authentication-related functionality (maybe you want to develop a "Forgot My Password" form), then you open that one file, and add everything you need to it. This keeps related functionality together, eliminates the need to remember a naming scheme across files, and is easily understood by even the most inexperienced programmers. This is the approach you'll take throughout the rest of this tutorial -- an approach I strongly recommended for *all* types of applications.

Splitting and naming your configuration files

So now that you have decided to break things up by functionality, it's *almost* time to take some action. But you need to do a little more preparation and planning. First, make a backup copy of your `struts-config.xml` file. You shouldn't take a chance on messing something up and not being able to easily back out your changes. While you're at it, make a backup of `web.xml` too.

With some backup insurance, your next step is to determine functionality that really

is global to the application. For example, you might have an `action` or `forward` related to errors that truly is used across your entire application; that is, all types of forms use this same bit of configuration to handle errors. Make a note of all the lines that seem to cut across the entire application.

See how many lines you've come up with. If you are stringent about making sure this is truly global data, you'll often end up with only 10 or 15 lines. If you have 20, 30, or more lines that you think are common to your entire application, take a second pass and see if you can narrow things down. Also, keep in mind that these are pieces of data that the entire application uses *directly*. For example, you might have flagged all the lines that relate to handling errors; perhaps you have several paths, a couple of actions, a resource bundle, and a database connection, all of which assist in dealing with errors. At first glance, it might seem like these are all common resources. However, most of them are not directly used by the application; in fact, you'll often only have a single `forward` that takes the user to the error page or form. The rest of those resources are then used to handle the error *after* the user has been redirected. So only the `forward` is really common; the rest of the lines can be moved into an error-handling-specific file. In this way, you really can cut things down to fewer than 20 lines.

Make another copy of your `struts-config.xml` (call it something like `struts-config-all.xml` if you like). Then, in your main `struts-config.xml`, remove all but the lines that you've flagged. The end result should be a very slim `struts-config.xml` file containing only application-wide data. In `struts-config-all.xml`, remove those lines. Now, in `struts-config.xml` *and* `struts-config-all.xml`, you have every line of configuration data for your application (with no lines duplicated).

Next, begin to break what's left in `struts-config-all.xml` into related pieces of functionality. Start with large pieces of functionality. Here are a few good suggestions for major groupings that could apply to any type of application:

- Authentication and security (also logging users in/out)
- Error handling
- Help/informational data

Beyond that, you'll of course have categories specific to your application. You might have a grouping related to managing users, adding accounts, placing orders, shipping items, or anything else. Use your best judgment, and keep these initial groupings fairly large. For each group, create a new file called `struts-functionality-grouping.xml`. So you might have `struts-authentication.xml`, `struts-error.xml`, `struts-shipping.xml`, and so on. It's a good idea to keep the `struts-` prefix so you can easily see which configuration files are related to Struts (and, as a bonus, directory listings will group these files together).

Next, do a line count for each file. If you did a fairly good job of choosing major

groupings, the files will often run between 30 and 300 lines each. (That's a wide variance, but it really depends on how your application is organized). If these files are really, really small (10 or 15 lines), you might want to merge some small groups together, assuming that makes sense. Keeping up with 50 files of 20 lines each isn't a lot better than keeping up with one 1000-line file; the idea here is to make things easier, not just different. On the other hand, if you still have files that are 300, 400, or 500 lines, consider repeating the process on just your very large files, breaking them into even smaller groups of functionality.

The goal is to find a nice balance between large files and large *numbers* of files. Ideally, you'll end up with 5 to 10 files, each with 50 to 100 lines of configuration data. Once you're to that point, you've got an application that you can actually manage -- and even let multiple groups work on. (The security guys aren't messing around with any files but `struts-authentication.xml`, the group writing help documentation is only working with `struts-help.xml`, and so on.) No matter how simple or complex your application, this is a *huge* improvement over one massive configuration file.

Referencing multiple configuration files

With your configuration data broken up into several files, you've completed the hardest part of your task. Now, you just need to let Struts know about all these various files.

Make sure the application isn't running while you do this. Open `web.xml` (which you've already backed up). You should see some lines that look like Listing 4, referencing `struts-config.xml`:

Listing 4. `web.xml` referencing `struts-config.xml`

```
<servlet>
  <servlet-name>banking</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet
    </servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

Listing 4 should look familiar to anyone who has worked with Struts for very long. With just a simple change, you can reference your additional configuration files, as shown in Listing 5:

Listing 5. `web.xml` referencing multiple configuration files

```
<servlet>
  <servlet-name>banking</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet
    </servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml,
                /WEB-INF/struts-authentication.xml,
                /WEB-INF/struts-help.xml
    </param-value>
  </init-param>

  <load-on-startup>1</load-on-startup>
</servlet>
```

Just add as many files as you have, all separated by commas. Don't forget to include the complete path to each file (relative to your Web application's root); other than that, this couldn't be easier. Restart your application, and you're all set.

Using resources from files other than the default struts-config.xml also couldn't be simpler. Internally, the Struts application assembles all of your configuration files into a single in-memory representation. Once your application is running, it has no "knowledge" of how many files you're using for your configuration data. You don't need to change a single line of code.

Adding configuration data

Another matter worth thinking about is how you *add* configuration data to this setup. You'll typically have one of three basic scenarios:

- You're adding a highly specific piece of functionality to an already established grouping. For example, you might add a new screen or link to your help system.
- You're adding a large set of functionality that is its own grouping. For example, you add an entire new, self-contained user-management system to your application.
- You're adding cross-cutting functionality to your application, and it affects your common data as well as one or more groupings. For example, you add a new error-handling system, affecting your global data (you have a new `forward` that all parts of the application use), your error-handling grouping, and your login grouping.

In the first case, you would just need to add the data to a single grouping-specific file, such as struts-help.xml. This is usually the easiest addition because you're working on a file that already exists, and you need to work with only one file.

In the second case, you still work with a single file, but it's a new file. So you might create struts-manageUsers.xml and enter all your new configuration data. However,

you then need to stop your application, open web.xml, and add the new configuration file. So while you're still making isolated changes to a single file, you must work with the web.xml file to effect the new changes.

In the third case, you must be extra careful. You need to change your struts-config.xml (for common data) and any additional files that are affected, such as struts-error.xml and struts-authentication.xml. You don't need to edit web.xml, but you need to restart your application anyway, so this type of change requires the most care.

Whatever type of change you make, resist the "fast fix" syndrome. You'll find it far too easy to simply crack open struts-config.xml and add a few lines (and then convince yourself that you've added application-wide data for the next several hours). Rather than make a 10-second decision that fouls up all your earlier hard work, be sure you know exactly where any new configuration data belongs -- and then put it there.

Section 5. Further cleanup

This section guides you through some additional steps you can take to organize your Struts configuration and streamline maintenance tasks.

Segregating configuration data

Any good servlet engine prevents users from accessing anything underneath the /WEB-INF/ directory tree in your Struts application. And, as you've seen, that's the choice location for all your configuration files. However, if you build up 8 or 10 configuration files and then add in web.xml, several tag library descriptors (TLDs), and a few other configuration files, this directory can become quite cluttered. If you have junior programmers digging around in this directory and working on various configuration files, the room for error increases. If this is a concern for you (or your boss), then you might want to think about creating a different directory for your struts-* configuration files, as shown in Listing 6:

Listing 6. Segregating configuration files

```
<servlet>
  <servlet-name>banking</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet
  </servlet-class>
  <init-param>
    <param-name>config</param-name>
```

```
<param-value>/WEB-INF/config/struts-config.xml ,  
                /WEB-INF/config/struts-authentication.xml ,  
                /WEB-INF/config/struts-help.xml  
</param-value>  
</init-param>  
  
<load-on-startup>1</load-on-startup>  
</servlet>
```

In some cases, you might like to create another directory off the root of your web application, such as `/config/`. However, servlet engines aren't set up to protect these directories automatically as they do the `/WEB-INF/` directory. You can certainly set things up in the servlet engine so that additional directories are protected, but why do the extra work? A much better alternative is to create a new subdirectory underneath `/WEB-INF/` for your files, such as `/WEB-INF/config`. Then, move all your `struts-*.xml` files into this subdirectory. Finally, update your `web.xml` file.

Creating this subdirectory isn't a mandatory step, but it's often useful for segregating your configuration data even further. Of course, using the same principles, you should consider putting your TLD files in their own directory, data configuration files in their own directories, and so on. Ultimately, you'll have a more complex directory structure, but everything is nicely organized and broken up and easy to both find and maintain.

Getting rid of tag library declarations

Once your configuration files are split into logical groupings, you've already dramatically decreased the maintenance for your application. However, you can still take several more easy steps to reduce your configuration complexity even further. First, consider the typical `web.xml` file and the numerous tag library declarations in that file. As an example, Listing 7 shows the last portion of the `web.xml` file from the Struts banking example used throughout this tutorial:

Listing 7. Tag library declarations in `web.xml`

```
<taglib>  
  <taglib-uri>/WEB-INF/struts-html.tld</taglib-uri>  
  <taglib-location>/WEB-INF/struts-html.tld</taglib-location>  
</taglib>  
  
<taglib>  
  <taglib-uri>/WEB-INF/struts-bean.tld</taglib-uri>  
  <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>  
</taglib>  
  
<taglib>  
  <taglib-uri>/WEB-INF/struts-logic.tld</taglib-uri>  
  <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>  
</taglib>
```

The problem here isn't quite as obvious as seeing hundred- or thousand-line

struts-config.xml files, but it's a problem nonetheless. It's fairly obvious that if you want to use another tag library -- the JSTL tag libraries, for example -- you must add another declaration or two to the web.xml file. That in itself isn't an immediate problem -- until you realize that web.xml makes or breaks your application. If you make a small typing mistake in that file or goof up the nesting, your entire application will probably stop working. It also probably means restarting your application (which in turn can mean extra testing, which takes time, and so on). Clearly, this isn't an ideal configuration situation.

So consider a very simple solution: Move your tag library declarations into a JavaServer Page (JSP). For example, you could put the three tag library declarations in Listing 7 into the JSP shown in Listing 8:

Listing 8. Tag library declarations in a JSP file

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-html"
  prefix="html" %>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-bean"
  prefix="bean" %>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-logic"
  prefix="logic" %>
```

Now, your JSPs just need to add an `include` directive to include this JSP file:

```
<!-- Include the standard tag library JSP -->
<%@ include file="/config/taglibs.jsp" %>
```

I usually put this file up at the top of my JSPs. Suddenly, I never need to mess with my web.xml file -- no application restarts for something as minor as adding a tag library, and my application is just that much easier to manage. I encourage you to make the same move and save yourself some headaches.

Using multiple message resource bundles

This is another Struts 101 configuration task. You'll find it in many Struts books, including the *Struts Cookbook* listed in [Resources](#). Still, it's probably in so many books because it's such a good idea.

You've already seen the `message-resources` element, used in the struts-config.xml file:

```
<message-resources parameter="BankingMessageResources" null="false"/>
```

However, it's not often recognized that you can use more than one of these elements in `struts-config.xml`. For example, this is a perfectly legal configuration fragment:

```
<message-resources parameter="BankingMessageResources" null="false"/>
<message-resources parameter="BankingHelpResources" key="help"/>
<message-resources parameter="BankingGuestUserResources" key="guest"/>
```

The first of these -- without a `key` attribute -- becomes the default resource set. So all your JSPs and servlets have access to that set of resources through the servlet context. However, often large portions of your application might need resources. For example, the preceding code example includes a set of resources just for the help portion of the application (messages, icons, errors, field labels, etc.), and another set specific to guest users (users who aren't logged in). Not much information is shared among the core application, the help portion of the system, and the guest user screens. So rather than mixing all of these resources into one file, you can split them into three separate files.

However, this works a bit differently from using three (or more) `struts-config.xml` files. In that case, the multiple files are essentially merged into one set of configuration data. In the case of multiple resource bundles, the files are *not* merged. Instead, your JSP pages must indicate a resource if they want to use the nondefault set (in this case, the `BankingMessageResources` bundle). To do this, you'll need to use the `message` tag, part of the `bean` tag library.

So, assume that:

- You've associated the `bean` tag library with the prefix `bean`.
- You've got a property in the `BankingHelpResources` bundle with the key `help.label.seeAlso`.

To reference this property, you would use the following line in your JSP:

```
<bean:message bundle="help" key="help.label.seeAlso" />
```

The additional attribute, `bundle`, lets you indicate a nondefault bundle to use, and the rest of the tag operates normally, just as you'd access a key in the default bundle.

You might also notice that even though there are multiple bundles, the example uses a bundle-specific prefix: the help-related resources are in `BankingHelpResources`, but the key is still prefixed with `help`, as in `help.label.seeAlso`. This might seem redundant at first -- why prefix the key with `help` when it's only available through the help-specific bundle? The answer is

all about documentation:

- It makes the purpose of the key that much clearer: no other programmer will wonder what the key is (whereas if the key were just `label.seeAlso`, it might not be as clear).
- If the name of the bundle itself became less self-documenting -- perhaps some junior programmer who hasn't read this tutorial changed it to `BankingExtraResources` -- then the label still helps identify the key's purpose.
- If you ever decide to roll in the resources from `BankingHelpResources` with another file, the keys still retain value and clarity.

You should always use clear naming, no matter *how* obvious you think the context or usage of the variable or key makes its purpose.

Section 6. Wrap-up

Summary

Even though I'm a programmer -- by trade and by choice -- I seem to spend more time configuring, setting up, deploying, and managing applications than I do actually coding. I don't particularly care for this (to be blunt, I don't like it one bit), but it's a fact of life. As a result, solutions like the one shown in this tutorial make my life easier. It takes a little more time to split up a Struts application's configuration files, but the results are well worth the cost. Instead of having a single point of failure -- whether that's one file, one programmer who understands that file, or even one server storing that file -- you break up the possible points of failure and reduce the chances of something seriously nasty going wrong. And that, as I said earlier, is worth at least a few extra nights of good sleep.

Downloads

- Demo: [Struts Online Banking application](#)

Resources

Learn

- [Struts User Guide: Installation](#): Learn how to install Struts on various servlet engines.
- [Struts User Guide: Configuring Applications](#): A guide to Struts configuration.
- "[Struts, an open-source MVC implementation](#)" (Malcolm Davis, developerWorks, February 2001): This article introduces Struts.
- "[Best practices for Struts development](#)" (Palaniyappan Thiagarajan and Pagadala Suresh, developerWorks, June 2004): Learn best practices you can follow to optimize your Struts applications.
- "[Using the Struts Validator](#)" (Brett McLaughlin, developerWorks, August 2005): This tutorial focuses on validation. It also covers Struts installation and setting up the Struts example application used in this tutorial.
- [WebSphere information center: Struts configuration files](#): Find out how IBM WebSphere and Struts play together in this guide to using Struts with WebSphere.
- [Programming Jakarta Struts, 2nd edition](#) by Chuck Cavaness and [Jakarta Struts Cookbook](#) by Bill Siggelkow: Two great books on Struts published by O'Reilly.
- [The Java technology zone](#): Hundreds of articles about every aspect of Java programming, including all the concepts covered in this tutorial.

Get products and technologies

- [Apache Software Foundations](#): Download Apache projects, and find out more about the Apache Software Foundation, at the Apache site.
- [Apache HTTP Server](#): Although I didn't talk about it in this tutorial, I always serve HTML pages with Apache and pass through Java requests to Tomcat.
- [Apache Struts](#): Struts, a top-level Apache project, is a well-documented open source offering.
- [Improve Struts Configuration File Editor for Eclipse](#): An Eclipse plug-in that lets you edit Struts configuration files in a graphical environment.

About the author

Brett McLaughlin, Sr.



Brett McLaughlin has worked in computers since the Logo days. (Remember the little triangle?) In recent years, he's become one of the best-known authors and programmers in the Java technology and XML communities. He's worked for Nextel Communications, implementing complex enterprise systems; at Lutris Technologies, actually writing application servers; and, most recently, at O'Reilly Media, Inc., where he continues to write and edit books that matter. His most recent book, [Java 1.5 Tiger: A Developer's Notebook](#), was the first book available on the newest version of Java technology, and his classic [Java and XML](#) remains one of the definitive works on using XML technologies in the Java language.