NetUI Overview

Table of contents

1. Introduction

NetUI is the piece of Beehive used to build the front-end of a web application. It contains two pieces: Page Flow and a powerful set of JSP tags.

2. Why Use NetUI?

Simply put, NetUI (Page Flow and a powerful set of JSP tags) helps you build a well-structured web application using a simple programming model.

First, because NetUI is an MVC framework (built on Apache Struts), it separates navigational control from presentation. This avoids:

- Limited reuse of navigational/flow logic.
- Cluttered, hard-to-maintain JSP source code.
- Difficulty in understanding the flow of an application.
- Unintended exposure of controller-logic code to team members who focus on other aspects of web development, such as content writers and visual designers.

Second, NetUI provides the Page Flow programming model which allows you to create *modular* "page flows" that can be inserted (and reused) inside of other flows. At root, it unifies the controller logic, state, and metadata for a piece of your application into a single class. On the View side, it offers a rich set of tags, such as the Tree and the Datagrid .

3. NetUI Features

NetUI makes building Java web applications easy and intuitive. When building applications with NetUI, the developer writes Java classes and pages --that's it. There is very little occasion to work with configuration files, or other components. NetUI also excels at separating presentation logic from data processing logic, resulting in uncluttered JSP code which is easy to understand and edit. Data processing and the web application configurables are handled in a single Java class using a simple declarative programming model.

Declarative Programming

Many common web app programming tasks are accomplished through a declarative programming model using "annotations", a new feature in Java 5. Annotations put configuration information (in general, "metadata") right alongside your code, alleviating the need for independent configuration files. Navigation, exception handling, validation, and other tasks are all defined in a single Java class: the Page Flow "controller" class that drives a piece of your web application.

Stateful Page Flows

When a user enters a page flow (by hitting an URL in the page flow's URL space), an instance of the page flow's controller class is created. While the user is in the page flow, the controller instance simply stores the flow-related state in member variables. Methods within the class -- particularly action methods and exception handlers -- have access to the member state. By default, the state is *automatically cleaned up* when the user leaves the page flow to enter another one. This behavior can be configured per-page flow, but auto-cleanup helps keep your session small, and focused on the task at hand.

Modular Page Flows

A single web application can have multiple page flows within it, allowing you to break up the application into separate, self-contained chunks of functionality. For an example, see the Petstore Sample , which has different page flows for browsing the Petstore, buying products, and handling user accounts.

Inheritance and Shared Flow

Page Flow inheritance is a powerful way to share actions, exception handlers, configuration, etc. among controller classes. It is normal Java inheritance, plus the ability to inherit/merge *annotations*.

Shared Flow provides an alternative way to make actions and exception handlers available to multiple page flows. The feature is useful for accessing shared state, for shared/templated user interface, and when you cannot change your controller class hierarchy.

Nested Page Flows

An entire page flow can be inserted, or "nested", inside of another page flow. At its heart, nesting is a way of pushing aside the current page flow temporarily and transferring control to another (nested) page flow with the intention of coming back to the original one. Nesting is useful when you want to do one of the following tasks:

- gather data from the user, for use in the current page flow;
- allow the user to correct errors or supply additional information en route to executing a desired action;
- show an alternate view of data represented in the current page flow;
- bring the user through a "wizard";
- show the user information that will be useful in the current page flow (e.g., help screens can be easily implemented as nested page flows); and
- in general, to further break up your application into separate (and in many cases reusable) pieces.

NetUI also offers special integration between nested page flows and popup windows.

Declarative Exception Handling and Validation

Exception handling and data validation are accomplished through a declarative programming model. The desired exception handling and validation behaviors are declared in the controller class (and additionally on form bean classes, for validation) alongside your Java code in the form of metadata annotations. This allows for single file editing and eliminates the need for separate configuration files.

Powerful JSP Tags

NetUI provides three tag libraries: (1) one library represents the core HTML tags, (2) another renders data grids and complex data sets as HTML, and (3) a third library provides page templating functionality.

The NetUI tags also support data binding (1) to JSP implicit objects (through the JSP 2.0 Expression Language) and (2) to other NetUI implicit objects. Note that many tags possess read-write access to these implicit objects.

First-class Integration with JavaServer Faces

NetUI has solid integration with JavaServer Faces. It treats JSF as a first-class view tier, where, for example, JSF components and command handlers can raise Page Flow actions, can databind to NetUI implicit objects, etc.

Struts Integration

Page Flow is built on top of Apache Struts 1.1. Each Page Flow controller is compiled into a Struts module. As a result, NetUI and Struts applications can work closely together.

Struts modules and page flows can co-habitate and interact with one another inside a web app. To forward from a page flow to a (pure) Struts module, simply reference the desired action within the Struts module. The same goes for the reverse direction: from a Struts module, simply configure an action to point to the desired method in the page flow.

You can also use the Struts merge feature to read configuration data from a pure Struts app into your Page Flow app's configuration files. Ordinarily, your Page Flow's configuration files are generated entirely from your application's JAVA source files (specifically from the *metadata annotations* that decorate the controller classes). But, in cases where you want to integrate a Struts module into your application, you can specify that the configuration files be generated from *both* the JAVA source files *and* the Struts module's configuration files, allowing you to change or add *any tag* in the generated configuration file. For example, suppose you want to override an action form's default scoping from request-scoping to

session-scoping. To do this, you simply create a Struts configuration file that overrides the appropriate parts of the Page Flow's configuration file, and then refer to this override file from within the Page Flow's JAVA source file (= the controller class) using a special annotation. In particular, you would specify the override file to state that such-and-such an action form should have session-scope rather then request-scope (so that the action form can now be shared with the Struts app).

4. The Logical Flow

Writing traditional web applications without a Page Flow controller class requires a fair amount of logic to be applied within the application's pages. For example, a site that provides a "My Page" functionality for logged in users would have to include logic on the home page to determine if the "My Page" link should take the user to the login form or directly to their customized page.

Using a page flow, the home page of the application would not link directly to either the login page **or** the user's "My Page" location, but rather would point back into Java code that makes the decision.

For the rest of this overview, the following **logical page flow** will be used:

logical page flow

This flow supports several routes from the home page of the application to the user's "My Page":

1. The user may directly navigate from `index.jsp` to `mypage.jsp` (by clicking a link), if the user is already logged in.
2. If the user is not already logged in, attempts to navigate from `index.jsp` to `mypage.jsp` will be intercepted and the user will be taken to the `login.jsp` instead. After successfully logging in, the user will be automatically taken to `mypage.jsp`
3. The user may directly navigate from `index.jsp` to `login.jsp` (by clicking a link). After logging in, the user will be automatically taken to `mypage.jsp` .

   In the event of a login failure, `login.jsp` will be redisplayed to give them another opportunity to authenticate themselves.
4. If the user desires to register with the site, he can click a link that will take him to `signup.jsp` . One signed up, the `thanks.jsp` will be displayed which offers a link to the `login.jsp` page.

5. The Implementation of the Flow: Controllers and Actions

In the above **logical flow** there are several *if* statements that cause the user flow to vary depending on their previous actions and other state.

- *If the user is not logged in...*
- *If the user is logged in...*
- *If the user's login attempt fails...*

NetUI moves this condition logic out of the JSPs and into a Java class that controls the movement through the application. This Java class is the **controller** portion of the **Model-View-Controller** (MVC) pattern. This allows a page to be written, for example, that appears to link directly from the home page of the application to the user's "My Page". The controller is given the opportunity to intercept the navigation between the two and redirect the user to the login page, if required.

Each of the interception points is an **action** of the particular controller class. Actions perform common application tasks. Here are some of the things that an action can do:

- navigate the user to a specified JSP
- perform conditional logic
- handle submitted data
- validate submitted data
- handle exceptions that arise in the application

Note that controller classes, and the actions they contain, are **URL addressable** . Hitting the following URL creates an instance of the controller class `foo.MyControllerClass` and runs its `begin` action. (When no other action is specified, the `begin` method is run by default.)

```
http://some.domain.com/foo/MyControllerClass.java
```

Hitting the following URL creates an instance of `foo.MyControllerClass` (if it doesn't already exist) and invokes the `someAction` action. Note that the controller class isn't mentioned by name: it's assumed that only one controller class exists in the directory, so there is only one candidate controller class to instantiate.

```
http://some.domain.com/foo/someAction.do
```

Note:

To make a Page Flow controller handle a default directory URL, e.g.,
http://some.domain.com/myApp/myPageFlow
you will need to add the page flow's URL to the `welcome-file-list` in `/WEB-INF/web.xml` :

```
<welcome-file-list>
  <welcome-file>Controller.jpf</welcome-file>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

This would cause the following URL to be hit for the above example:
http://some.domain.com/myApp/myPageFlow/Controller.jpf
On some servers (like Tomcat), you would also need to make sure that a *file* called `Controller.jpf` also exists in the web content under `/myPageFlow` , even though the class `myPageFlow.Controller` actually handles the request. (The file

can be blank.)

Actions may perform any required complex logic. For example, if a user clicks on the "My Page" link, the action may check if the user is logged in, and if so, navigate the user to the `mypage.jsp` page; otherwise it will navigate the user to the `login.jsp` page.

With normal HTML pages, each page is linked directly to other pages.

- **page > page > page > page**

When using page flows, pages and actions are interwoven, transparently.

- **page > action > page > action > page > action > page**

The above **logical page flow** can be redrawn with Page Flow controller actions in mind, as:

implementation page flow

Now it is apparent that to navigate from `index.jsp` to `mypage.jsp` , the user traverses across the `myPage` action. This action performs the necessary check to determine if the user has already been authenticated. If the user has logged in already, it will direct the user straight to `mypage.jsp` ; otherwise it will direct the user to `login.jsp` .

6. Next...

Next, learn about writing a **controller** class with actions.

- Controller Classes