Page Flow Controllers

Table of contents

---

1. Introduction

This topic explains the basics behind implementing **controller** files and **actions**. As introduced in the previous topic (NetUI Overview) the following web application schematic will be used.

implementation page flow

2. Starting the Controller Class

The first step to writing a controller class is to create a new basic class named `Controller.java`.

```
public class Controller
{
}
import org.apache.beehive.netui.pageflow.PageFlowController;

public class Controller
    extends PageFlowController
{
}
```

Additionally, Beehive weaves magic into controller classes using metadata annotations. The `@Jpf.Controller` annotation is a required marker on any NetUI controller class. The `@Jpf.Controller` annotation alerts the compiler that this class is a special Page Flow controller class, instead of a typical Java class.

```

import org.apache.beehive.netui.pageflow.PageFlowController;
import org.apache.beehive.netui.pageflow.annotations.Jpf;

@Jpf.Controller
public class Controller
    extends PageFlowController
{
}
```

Now we have the beginnings of a controller implementation.

3. Fleshing Out the Controller

Now that the boilerplate `Controller.java` is in place, we can begin to implement the actions that determine which JSP should actually be displayed. In the above model, there are 5 actions, plus one more action required by all Controller classes, the `begin` method. (Details about the `begin` method appear below.)

- `begin`

- `login`
- `myPage`
- `signUp`
- `processLogin`
- `processSignUp`

There are two basic ways to implement actions: you can implement an action either as a (1) ***simple action*** or as an (2) ***action method***.

Simple Actions are class-level annotations, that is, annotations that decorate the controller class. (You can also think of simple actions as *configurations* of the controller class. If you are familiar with Struts, it might help you to know that simple actions turn into `<action>` elements in the struts-config.xml file that is automatically generated when a controller class is compiled.) Syntactically they appear as follows:

```
@Jpf.Controller(
    simpleActions={
        @Jpf.SimpleAction( name="someName", path="somePage.jsp", [...other
properties...] )
    }
)
public class Controller
{
...
}
```

Simple actions can handle navigation, form submission, and form validation. If that is all your action needs to accomplish, you should implement the action as a simple action. What simple actions *can't* do is handle decision logic. If your action needs to make a decision and conditionally execute code based on that decision, you should implement the action as an *action method*.

Action Methods are Java methods that have been endowed with all of the magic of actions: that is, they can navigate users around the page flow, handle form submissions, validate form data, handle decision logic, etc. (You can also think of the action methods as *configurations* of individual methods, in contrast to simple actions, which configure the entire class. Again, if you are familiar with Struts, know that action methods, just like simple actions, are complied as `<action>` elements in the struts-config.xml file.) Syntactically speaking, an action method is a Java method that (1) returns the type `Forward` and (2) is decorated with the `@Jpf.Action` annotation:

```
@Jpf.Action(
        forwards = {
            @Jpf.Forward( name="someName", path="somePath.jsp", [...other
properties...] )
        }
    )
    public Forward someMethod()
    {
        ...
    }
```

3.1. Simple Actions

Three of our five actions are purely navigational, and, as such, implementable as simple actions. Those actions are `begin`, `login`, and `signUp`. The remaining actions require object oriented programming, so they will be implemented as action methods.

The simple action implementations appear below. The following `@Jpf.SimpleAction` annotations define a set of mappings between action names and JSP destinations. When a particular action is invoked, the user is carried to the corresponding JSP.

> **Note:**
>
> Each Controller class requires a simple action or action method named `begin`--without it the class will not compile. The begin action functions as the entry-point into the page flow. In this case the begin action simply navigates the user to the `index.jsp` page.

```
import org.apache.beehive.netui.pageflow.PageFlowController;
import org.apache.beehive.netui.pageflow.annotations.Jpf;

@Jpf.Controller(
    simpleActions={
        @Jpf.SimpleAction(name="begin", path="index.jsp"),
        @Jpf.SimpleAction(name="login", path="login.jsp"),
        @Jpf.SimpleAction(name="signUp", path="signup.jsp"),
    }
)
public class Controller
    extends PageFlowController
{
}
```

3.2. Action Methods

Now it is time to re-implement the three action methods: `login`, `processLogin`, and `processSignUp`.

The `myPage` action must determine if the user has already authenicated himself or not and the action must behave differently depending on the result of that determination. If the user has already been authenticated, then the page `myPage.jsp` will be displayed; if the user has not been authenticated yet, then the page `login.jsp` will be displayed.

We will implement this behavior in two steps: (1) first will implement a **rudimentary action method**, (2) second we will add the **conditional navigational behavior** to the method.

3.2.1. Rudimentary Action Methods: Constant Forwards

An action method must have two features: (1) it must the type `Forward` and (2) must be decorated with the `@Jpf.Action` annotation.

The first step in the re-implementation process is to remove the simple action named `mypage` and replace it with a method named `myPage()`. By returning a `Forward` object, the method indicates which page to display to the user.

```
import org.apache.beehive.netui.pageflow.Forward;
import org.apache.beehive.netui.pageflow.PageFlowController;
import org.apache.beehive.netui.pageflow.annotations.Jpf;

@Jpf.Controller(
    simpleActions={
        @Jpf.SimpleAction(name="begin", path="index.jsp"),
        @Jpf.SimpleAction(name="login", path="login.jsp"),
        @Jpf.SimpleAction(name="signUp", path="signup.jsp"),
        @Jpf.SimpleAction(name="processLogin", path="mypage.jsp"),
        @Jpf.SimpleAction(name="processSignUp", path="thanks.jsp")
    }
)
public class Controller
    extends PageFlowController
{

    public Forward myPage()
    {
        ...
    }

}
```

To help with configuration and to avoid having JSP names within the body of a controller method, Beehive once again uses annotations. The `Jpf.Action` and `Jpf.Forward` annotations are used on each action method to build a **mapping** between **forward names** and **JSPs**. The method then works only in terms of the forward name, and doesn't directly refer to the JSP path.

The general form the of `Jpf.Action`/`Jpf.Forward` annotations are:

```

@Jpf.Action(
  forwards = {
    @Jpf.Forward( name="...", path="..." ),
    @Jpf.Forward( name="...", path="..." ),
    @Jpf.Forward( name="...", path="..." )
  }
)
```

By convention, forward names such as **success** and **failure** are used, but by no means are required. It is good practice, though, to avoid naming the forward based upon the JSP name since doing so would remove some of the decoupling that Beehive applications attempt to achieve.

```
import org.apache.beehive.netui.pageflow.Forward;
import org.apache.beehive.netui.pageflow.PageFlowController;
import org.apache.beehive.netui.pageflow.annotations.Jpf;

@Jpf.Controller(
   ...
)
public class Controller
    extends PageFlowController
{
    @Jpf.Action(
        forwards = {
            @Jpf.Forward( name="success", path="mypage.jsp" )
        }
    )
    public Forward myPage()
    {
        ...
    }
}
```

All that is left is a `return` statement to return the appropriate `Forward` object. This is accomplished simply by constructing a new `Forward` with the appropriate name.

```
import org.apache.beehive.netui.pageflow.Forward;
import org.apache.beehive.netui.pageflow.PageFlowController;
import org.apache.beehive.netui.pageflow.annotations.Jpf;

@Jpf.Controller(
   ...
)
public class Controller
    extends PageFlowController
{
    @Jpf.Action(
        forwards = {
            @Jpf.Forward( name="success", path="mypage.jsp" )
        }
    )
    public Forward myPage()
    {
        return new Forward( "success" );
    }
}
```

Now we have re-implemented one of our simple actions as an action method. However, our new action method doesn't do anything more than the original simple action. The new action method remains a purely navigational action: it is not yet capable of any decision logic and conditional execution. In the next section we will endow the action method with conditional navigational behavior.

3.2.2. Advanced Action Methods: Conditional Forwards

The first step in adding conditional navigational behavior is to define *two* forwards named **authenticated** and **not_authenticated**, which are mapped to `mypage.jsp` and `login.do` respectively.

```
import org.apache.beehive.netui.pageflow.Forward;
import org.apache.beehive.netui.pageflow.PageFlowController;
import org.apache.beehive.netui.pageflow.annotations.Jpf;

@Jpf.Controller(
    ...
)
public class Controller
    extends PageFlowController
{
    @Jpf.Action(
        forwards = {
            @Jpf.Forward( name="authenticated",     path="mypage.jsp" ),
            @Jpf.Forward( name="not_authenticated", path="login.do" )
        }
    )
    public Forward myPage()
    {
        ...
    }
}
```

But how does the method decide which forward to invoke? In this case, the determination of authentication is performed by checking a **session attribute** to see if the `authenticated_user` attribute has been set.

```
import org.apache.beehive.netui.pageflow.Forward;
import org.apache.beehive.netui.pageflow.PageFlowController;
import org.apache.beehive.netui.pageflow.annotations.Jpf;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

@Jpf.Controller(
    ...
)
public class Controller
    extends PageFlowController
{
    @Jpf.Action(
        forwards = {
            @Jpf.Forward( name="authenticated",     path="mypage.jsp" ),
            @Jpf.Forward( name="not_authenticated", path="login.do" )
        }
    )
    public Forward myPage()
    {
        HttpServletRequest request = getRequest();
        HttpSession session = request.getSession();

        if ( session.getAttribute( "authenticated_user" ) != null )
        {
            return new Forward( "authenticated" );
        }

        return new Forward( "not_authenticated" );
    }
}
```

Now that we have a method with two possible navigation outcomes, the flow diagram appears as follows. Notice the two named arrows exiting the `myPage()` method.

conditional forwards

You may notice that the body of `myPage()` has no particular logic regarding the JSP "myPage.jsp" itself. It simply operates in terms of authentication and generically named `Forward` objects. This presents a possibility of sharing this logic with other controller methods that are concerned with authentication. .

3.3. Handling Forms

Handling form data works similar to other controller methods. By providing a parameter to the controller method the HTML form data is made available to the controller method. In the above model, controller methods that process forms have been named with the `processXXX(..)` convention.

- `processLogin(...)`
- `processSignUp(...)`

First, define a JavaBean to represent the HTML form to be submitted. This JavaBean can be of any Java type, as long as it conforms to standard JavaBean syntax.

The JavaBean may be defined (1) as a `static` inner class of the controller itself (see example below) or (2) as a stand-alone Java class in a separate file. The JavaBean class follows normal JavaBean conventions and requires no special annotations.

```

import org.apache.beehive.netui.pageflow.Forward;
import org.apache.beehive.netui.pageflow.PageFlowController;
import org.apache.beehive.netui.pageflow.annotations.Jpf;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

@Jpf.Controller
public class Controller
    extends PageFlowController
{
    ...
    ...

    public static class LoginForm implements java.io.Serializable
    {
        private String username;
        private String password;

        public void setUsername(String username)
        {
            this.username = username;
        }

        public String getUsername()
        {
            return this.username;
        }

        public void setPassword(String password)
        {
            this.password = password;
        }

        public String getPassword()
        {
            return this.password;
        }
    }
}
```

Defining the `processLogin(...)` method to take a `LoginForm` parameter is all that is required to have a controller method that can operate upon the submitted form.

```
import org.apache.beehive.netui.pageflow.Forward;
import org.apache.beehive.netui.pageflow.PageFlowController;
import org.apache.beehive.netui.pageflow.annotations.Jpf;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

@Jpf.Controller
public class Controller
    extends PageFlowController
{
    ...
    ...
    public Forward processLogin(LoginForm form)
    {
        ...
    }


    public static class LoginForm
    {
        ...
        ...
    }
}
```

Once again, `processLogin(...)` is a conditional forward controller method. If a user has entered a correct username and password, then they should be directed to `mypage.jsp`, otherwise they will be returned back to the `login.jsp` for another attempt. Checking username and password is outside of the scope of Page Flow, and in this example, we rely upon a mythical `MyAppUtils` class to perform this logic.

```
import org.apache.beehive.netui.pageflow.Forward;
import org.apache.beehive.netui.pageflow.PageFlowController;
import org.apache.beehive.netui.pageflow.annotations.Jpf;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

@Jpf.Controller
public class Controller
    extends PageFlowController
{
    ...
    ...

    @Jpf.Action(
        forwards = {
            @Jpf.Forward( name="login_success",     path="mypage.jsp" ),
            @Jpf.Forward( name="login_failure", path="login.jsp" )
        }
    )
    public Forward processLogin(LoginForm form)
    {
        if ( MyAppUtils.authenticate( form.getUsername(),
                                      form.getPassword() ) )
        {
            HttpServletRequest request = getRequest();
            HttpSession        session = request.getSession();

            session.setAttribute( "authenticated_user",
                                  form.getUsername() );

            return new Forward( "login_success" );
        }
```

```
        return new Forward( "login_failure" );
    }
}
```

Having fleshed out the `processLogin()` action method, the diagram appears as follows.

implementation page flow

Similar implementation would be done for `processSignUp(...)`, involving another form class such as `SignUpForm`.

3.4. Handling Exceptions

Suppose a new user completes the signup form and submits her user profile. But when the profile is processed, it is discovered that the username has already been taken by another user. What then?

A natural design choice would be to have the `processSignUp` action throw an exception and then have the controller class handle the exception by returning the user to the original signup page. The following diagram shows how you can interweave exception handling into the page flow to further refine the paths through the flow.

page flow exception handling

You can implement exception handling using the `@Jpf.Catch` and `@Jpf.ExceptionHandler` annotations. The `@Jpf.Catch` defines some exception to handle should it arise within the controller class. `@Jpf.ExceptionHandler` annotation is used to define a dedicated method for handling the exception.

```
@Jpf.Controller(
    catches={
        @Jpf.Catch(type=AccountAlreadyExistsException.class,
method="handleAccountAlreadyExistsException")
    },
    simpleActions={
        ...
    }
)
public class Controller
    extends PageFlowController
{
    ...
    ...

    @Jpf.ExceptionHandler(
        forwards={
            @Jpf.Forward(name="signup", path="signup.jsp")
        }
    )
    protected Forward
handleAccountAlreadyExistsException(AccountAlreadyExistsException ex,
String actionName, String message, Object form)
    {
        return new Forward("signup");
    }

}
```

To protect a method with this error handling system, you only need to specify that the method throws the appropriate sort of exception, in this case, `AccountAlreadyExistsException`.

```
@Jpf.Controller(
    catches={
        @Jpf.Catch(method="handleAccountAlreadyExistsException",
type=AccountAlreadyExistsException.class)
    },
    simpleActions={
        ...
    }
)
public class Controller
    extends PageFlowController
{
    ...
    ...

    public Forward processSignUp(SignUpForm form)
            throws AccountAlreadyExistsException
    {
        ...
    }

    @Jpf.ExceptionHandler(
        forwards={
            @Jpf.Forward(name="signup", path="signup.jsp")
        }
    )
    protected Forward
handleAccountAlreadyExistsException(AccountAlreadyExistsException ex,
String actionName, String message, Object form)
    {
        return new Forward("signup");
    }

}
```

4. Form Validation

For details on form validation see the topic Data Validation

5. Next...

Next, learn about linking this controller class to the JSPs to allow for the interception to occur.

- JSP Files