CHAPTER 9

State Navigation Pattern

Intent

The State Navigation pattern provides an infrastructure in which HTML content can be navigated, and the state is preserved when navigating from one piece of content to another.

Motivation

Web applications have major state and consistency problems, and some Ajax applications amplify those problems. To illustrate, let's go through the process of buying a plane ticket and note the problems.

I fly regularly for business and as such am always looking for the best price. Because of my ticket-searching capabilities, I have become the travel agent for my wife and a few other people. I search many travel sites and use the permutations and combinations strategy to find the cheapest or most convenient ticket. If I find a ticket, I then try to find a better ticket by moving the dates forward or back, switching to a different airport, or even switching travel sites. Trying to find the best ticket by using a web browser has its challenges because of the way many travel websites "remember" the ticket information. The main problem is that travel websites may not remember my original flying times when I click the Back button, or may not remember old flight information when I open a second browser. And worse yet, some sites perform redirections to other sites not asked for, or require you to fly from certain airports. All travel sites at the time of this writing have one problem or another.

Consider Figure 9-1, which is a snapshot from one travel site. I have blanked out the travel site details because the figure is used to illustrate a problem. Figure 9-1 is the result of selecting the starting and ending points of the initial leg of a flight and being offered a set of times and conditions to choose from.

Figure 9-1 shows found flight details, and the user needs to choose one flight before continuing. To continue and to select the return leg of the flight, the user clicks Rueckflug. To go back and start again, the user clicks Zurueck. For interest sake, let's click Zurueck and see what happens. Figure 9-2 shows the resulting HTML page.

Figure 9-1. *Found flight details*

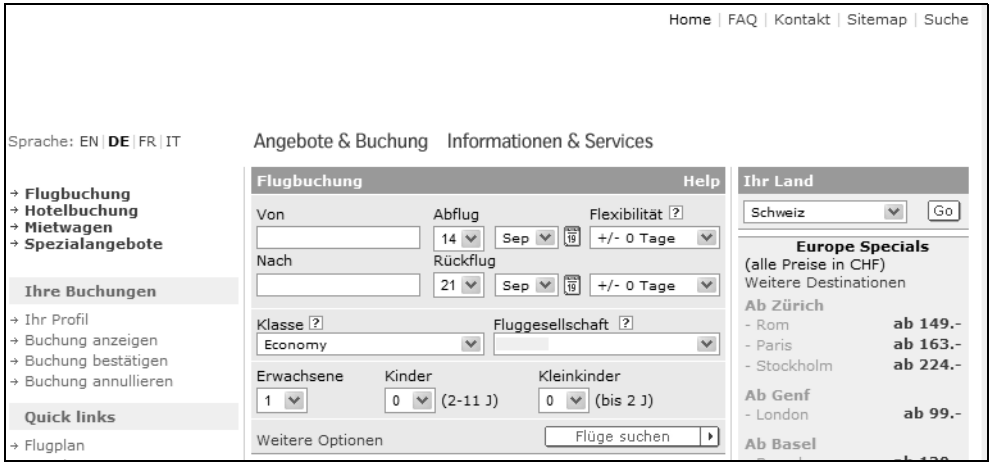

Figure 9-2. *Initial search page*

Figure 9-2 shows that by going back to start a search, you reset all of your search parameters and have to start from scratch again. This is irritating. The process becomes even more confusing if the user decides to use the navigation buttons and clicks the Back button of the web browser. Figure 9-3 illustrates what happens if the Back and Forward web browser buttons or a combination of those is pressed.

Figure 9-3. *Error that results from pressing the wrong web browser buttons*

Figure 9-3 illustrates how clicking the web browser Back button when pages have been posted using the HTTP `POST` method can cause the browser to generate errors and dialog boxes asking for further help. In these situations, depending on your actions, either the web application works or you bought a second plane ticket. The user experience is inconsistent and problematic.

To solve these experience inconsistencies, web browsers "remember" what the user entered. The web browser solution involves remembering the state contained within the form elements of the HTML page. This remembering of state causes the web browser to automatically fill in logins or form details without requiring the user to type everything in yet again. Yet the web browser solution does not always work because the state is not always consistently remembered. We can't blame the web browsers for not remembering a correct state because it is not the fault of the web browser. The web browser is just trying to make the best of a bad situation that many web application developers and web application frameworks have forced upon us.

Where heck broke out was when software vendors decided to fix the inherent statelessness of the HTTP protocol. These fixes very often conflict with the functionality of the web browser and introduce their own navigation paradigm. One of the reasons we deviated from the original intention of the Web and the HTTP protocol is our desire to improve and mold things to our liking. Put all of these factors to work and you have the reasons why page navigation is broken.

Applicability

The State Navigation pattern applies in all of those contexts where editable state is associated with an HTML page(s). In most cases, that means workflow or business process operations. It does not mean that only HTML forms that create a workflow are applicable. The state must not

be editable in the HTML representation form, because the Representation Morphing pattern can be applied to convert a static representation into an editable representation.

Not all states are created equally. There is binding state and nonbinding state. *Binding state* is the focus in this pattern. *Nonbinding state* is used to represent the binding state. For example, the nonbinding state associated with a mailbox could indicate how to sort the e-mails that are displayed in the mailbox. The resource that represents the e-mails is a listing and is considered a binding state. Nonbinding state can be lost without ramifications to the binding state. If the sorting order were to be lost, the resulting e-mails would have another ordering but no information would be lost. At the worst, the end user is inconvenienced.

Associated Patterns

The State Navigation pattern uses the materials presented in Chapter 2 that define the `Asynchronous` type. When the State Navigation pattern is implemented, it is assumed that the Permutations pattern is used. The State Navigation pattern does assume the state is defined as a chunk, as defined by the Content Chunking pattern, and that the state of the HTML page uses the Representation Morphing pattern. Where the State Navigation and Content Chunking pattern deviate is that a State Navigation implementation is a single chunk only.

Architecture

When implementing the State Navigation pattern, the primary focus is to manage the state associated with an HTML page. The State Navigation pattern infrastructure is responsible only for serving and receiving the state content. The HTML page is responsible for processing and generating the state. The HTML page is in control of calling the State Navigation pattern infrastructure. If the HTML page does not implement any calls to the State Navigation pattern infrastructure, there is no state managed. The State Navigation pattern infrastructure is defined on a per-HTML page basis. This makes it possible for a web application to have mixed state, where some pages are associated with a state, and other pages are not.

Moving Toward an Ideal Solution from the User's Perspective

It would be simple to say, "This is the way that the State Navigation pattern is designed regardless of how the HTTP or HTML infrastructure functions." However, that is not possible because the HTTP and HTML infrastructure has its own rules. Therefore, the State Navigation pattern solution is entirely dependent on what works best with HTML and HTTP.

Before I illustrate a solution at the technical level, I am going to show you the desired solution from a user's perspective that involves manipulating and navigating HTML pages in a simplistic and fictitious workflow application. Figure 9-4 shows the first HTML page of the workflow application.

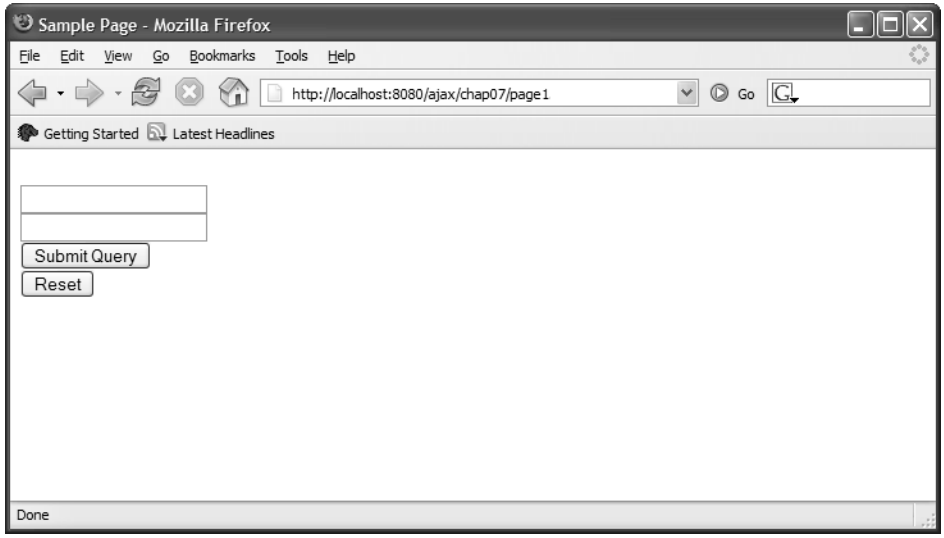

Figure 9-4. *Initial HTML page*

In Figure 9-4, the HTML page has two text boxes and two buttons and is called an *HTML form*. Implementing the Permutations pattern by using an HTML form is a challenge from a resource definition perspective. Contrast the HTML form to a traditional HTML page. When Mary Jane (for example) reads an e-mail entry, the Permutations pattern is applied on the contents of the e-mail entry. The resource is the URL `http://mydomain.com/mailbox/maryjane/entry1234`, and the representation is a transformation of the e-mail entry to content that the client wants. The resource and representation of the e-mail entry cannot be edited or modified.

When the Permutations pattern is applied to an HTML form, the HTML form without any content is the representation of a resource. The URL `http://mydomain.com/resource/step1` is an example resource that maps to the HTML form `http://mydomain.com/resource/step1.html`. The end user fills in the form, and a state is created. If the end user were to press the Back button and then the Forward button on the web browser, the created state would be lost (to avoid losing the state, most browsers "remember" the HTML form element contents). The problem is that the resource URL is for an empty HTML form representation. To remember the state, another URL (for example, `http://mydomain.com/resource/step1/state/1234`) needs to be defined. Defining a new URL is not a problem, and this topic is covered shortly. What needs to be noted is that editable HTML pages have multiple resource URLs that are dependent on the state.

So the problem in Figure 9-4 is how to associate a state with the resource (the example illustrated uses a URL). The association cannot be embedded in an HTTP header or HTML text field or cookie because that would violate the intent of the State Navigation pattern and the general design of URLs. But even simpler, when a URL is copied and pasted into another browser, there is no chance of copying and pasting a cookie, HTTP header, or HTML text field. Therefore, the state or reference to the state must be in the URL.

The solution of associating a state with a resource is to use a special URL, but one that is specialized by using a state identifier. Figure 9-5 illustrates an example HTML page that has been specialized by using a state identifier. For reference purposes, the state was loaded by the `XMLHttpRequest` object using the Content Chunking pattern.

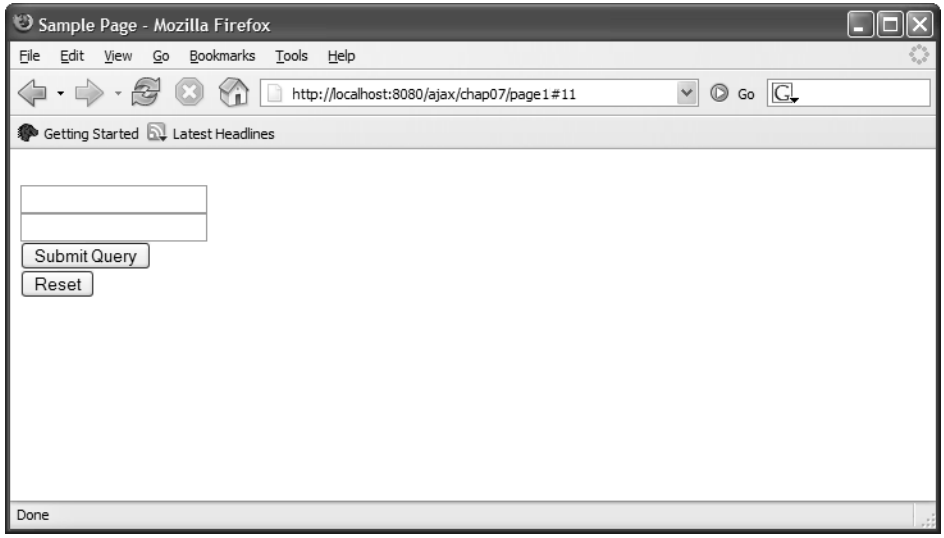

Figure 9-5. *Initial HTML page with session state loaded*

In Figure 9-5, the HTML page has loaded the associated session state. Comparing Figure 9-4 and Figure 9-5, you would not be able to tell because almost nothing has changed. The one change is the appending of the text `#11`, which is used to indicate the state identifier.

When a URL is associated with a hash character, the browser considers the new URL as unique but does not force a refresh of the browser. For example, if the URL were `http://mydomain.com/ajax/chap07/page1/state/11`, the HTML page would have been reloaded and the server would have been responsible for associating the HTML form state with the HTML page. The problem with changing the URL to associate a state is that it forces a reload for each and every state. A redirection occurs, and the web application is complicated and potentially prone to fail or load the wrong content. Using the # character does not require a reloading of the HTML page. The # character is ideally suited to be used in the context of the Content Chunking pattern because a script can parse the URL and extract the state identifier. The script then uses the `XMLHttpRequest` type to retrieve the state from the server and saves it in the HTML page. From the perspective of the server, there exists only the HTML form with no content, and the state. The client-side script associates the state with the HTML form. Do remember that state does not have to be associated with an HTML form, but could be associated with an HTML page that is transformed by using the Representation Morphing pattern.

Going back to Figure 9-5, if the user were to enter some data in the text boxes and then click the Submit Query button, the HTML page in Figure 9-6 would result.

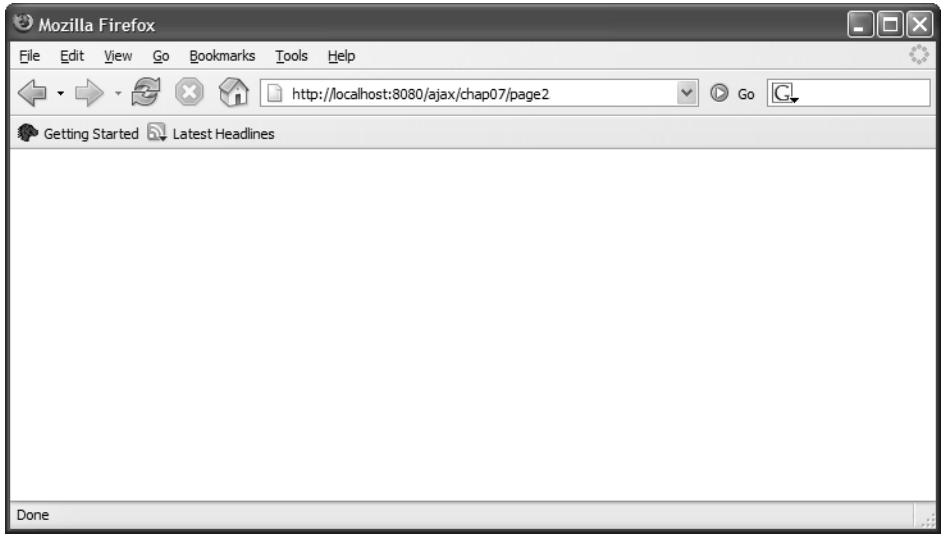

Figure 9-6. *Second page resulting from processed page*

Figure 9-6 is plain vanilla because there is nothing to display. This figure illustrates the process of saving state and loading another HTML page that does not have any state. What happened is that the script saved the state of the previous page and posted the state to the server. After the state was successfully posted, the next HTML page was retrieved by using standard navigation techniques. Separating the posting of the state and retrieving of the next appropriate content solves many problems, with one being the unnecessary dialog box illustrated in Figure 9-3. When the user clicks the Back button, the state in Figure 9-7 is displayed, illustrating the State Navigation pattern.

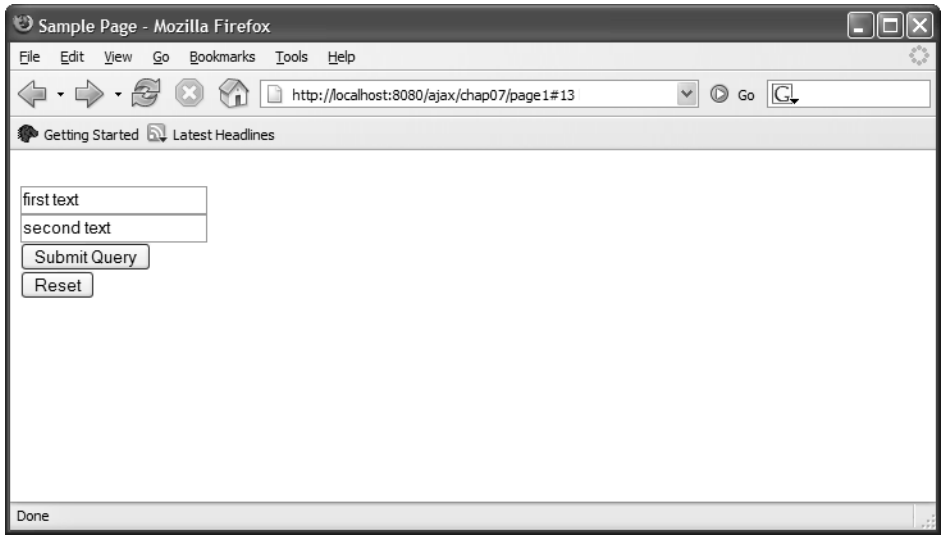

Figure 9-7. *Original page1 is reloaded with last known state*

Looking closer at Figure 9-7, you can see several changes, namely the text boxes are filled with some text, which is the state, and the state identifier has changed to `13`, from `11`. At this point, you could click the Forward and Back buttons as many times as you wanted; the state would be constantly reloaded, while the state identifier would update itself.

Extending the Solution for a Web Application

The presented solution illustrates how the hash character can be used to load and save the state of an individual HTML page. The individual HTML page or resource saves state on the server side that can be combined into a series of HTML pages that are related to each other. When stringing together multiple HTML pages that are related, problems can occur with a web browser, and two problems are pronounced. The first problem relates to multiple browsers attempting to access the same content that has a state. When this situation occurs, the state associated with the content is dependent on the timing of what content is submitted by one of the browsers. The second problem relates to navigating content by using the Forward and Back buttons, as that can wreak havoc on content that has an associated state.

To illustrate the problem of multiple browsers displaying the same content, let's go through an example. The example will navigate through the resources `/resource`, `/resource2`, and `/resource3`. If users navigated the resources individually, or in an ad-hoc fashion, the web application would have no idea how to manage the state because there is no order to the navigation. To provide order, the resources are strung together by using cookies that manage which resources are called and the state of the called resources. But the HTTP cookies give a false sense of security. The problem is that HTTP cookies do not distinguish between different web browser instances. It does not mean that cookies cannot be used, but they cannot be the reference point used by the server to manage the state of the resources. However, a solution using cookies will not be illustrated because the focus is on the page transitions and managing of the state.

Getting back to the problem of being unable to distinguish between a browser window instance, that problem is best illustrated in Figure 9-8.

In Figure 9-8, the initial browser loaded the URL `/resource`, and some content was generated. After the user fills in the HTML form and clicks Submit Query, the state of the URL `/resource` is saved, and `/resource2` is loaded. Having loaded `/resource2`, the user decides to open another browser and copies the URL `/resource2` to the second browser that is loaded. At this moment, there are two browser instances that loaded the same content, and both browsers reference the same cookie identifier. Where the server becomes confused is when the user switches back to the first browser, fills in the data, and clicks the Submit Query button, causing `/resource3` to be loaded.

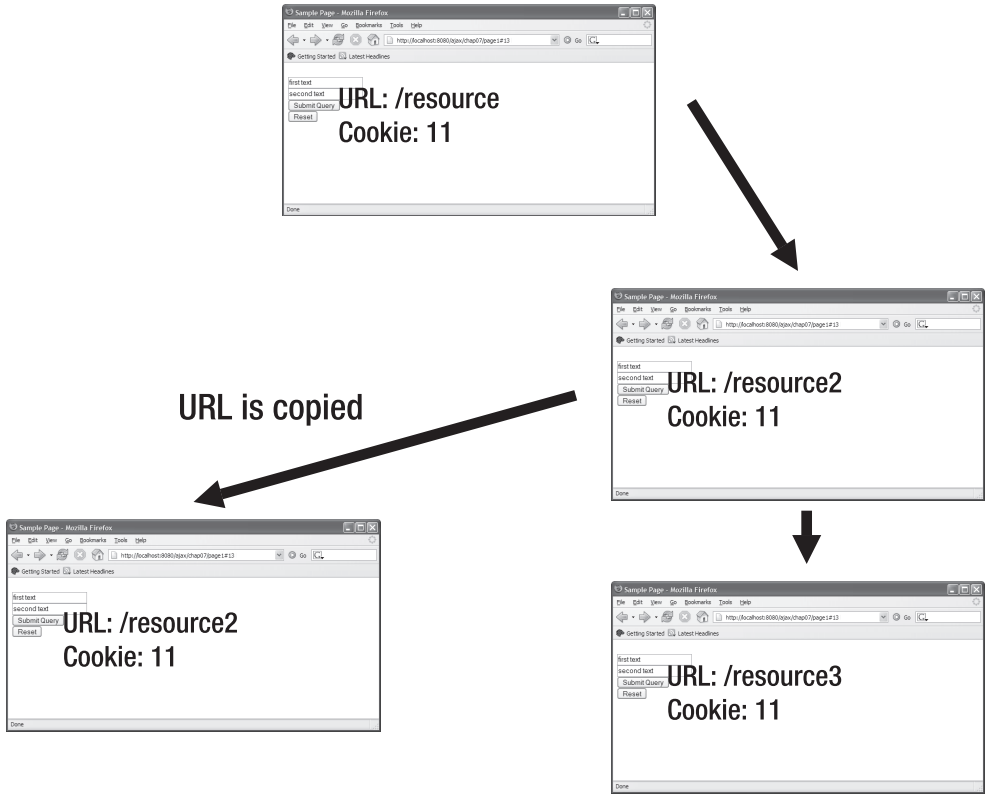

Figure 9-8. *Processing multiple pages in a web application by using cookies*

The confusion with an HTTP cookie occurs when the first web browser loads the representation associated with the resource `/resource3`, while the second web browser loads the resource `/resource2`. If the second browser attempts to navigate to the resource `/resource3`, the server will become confused as to what stage the web browser is really at. The server cannot distinguish between browser instances, and therefore overwrites new data over old, or old data over new, causing consistency problems. The behavior of the cookie is correct, as the cookie specification explicitly says that a cookie is associated with a domain and not a browser instance.

Using a state identifier to manage the state and resources creates a solution in which the state is accumulated. Accumulation of state makes it possible to fork the state if multiple browsers are accessing multiple versions of the state. To understand the logic, consider Figure 9-9, which indicates how unique state identifiers are created.

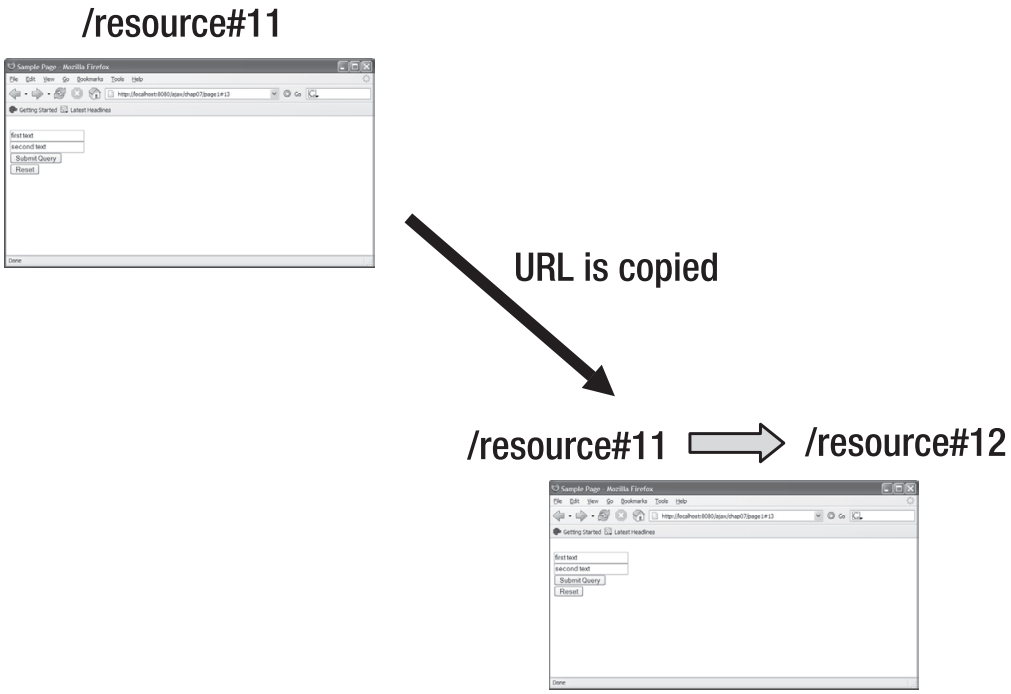

Figure 9-9. *An example of the state identifier being updated after the URL is copied to another web browser instance*

In the upper-left corner of Figure 9-9 is a web browser that has downloaded some content with the URL `/resource#11`. The loaded content is the resource `/resource` with the state identifier `11`. Imagine the user opening a second browser and copying the link `/resource#11`. The State Navigation pattern will load the resource `/resource` and the state associated with the identifier `11`. In the second browser, the state identifier is updated to reference `12`. If the second browser has the same state identifier as the first, that binds both browsers to the same state and creates concurrency problems. Imagine that the client modifies the state in the first browser; then the second browser would see the same state. This is not desirable, and therefore the state identifier `11` is copied to a new state identifier `12`. Then the first and second browser instances for the time being have the same state values, but different references.

The solution in Figure 9-9 needs one additional twist to make it work properly. If the browser were to request the URLs `/resource#11` and `/resource#12`, the resource `/resource` would be issued twice. This relates back to the purpose of the hash character, which is a reference to a link on an HTML page. This is a good thing, because the State Navigation pattern has separated the resource from the state of the resource. So when the resource `/resource#11` is called, the URLs `/resource` (for example, HTML page) and `/resource/state` (for example, HTML page state) are called.

By using the `XMLHttpRequest` object, it is easy to separate the two URL requests, and there are multiple ways to implement the two URLs. But using two URLs is not enough. You also need to use HTTP headers to uniquely identify the request. Figure 9-10 illustrates how Figure 9-8 is fixed by using HTTP headers.

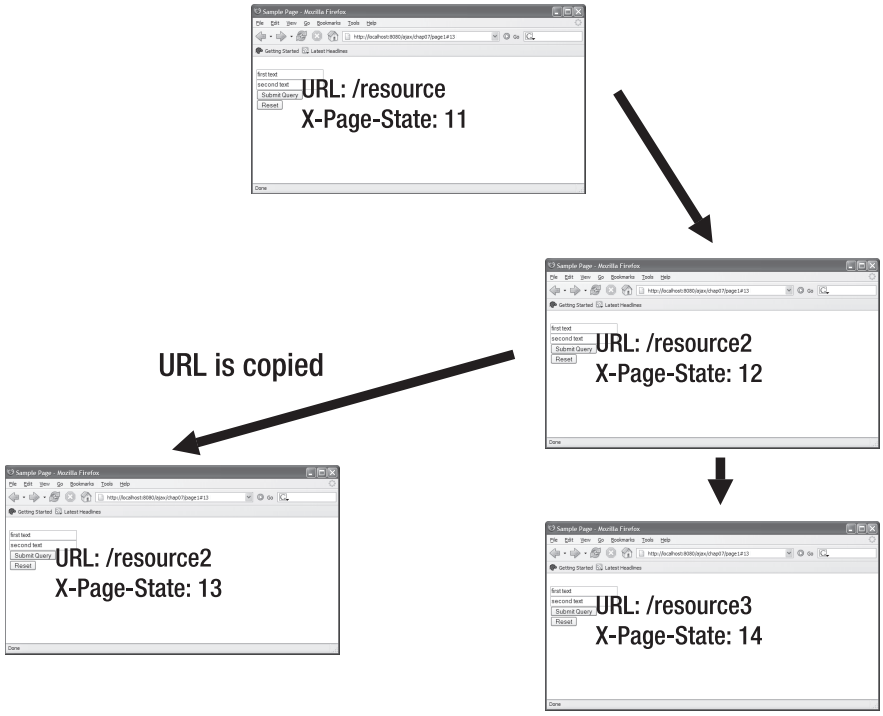

Figure 9-10. *Rearchitecting multiple resources to use a state identifier*

In Figure 9-10, each of the web browser instances is a URL, with a hash code–identified state identifier that is converted into an `X-Page-State` HTTP header. Each instance of the web browser has an HTTP header that is unique. This is a good thing because now, even though there are two browsers, the resource `/resource2` has two separate instances of the associated state.

Having the unique state identifiers works, except it exposes another problem: there is no stringing together of the individual HTML pages to build a web application. What we don't know is how the states relate to each other. Visually, we know that the states `11`, `12`, and `14` are a single chain. And visually we know that `11` and `13` are another chain. But the server does not know that because the server does not know that state `13` is the result of opening a second browser.

To finish the solution, the history of the URLs is needed. The web browser has that information because it is required for navigating the Forward and Back buttons. The simple solution would be to access the browser-exposed `history` object and pass those URLs to the HTTP `POST`. The problem with the simple solution is that it is not generally viable. Accessing the `history` object by using a script is a security issue, and unless the client has allowed access, will generate an exception.

A more feasible solution is to add an additional HTTP header that uniquely identifies the window used to chain together the HTML pages. Specifically, the property `window.name` can be assigned and is ideally suited to uniquely identify the individual HTML windows. Figure 9-11 illustrates the final solution.

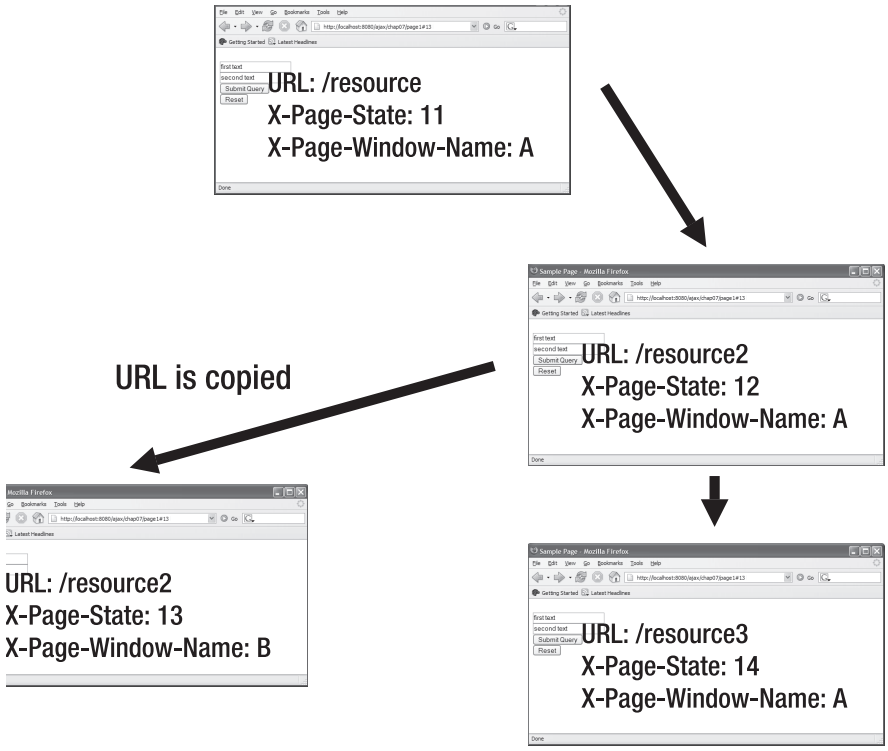

Figure 9-11. *Final solution used to uniquely identify the HTML windows, URL, and associated state*

In Figure 9-11, each browser window instance is unique and can be identified. For example, if the user requests `/resource`, the window identifier is `A`, and the state identifier is `11`. If the user processes the data, `/resource2` is retrieved with a new state identifier `12` and window instance `A`. If the user were to copy the URL to a new browser instance, the URL `/resource2#12` would be copied. The state identifier `12` would be loaded, but the window browser instance is `B`, and therefore the server knows a new window instance has been created, and a new history is being generated. The server will then associate the state `11` with the newly created state identifier `13`. Now both browser instances, `A` and `B`, both share the state identifier `11` in their history. Then if the user clicks the Submit Query button of either window, `A` or `B`, two unique results will occur that do not conflict with each other. If the example were a plane ticket application, two tickets that start at the same location but end in different locations could be purchased.

A new state identifier is created when the page is refreshed. Considering that we can identify the browser instance by using the window name, the state identifier is not necessary. Using a window name as a state identifier creates a state that is accumulated and organized by resource. When a new browser instance and old URL are copied (for example, state identifier `13`), the server is responsible for copying the old state into a new state. The downside of using an accumulated state is that it is not as fine-grained as a state identified by unique identifiers.

Managing State at the Protocol Level

Moving down one level on the technological scale, this section illustrates the HTTP communications between the client and server. The communications are started by having a web browser ask for the resource `http://mydomain.com/ajax/chap07/page1`, which is illustrated by the following request. Note that the illustrated requests and responses are abbreviated and show only the HTTP information that is relevant for the discussion:

```
GET /ajax/chap07/page1 HTTP/1.1
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,
text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
```

The server accepts the request and responds with the following:

```
HTTP/1.1 200 OK
ETag: W/"1017-1126885576349"
Last-Modified: Fri, 16 Sep 2005 15:46:16 GMT
Content-Type: text/html
Content-Length: 1017
Server: Apache-Coyote/1.1
```

In the response, there is an `ETag` indicating that the content could be cached by the web browser. If the `ETag` were sent in response to an `XMLHttpRequest` request, the Cache Controller pattern could have been used. The server-generated response uses the Permutations pattern and contains information that can be represented by a web browser. The generated response represents the empty or generic representation that does not contain a state. When the generated content has been converted into a processed HTML page, the HTML body `onload` event is triggered. Triggering the `onload` event generates a request for the state associated with the resource. Following is the `XMLHttpRequest`-generated request:

```
GET /ajax/chap07/page1/state HTTP/1.1
Accept: application/xml
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
X-Page-Window-Name: window-2005-10-03-10-10-10-1245
X-Page-State: none
```

What is unique in the request for the state from the `XMLHttpRequest` object is that the URL is similar in structure to the resource URL, except that the `state` keyword is appended to the URL. The `state` keyword is necessary so that all proxies and browsers can uniquely identify the resource and the state associated with the resource. Using the same URL would cause problems. In the HTTP request, the additional HTTP headers `X-Page-State` and `X-Page-Window-Name` are used. The header `X-Page-State` defines the state identifier, and the header `X-Page-Window-Name` identifies the name of the window asking for the state. What triggers the server-side State Navigation pattern implementation is either the appended `state` identifier or the `X-Page-State` HTTP header. More about the trigger will be discussed in the server-side code implementation.

And last, notice how the Accept HTTP header accepts only the type application/xml. This is on purpose even though a MIME type such as application/ajax-state would have been more appropriate. It is critical to use application/xml because then the XMLHttpRequest and web browser will recognize the returned data as XML. Using another MIME type causes the XMLHttpRequest type to not parse the generated XML and returns the content only as a text stream. As an architectural side note, a format such as JSON could be used to define the state.

When asked for a state for the first time, the server will not have an associated state and will need to create an empty state. The empty state response is illustrated as follows:

```
HTTP/1.1 200 success
X-Page-State: 11
Date: Sun, 18 Sep 2005 11:19:30 GMT
Server: Apache-Coyote/1.1
```

In the response, the server issues an HTTP 200 command to indicate that the request was a success. The body may be empty, but in the case of the example would be the XML <state></state> to indicate an empty state. An empty state is generated so that the requesting client can go through the hoops of asking for a state, but nothing will be modified. The HTTP header X-Page-State is returned to the client to indicate what the state identifier is, and in this case the state identifier 11 is returned.

When associating a state with a URL, that state could be accessible from every browser regardless of location and therefore be considered a security risk. However, in this example, the state is not accessible everywhere because the URL, window name identifier, and state identifier are tied together. A hacker would have to know all three before being able to access the state. Additionally, for extra security, HTTPS or some form of authentication can be used. Depending on the nature of the state, the solution could involve using requesting IP addresses, authentication information, or even cookies.

It is important to realize that if cookies are used to authenticate a user, the usefulness of the State Navigation pattern is extremely limited. Cookies can cross web browser instances, but not different web browser types or computer locations. The better solution would be to use HTTP authentication because the web browser can ask for HTTP authentication regardless of browser or computer location.

When the HTML page and state requests have been processed, the client can fill out the form with some data. Having added all the data into the form, the user can click the Submit Query button. Clicking the button causes the onsubmit event to be triggered, which results in the State Navigation pattern implementation on the client side to call the server by using the XMLHttpRequest object. The call generates a request that is illustrated as follows:

```
POST /ajax/chap07/linkToPage2.xml HTTP/1.1
Accept: application/xml
Accept-Language: en-ca
Accept-Encoding: gzip, deflate
Content-Type: application/xml
Content-Length: 364
Connection: Keep-Alive
Cache-Control: no-cache
X-Page-Original-Url: /ajax/chap07/page1
X-Page-Window-Name: window-2005-10-03-10-10-10-1245
X-Page-State: 11
```

In the example, the request is an HTTP POST that posts to the static file linkToPage2.xml. What is odd is that an HTTP POST has been made to a document that cannot process the post because the file is obviously not a script. This is the uniqueness of the State Navigation pattern in that when data is posted, the result does not have to be another page that is viewed by the web browser.

Normally, when executing an HTTP POST, a server-side script will process the request and generate some output. Unlike an HTTP GET, where data is retrieved, the HTTP POST expects data to be sent before it is retrieved. When creating a workflow application, HTML pages are tied to each other. For example, if the resource /ajax/chap07/page1 contains a POST to /ajax/chap07/page2, only page1 can call page2, because page2 expects data arriving from page1. Of course, the developer could write within the script of page2 a decision block to test how the script is being called and what it should do. Nevertheless, this makes for messy coding.

What is different with the State Navigation pattern is that the sending of the data and retrieving of the next content are separated. The State Navigation posts to a URL that may or may not process the posted content. The State Navigation would capture the request, store the state, and pass the request to another processor on the HTTP server. Another processor would intercept the request, process the state, and let the HTTP server send the data to the client. The state that is sent to the server can be stored or processed and is the discretion of the web server application. The advantage of this approach is that by using the State Navigation pattern, a URL can cumulatively process the state and generate a single transaction.

The solution provided by the State Navigation pattern is to use the POST as a mechanism to record and process data. The workflow is created by documents that contain a link to the next resource. This makes it possible to separate the dependency of page2 to page1. The result is that the web application allows a reorganization of the HTML content flow, allowing decisions to be made on the fly.

Continuing the communications, the server would respond to the HTTP POST with the following answer:

```
HTTP/1.1 200 OK
X-Page-State: 11
ETag: W/"137-1126885576359"
Last-Modified: Fri, 16 Sep 2005 15:46:16 GMT
Content-Type: application/xml
Content-Length: 137
Server: Apache-Coyote/1.1
```

When the client has loaded the returned data, the script searches for a link that indicates the URL to be loaded by the client. The script then redirects the browser by reassigning location.href, causing the following request to be made by the browser:

```
GET /ajax/chap07/page2 HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg,
image/pjpeg, application/x-shockwave-flash,
application/vnd.ms-excel, application/vnd.ms-powerpoint,
application/msword, */*
Accept-Language: en-ca
Accept-Encoding: gzip, deflate
```

The server will respond with an HTML page that contains no state information. If the loaded HTML page has implemented the State Navigation pattern, the implementation will start from the beginning. The state identifier loaded in the previous page (`page1`) is lost. The identifier is lost to the script, but in the history of the web browser the state identifier has been recorded. When the user clicks the Back button, the browser will make the request for the URL `/ajax/chap07/page1`, and the HTML page state with the identifier `11` will be loaded from the history. This again causes the State Navigation pattern to start from the beginning, except this time there is a state identifier requiring a state to be loaded. When the page from the resource has been loaded again, the `onload` function is called, causing the `XMLHttpRequest` object to search for the state with the identifier `11`, and resulting in the following request:

```
GET /ajax/chap07/page1/state HTTP/1.1
Accept: application/xml
X-Page-Window-Name: window-2005-10-03-10-10-10-1245
X-Page-State: 11
Accept-Language: en-ca
Accept-Encoding: gzip, deflate
```

The server processes the request and generates the following response:

```
HTTP/1.1 200 success
X-Page-State: 12
Content-Type: application/xml;charset=ISO-8859-1
Content-Length: 364
Date: Sun, 18 Sep 2005 11:15:16 GMT
Server: Apache-Coyote/1.1
```

In the response, the state identifier `12` is returned along with some data that the `XMLHttpRequest` object can process. Illustrating the State Navigation pattern visually and at the protocol level explains how this pattern functions. The remaining step is to explain how the client and server code make everything happen.

Implementation

For the State Navigation pattern to function properly, several other patterns need to be combined. Combining the patterns results in an overall architecture that is used to process requests. The question, though, is how to combine the various patterns.

One solution would be to use the Decorator pattern. That would be a good solution, but is implemented by using the already-existing Decorator pattern facilities of the HTTP server. On the HTTP server side, the Decorator pattern is implemented using HTTP filters. An HTTP filter is used to modify or decorate the request and response, without actually changing the intention of the request. For example, to encrypt or decrypt the contents of the response or request, respectively, an HTTP filter would be used. On the client side, the Decorator pattern is not implemented as an HTTP filter, but as a series of encapsulations. Each encapsulation implements an added value functionality.

Figure 9-12 is a graphical representation of the architecture in terms of layers.

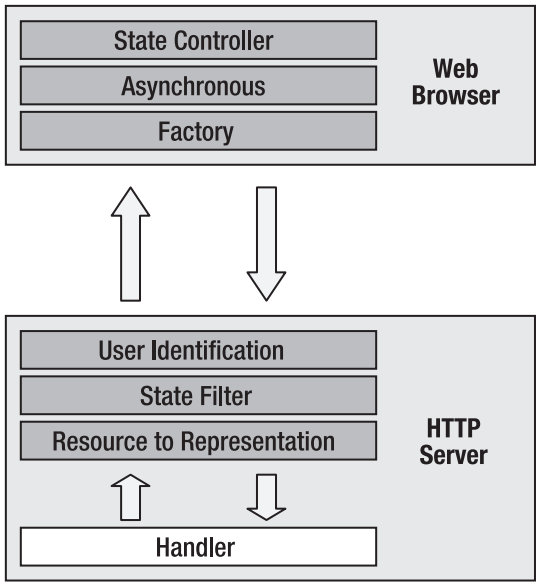

Figure 9-12. *Layered architecture of the State Navigation pattern*

In Figure 9-12, the client side has three layers: Factory, Asynchronous, and State Navigation. The layers Asynchronous and Factory were outlined and implemented in Chapter 2. The layer State Navigation is an implementation of the State Navigation pattern on the client side. The server side has multiple layers, and the User Identification and Resource to Representation layers are modularized implementations of the Permutations pattern. The State Filter layer is an implementation of the State Navigation pattern on the server side. Throughout the rest of this chapter, I will discuss the State Navigation and State Filter layers.

Processing the Requests on the Client

Starting with the client side, the purpose of the State Navigation layer is to send and retrieve the state. As illustrated by the protocol communication, the sequence of events for downloading a resource and its associated state is as follows:

1. Download a URL's representation in the browser.
2. When the download has completed, the HTML body event `onload` is executed.
3. In the `onload` event, the `XMLHttpRequest` object requests the state associated with the resource.
4. The state is downloaded and used to manipulate the HTML document to fill data in an HTML form or to manipulate elements in the HTML document.

If the HTML page contains a Submit button or another HTML element that can be used to send data to the server, the State Navigation layer must be used. Not using the State Navigation layer bypasses the pattern and could corrupt the state. Generally, the sequence of events used to implement the State Navigation pattern is as follows:

1. Clicking, checking, or some other HTML action triggers another HTML action that causes an HTML event.
2. The HTML event, which could be an HTML `POST` or button `onclick`, builds a state that is used by the client State Navigation layer to create a request.
3. The request is sent to the server, which may or may not be processed by the server-side State Filter layer, and is passed on to the handler.
4. If the State Filter layer filters the request, it does so transparently without modifying the contents.
5. The handler generates content that contains a link used by the State Navigation pattern to load the next resource.

In this client-side sequence of events, the State Navigation pattern has three responsibilities: populating the representation with state, generating a state from a representation, and redirecting the page when necessary. The client-side ramification of using the State Navigation pattern is that a representation must possess a state, and that the state can be submitted and retrieved only by using the State Navigation pattern.

The state and reference information that is passed between the client side and server side uses XML, but could include other formats such as JSON. For the scope of this pattern, XML is the default data format. The choice of whether to use another format is left up to you.

Using the State Navigation from an HTML Page

On the client side, the implemented HTML page needs to carry only two major tasks: loading and saving the state, and initiating the state loading and saving process. From an architectural perspective, the HTML page initiates a call to the State Navigation layer, which will call back into the page to carry out the application-specific persistence method calls.

Following is an example implementation of the State Navigation pattern that contains an HTML form as illustrated in Figures 9-4 to 9-7:

```
<html>
<head>
<title>Sample Page</title>
</head>
<script language="JavaScript" src="../lib/factory.js"></script>
<script language="JavaScript" src="../lib/asynchronous.js"></script>
<script language="JavaScript" src="../lib/xmlhelpers.js"></script>
<script language="JavaScript" src="../lib/statecontroller.js"></script>
<script language="JavaScript" type="text/javascript">
StateController.onSaveState = function() {
    return this.saveForm( document.getElementById( "BasicForm"));
}

StateController.onLoadState = function( status, xmlstate) {
    this.loadForm( xmlstate);
}
```

```
</script>
<body onload="StateController.loadState()">
<div id="replace">Nothing</div>
<form id="BasicForm" name="BasicForm"
      onsubmit=" return StateController.saveState()"
      action="/ajax/chap07/link.xml" method="POST" >
    <input name="param1" type="text" /><br />
    <input name="param2" type="text" /><br />
    <input type="submit"/><br />
    <input type="reset"/>
</form>
</body>
</html>
```

To explain the HTML page, the code will be cross-referenced to the events that are used to load or save the state.

When an HTML page has finished loading, an event is triggered to indicate that the document is complete and can be manipulated. The event `onload` is attached to the HTML `body` tag and assigned to calling the method `StateController.loadState()`. The variable `StateController` is a global instance that implements the client-side State Navigation layer. This variable is a global instance because an HTML page cannot contain two states, and hence making the variable a type would be pointless. The `StateController.loadState()` method is called when the HTML document has finished loading and is used to retrieve the state of the HTML page from the server side. In the implementation of the `loadState` method, the `XMLHttpRequest` object is called and asks the server for the state of the page. The server eventually responds, and `StateController` will automatically call the `StateController.onLoadState` method defined in the HTML page. The `onLoadState` method is called after the server has responded with the associated state of the page. The client is responsible for picking apart what the state is and updating the HTML page with the information. In the example HTML page, the method `loadForm` is called and delegates the state restoration to a `StateController`-implemented standard function that deserializes the incoming state.

Saving the state is a bit more complicated because it requires that the HTML page interject the HTML form-posting process. To interject, the `onsubmit` event is implemented and calls the method `StateController.saveState()`. Calling the `saveState` method will trigger a State Navigation–defined process that calls the `StateController.onSaveState` method defined in the HTML page. Within the `onSaveState` method, the HTML page will generate a user-defined state that is saved. In the case of the HTML page, the method `saveForm` is called, which will serialize a particular HTML form. In most cases, the Representation Morphing pattern is implemented.

From the perspective of the HTML page, saving and loading the page state requires making the right calls at the right moment. What is important is that the user has the ability to define when and how the state associated with the HTML page is managed. This means that a page state could be saved as the result of a specific hyperlink that is clicked, and that the page state is reloaded as the result of some button that is clicked. Or the developer could choose to ignore saving the state by adding a Cancel or Ignore button. It is the choice of the developer.

As a matter of simplicity and illustration of the Representation Morphing pattern, I chose to define the state of the HTML page as an HTML form, but I could just as easily have defined

it as being some text in an HTML div section. It is the responsibility of the HTML methods onSaveState and onLoadState to load and save the data.

In the HTML page implementation, the methods onSaveState and onLoadState could be construed as black magic in that they generate and process data without explaining what the data is. What happens is that the data generated and processed by the HTML page is a blob that is sent and received from the server. Only the client needs to know what the data is, and not the State Navigation implementation. In the example, the variable StateController uses XML persistence as a default persistence format. As a result, when the client uses the standard method calls saveForm and loadForm, the generated data is XML based and would be identical to the following:

```
<state>
    <html-page>
        <form id="BasicForm" >
            <element id='param1' type='text'>Value 1</element>
            <element id='param2' type='text'>Value 2</element>
        </form>
    </html-page>
</state>
```

The root node is data, and contained within it is the HTML state that includes reference information about the state, and the state information associated with the HTML page. Specifically, the individual XML tags are identified as follows:

- html-page: Is a parent XML element used to contain the state details associated with an HTML page.
- form: Is a parent XML element used to contain the values for all HTML form elements.
- element: Identifies a state that is associated with an HTML form element. In the example, all HTML form elements are associated with the XML tag element. But it is also possible to use the id attribute as an XML element identifier. This results in the transformation <param1 ... /> for the XML element with the attribute value param1. Which approach you use depends on your preference; either approach is acceptable.

For consistency, the state that is sent is identical to the state that is received.

The Details of the State Navigation

StateController is a variable instance and a custom single kind of a type. StateController could have been defined as a type that is instantiated, but that would be adding unnecessary complexity. As I am explaining the technical details of StateController, I won't explain all of the code at once. What I will explain is the source code in three pieces: the first piece contains the data members, the second piece contains the logic used to load the state, and the last piece contains the logic to save the state.

Following is the code piece that defines the data members of the StateController variable:

```
var StateController = {
    username: null,
    password: null,
    postURL: null,

    constPageStateHeader : "X-Page-State",
    constPageWindowName  : "X-Page-Window-Name",
    constPageOriginalURL : "X-Page-Original-URL",
    constPageWindowNamePrefix : "StateController",
    constResourceStateContentType : "application/xml",
    constURLStateIdentifier : "/state",

    constStateTag : "state",
    constHtmlPageStateTag : "html-page",
```

StateController has nine data members, which all relate to sending and receiving data to and from the server. The data members username and password are the authentication identifiers used when accessing protected resources. The data member postURL indicates the URL used to post the data to the server. In a traditional HTML form, postURL would be the action attribute identifier. As an optimization, if an HTML form is serialized, the data member postURL is assigned the HTML form action attribute value. The remaining data members are used to dissect the communications between the client and server, and to generate and parse the XML data.

Loading the State

When the HTML page is loaded, the event onload is triggered and causes the HTML state to be retrieved and added to the HTML page. The onload event is the usual place to put the state-loading functionality, but any other event could be used. Regardless of where the state-loading functionality is added, three functions are related to state loading: client implementation, default form loading, and overall controlling functionality.

Following is the default client implementation:

```
onLoadState : function( status, responseXML) { },
```

The method onLoadState, when it is not implemented by the HTML page, is an empty implementation that does nothing. In the example HTML page, the onLoadState method called the method loadForm, which is used as a prepackaged function to load the state of an HTML form. The method loadForm is implemented as follows:

```
extractFormData : function( element, objData) {
    if( element.nodeType == 1) {
        if( element.nodeName == "form") {
            objData.formId = element.attributes.getNamedItem( "id").nodeValue;
            objData.formNode = document.forms[ objData.formId];
        }
```

```
        else if( element.nodeName == "element") {
            if( objData.formNode != null) {
                var elementIdentifier =
element.attributes.getNamedItem( "id").nodeValue;
                if( element.childNodes[ 0] != null) {
                    var elementValue = element.childNodes[ 0].nodeValue;
                    objData.formNode.elements[
elementIdentifier].value = elementValue;
                }
            }
        }
    }
},
loadForm : function( xmlState) {
    this.verify = this.extractFormData;
    var objData = new Object();
    XMLIterateElements( this, objData, xmlState);
},
```

The implementation of the method `loadForm` is relatively Spartan, and is used to prepare the iteration of the XML file by using the standard function `XMLIterateElements` and the user-defined function `extractFormData`. The assignment of the `this.verify` method is used to determine whether an iterated XML element is of interest to the user. The creation of the `objData` variable is required for the function `XMLIterateElements` and will contain the found data. The combination of `objData` and the `this.verify` method deserialize XML content into JavaScript data members. The function `XMLIterateElements` is used to process an XML file and is implemented as follows:

```
function XMLIterateElements( objVerify, objData, element) {
    objVerify.verify( element, objData);
    for( var i = 0; i < element.childNodes.length; i ++) {
        XMLIterateElements( objVerify, objData, element.childNodes[ i]);
    }
}
```

The function `XMLIterateElements` has three parameters. The parameter `objVerify` is an object instance that is called to process an XML element. The parameter `objData` is a data object that is manipulated by the `objVerify.verify` method. An object instance is used so that a `verify` implementation can access the data members of a current object instance. The parameter `element` represents an XML node. After having called the `objVerify.verify` method, the child nodes of the element XML node are iterated, and for every iteration the function `XMLIterateElements` is called recursively. The result is that for each and every element, the method `objVerify.verify` is called. The purpose of `objData` is to allow the method `objVerify.verify` to assign some data members that can be referenced at some later point, during and after the iteration of all the XML elements.

Let's look back at the implementation of `loadForm`. Notice that it is calling the function `XMLIterateElements` and that the `objVerify.verify` method refers to the method `extractFormData`. Looking at the implementation of `extractFormData`, you can see that a

number of XML DOM methods (for example, `element.[example method]`) are called. What happens in the `extractFormData` method is that the XML elements `form` and `element` are searched for. If a `form` XML element is found, it is a reference to an HTML form that is retrieved from the current HTML page by using the method `document.forms[objData.formId]`. The HTML form reference is needed to assign individual HTML form elements. If an `element` XML element is found, the value is assigned by retrieving the identifier from the XML attribute (`element.attributes.getNamedItem("id").nodeValue`) and the child value (`element.childNodes[0].nodeValue`). The form element is assigned the value by using the reference `objData.formNode.elements[elementIdentifier].value = elementValue`. In the example, there is no attempt made to test whether the HTML element is a check box or list box. This was done on purpose to keep the explanation simple, and in the complete implementation of `extractFormData` those additional attributes would be tested.

The method `loadState` is used to begin the HTML page-loading process and is defined as follows:

```
loadState : function() {
    if( location.hash != null) {
        var asynch = new Asynchronous();
        var thisReference = this;
        asynch.openCallback = function( xmlhttp) {
            if( location.hash.length == 0) {
                xmlhttp.setRequestHeader(
                    thisReference.constPageStateHeader, "none");
            }
            else {
                xmlhttp.setRequestHeader( thisReference.constPageStateHeader,
                    location.hash.slice(1));
            }
            xmlhttp.setRequestHeader( "Accept",
                thisReference.constResourceStateContentType);
            thisReference.verifyWindowName();
            xmlhttp.setRequestHeader( thisReference.constPageWindowName,
                window.name);
        }
        var xmlhttp = asynch._xmlhttp;
        asynch.complete = function( status, statusText, responseText, responseXML) {
            thisReference.verify = thisReference.extractUserData;
            var objData = new Object();
            XMLIterateElements( thisReference, objData, responseXML);
            if( objData.foundElement) {
                thisReference.onLoadState( status, objData.foundElement);
            }
            location.replace( location.pathname + "#" +
                xmlhttp.getResponseHeader( thisReference.constPageStateHeader));
        }
```

```
        asynch.username = this.username;
        asynch.password = this.password;
        var splitLocation = location.href.split( "#");
        asynch.get( splitLocation[ 0] + this.constURLStateIdentifier);
    }
},
```

In the implementation of `loadState`, the `Asynchronous` class that was explained in Chapter 2 is used. The implementation is relatively straightforward in that an HTTP `GET` is executed, as illustrated by the method call `asynch.get`. To let the server know that the request is a state request, the HTTP headers identifying the state (`X-Page-State`, `X-Page-Window-Name`) are sent by using the method `xmlhttp.setRequestHeader`. If the HTTP headers were not present, the server side would consider the request as a generic HTTP `GET`. The HTTP headers are used, as they are easy to verify for existence. Another approach would have been to search for the `state` keyword in the URL, but that would require parsing the URL. The method `thisReference.verifyWindowName` is used to generate a State Navigation pattern–compliant window name if there is no window name. Additionally, to identify the request as a state request, the `/state` is appended to the URL when the `asynch.get` method is called. Not adding the `/state` would result in confusing the proxy and potentially having two different representations with a single URL, and as per the Permutations pattern, doing so would be incorrect.

In the implementation of the `asynch.complete` inlined function, the returned XML document is processed. The `XMLIterateElements` function is used to find the XML node that contains the HTML page state. If the XML node (`objData.foundElement`) is found, the loading of the state is passed to the HTML page–implemented function `onLoadState`. Because a state is retrieved and the server has the power to determine the state identifier, the State Navigation layer must replace the current state identifier. The state identifier is replaced by using the `location.replace` method so that the old page will be replaced in the history log. Not using the `location.replace` method would result in the addition of an HTML page to the history, and that would confuse the user from a Forward and Back button navigation perspective. For example, if the history contained `/other/url`, `/resource`, `/resource2`, not using the `replace` method would generate a history `/other/url`, `/resource`, `/resource#11`, `/resource2`, instead of `/other/url`, `/resource#11`, `/resource2`.

Saving the State

The implementation of saving the state requires three methods: `onSaveState` implemented by the client to save the state, the helper method `saveForm` to convert an HTML form into a state, and `saveState` to initiate the state persistence.

The method `onSaveState` needs to be implemented by the HTML page, but the default is that the `StateController` provides an empty implementation that does nothing. If `StateController` did not provide a default implementation, a JavaScript error would result. Following is the default empty implementation of the method `onSaveState`:

```
onSaveState : function() {
    return "";
},
```

The default implementation of `onSaveState` illustrates the most important thing that any implementation must do: return a buffer that contains the persisted state. It is expected that

the buffer that is returned is formatted in XML and that the returned data is XML compliant. The onSaveState-generated XML is inserted as a child of the html-child XML element.

In the HTML client, the implemented method onSaveState calls the method saveForm, which is used to convert the HTML form into an XML data structure. Following is the implementation of the method saveForm:

```
saveForm : function( form) {
    this.postURL = form.action;
    var buffer = "";
    buffer += "<form id=\"" + form.name + "\" >\n";
    for( var i = 0; i < form.elements.length; i++) {
            if( form.elements[ i].type != "submit" &&
                form.elements[ i].type != "reset") {
                buffer += "<element id='" + form.elements[i].name + "' type='" +
                                form.elements[ i].type + "'>" +
                                form.elements[ i].value + "</element>\n";
            }
    }
    buffer += "</form>\n";
    return buffer;
},
```

In the implementation of saveForm, the parameter |form represents the HTML form to persist. The first step of the implementation is to assign to the local instance the form post URL (this.postURL) from the form.action value. Next, the root form XML element is created and assigned to the name of the form. Then the individual HTML form elements are added to the state by iterating the collection form.elements. All HTML form elements are added to the XML data structure so long as the elements are not of type submit or reset. In the saveForm method implementation, only text box elements can be persisted. Normally, though, all HTML form elements could be persisted, but for this explanation the implementation was kept simple.

Calling the method saveState, which is implemented as follows, starts the saving of the state:

```
saveState : function() {
    var buffer = "<" + this.constStateTag + ">";
    buffer += "<" + this.constHtmlPageStateTag + ">\n";
    buffer += this.onSaveState()
    buffer += "</" + this.constHtmlPageStateTag + ">\n";
    buffer += "</" + this.constStateTag + ">";
    var request = new Asynchronous();
    var thisReference = this;
    var oldPath = location.pathname;
    request.openCallback = function( xmlhttp) {
        if( location.hash.length == 0) {
            xmlhttp.setRequestHeader( thisReference.constPageStateHeader, "none");
        }
        else {
            xmlhttp.setRequestHeader( thisReference.constPageStateHeader,
                location.hash.slice(1));
        }
```

```
        thisReference.verifyWindowName();
        xmlhttp.setRequestHeader( thisReference.constPageWindowName, window.name);
        var splitLocation = location.href.split( "#");
        xmlhttp.setRequestHeader(
thisReference.constPageOriginalURL, splitLocation[ 0]);
    }
    var xmlhttp = request._xmlhttp;
    request.complete = function( status, statusText, responseText, responseXML) {
        if(status == 200 && responseXML != null) {
            thisReference.verify = thisReference.extractLink;
            var objData = new Object();
            XMLIterateElements( thisReference, objData, responseXML);
            location.replace( oldPath + "#" + xmlhttp.getResponseHeader(
                thisReference.constPageStateHeader));
            location.href = objData.redirectURL;
        }
    }
    request.username = this.username;
    request.password = this.password;
    request.post( this.postURL, this.constResourceStateContentType,
        buffer.length, buffer);
    return false;
}
```

The implementation of the method `saveState` is more complicated and is responsible for generating the XML state and for sending the state to the server by using an HTTP `POST`. The variable `buffer`, and those lines at the beginning of `saveState` that reference `buffer`, are used to build the XML state. In the building of the state, the method `this.onSaveState` is called, letting the HTML page generate the custom parts of the persistence. After the XML state is constructed, it needs to be posted to the server.

As in previous pattern implementations, an HTTP `POST` is created by using the `Asynchronous` class type outlined in Chapter 2. What is unique in this posting is the assignment of the custom HTTP headers (`X-Page-State`, `X-Page-Window-Name`, and `X-Page-Original-URL`) and the processing of the response. In the inlined function implementation of `request.openCallback`, the state reference identifier is extracted from the local URL as stored in the variable `location.hash`. If the `location.hash` value does not exist, a `none` is sent as the page state header to indicate that a state identifier should be created. Otherwise, the `location.hash` value is sent, minus the prefixed # character. What is new for this request is the assignment of the original URL (`X-Page-Original-URL` or `constPageOriginalURL`). This assignment is necessary so the server can cross-reference the state with the resource. Moving to the end of the `saveState` method implementation, you can see that the method `request.post` is used to post the data to the server.

When the HTTP `POST` returns, the inlined method implementation `request.complete` is called to process the returned XML. The returned XML looks similar to the following:

```
<data>
    <link id="redirect" href="/ajax/chap07/page2" />
</data>
```

The returned XML content is processed by the inlined method `request.complete`. In the inlined method `request.complete`, the returned XML content is parsed by using the function `XMLIterateElements`. The function `XMLIterateElements` is a helper function used to process the returned XML content. Specifically, the function `XMLIterateElements` extracts the destination link from the returned XML content, which in the example happens to be the URL `/ajax/chap07/page2`. In the inlined method, the extracted URL is assigned to the variable `objData.redirectURL` by the function `XMLIterateElements`. But before the extracted URL is reassigned, the state hash code is updated by calling the `location.replace` method. Then after the script replaces the URL, the script can navigate to the extracted URL by using the method `location.href`.

A little side note needs to be added about the purpose of the returned XML. Figure 9-3 shows the problems when HTTP `POST` is used to navigate from HTML page to HTML page. As the State Navigation pattern navigates using the returned XML, the posting of the same content multiple times does not occur. Navigation occurs when using a script that uses HTTP `GET` techniques that are called after a successful posting. And the posting of the state twice is impossible because the state identifier is incremented for each posting. So if a resource is responsible for charging a credit card, posting the same content multiple times can be caught by the server and curtailed.

Processing the Requests on the Server

After the HTML page has been implemented and the state has been loaded and saved on the client side, the remaining responsibility lies with the server.The server will receive the state and store it somewhere, and send the state when it is asked for. However, the general pattern implementation on the server does not attempt to interpret the state, because doing so would add processing that is not necessary. The exception occurs when the processing of the state is application related. The state is processed on the server side by using HTTP handlers.

Knowing When and How to Trigger HTTP Filters

When implementing an HTTP filter, the idea is not to process the request, but to modify and redirect the request. With respect to the HTTP protocol, there are two ways to trigger an HTTP filter: URL and HTTP header. It is possible to trigger an HTTP filter based on some piece of data sent to or generated by the server in the HTTP body. For example, in the state sent to the server, there could be a keyword indicating further actions for the HTTP filter. But using a piece of data in the HTTP body itself is a bad idea because it requires extra processing by the server. The data in the HTTP body is specific to the HTTP handler and should be considered as a single entity. This does not mean that an HTTP filter, after it has been triggered, cannot inspect and manipulate the payload. What this means is that for trigger purposes only, the URL or HTTP header is inspected.

Let's go through the differences of URL vs. HTTP header by using the State Navigation pattern as an example. In the section "Managing State at the Protocol Level," an HTTP header is used to request a state that is associated with a URL, as illustrated by the following HTTP request part:

```
GET /ajax/chap07/page1/state HTTP/1.1
Accept: application/ajax-state
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
X-Page-Window-Name: window-2005-10-03-10-10-10-1245
X-Page-State: none
```

The request has two pieces of information that could trigger a filter: URL (`[url]/state`) and HTTP header (`X-Page-State`). To trigger a filter via a URL, the URL must be processed. Processing a URL is a relatively expensive step and potentially buggy. The bugginess results when a URL has the same text as a trigger. The header is necessary only when the state is retrieved, and that occurs only when the `XMLHttpRequest` object is used. Thus adding an HTTP header is not complicated or inconvenient. Which solution you use depends on your URLs and what you are comfortable with.

The rule of thumb is that an HTTP header can be used when both the client and the server are capable of processing the custom header, and a URL should be used whenever the server knows what kind of client will process the data.

When implementing an HTTP filter, a basis class that executes the trigger and runs the filter action is illustrated by using the following Java filter code. On other platforms and programming languages, the code will be similar because other platforms also have the concept of an HTTP filter.

```
public abstract class TriggerFilter implements Filter {
    public abstract Object initializeRequest();
    public abstract void destroyRequest( Object objData);
    public abstract boolean isTrigger( Object objData,
        HttpServletRequest httprequest, HttpServletResponse httpresponse);
    public abstract void runFilter( Object objData,
        HttpServletRequest httprequest, HttpServletResponse httpresponse,
        FilterChain chain) throws IOException, ServletException;
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
        HttpServletRequest httprequest = (HttpServletRequest)request;
        HttpServletResponse httpresponse = (HttpServletResponse)response;
        Object data = initializeRequest();
        if( isTrigger( data, httprequest, httpresponse)) {
            runFilter( data, httprequest, httpresponse, chain);
        }
        else {
            chain.doFilter( request, response);
        }
        destroyRequest( data);
    }
}
```

When implementing a Java filter, the class implements the `Filter` interface. Two methods are not illustrated: `init` and `destroy`, which are used to initialize and destroy, respectively, the

filter instance. They are not illustrated for simplicity purposes. The class `TriggerFilter` is implemented as an abstract class because `TriggerFilter` on its own is not very useful and provides a basic functionality that would otherwise be constantly implemented. `TriggerFilter` implements the Template pattern, and therefore to have anything happen, some class has to subclass `TriggerFilter`.

The method `doFilter` is part of the interface `Filter` and is called whenever an HTTP request is made. When the method `doFilter` is called depends on the order of the filter in the configuration file. The order in the configuration file is a Java feature, and other platforms may have other ways to define the order indicating when a filter is called.

When the method `doFilter` is called, the parameters `request` and `response` are converted into the types `HttpServletRequest` and `HttpServletResponse`, respectively. This is necessary because the `Http` types offer methods and properties that help process an HTTP request.

The methods `isTrigger` and `runFilter` are declared abstract, which means any class that extends `TriggerFilter` will need to implement the abstract methods. The method `isTrigger` is called to check whether the request should be processed by the implemented subclass. The method `runFilter` is executed to process the HTTP request. If `isTrigger` returns a value of `false`, the HTTP request processing continues as usual, and in the case of the example the method `chain.doFilter` is called.

The State Navigation pattern is implemented by using the `TriggerFilter` class, but before the architecture of the State Navigation pattern is detailed, the Permutations Pattern is rewritten to use the `TriggerFilter` class.

Rewriting the Permutations Pattern Implementation

The purpose of rewriting the Permutations pattern is to illustrate how the pattern can be implemented as a filter instead of a handler. The difference between a handler and filter is not huge, but there are some structural changes. Following is the implementation of the Permutations pattern using Java:

```
public class ResourceEngineFilter extends TriggerFilter {
    private FilterConfig _filterConfig;

    private Router _router;
    private String _clsRewriter;

    public void init(FilterConfig filterConfig) throws ServletException {
        _filterConfig = filterConfig;
        try {
            _router = (Router)ResourceEngineFilter.class.getClassLoader().loadClass(
                filterConfig.getInitParameter("router")).newInstance();
            _router.setProperty( "base-directory", baseDirectory);
            _clsRewriter = filterConfig.getInitParameter("rewriter");
        }
        catch (Exception e) {
            throw new ServletException( "Could not instantiate classes ", e);
        }

    }
```

```
    public Object initializeRequest() {
        return null;
    }
    public void destroyRequest( Object objData) {

    }
    public boolean isTrigger(Object objData, HttpServletRequest request,
        HttpServletResponse response) {
        if (_router.IsResource( request)) {
            return true;
        }
        return false;
    }
    public void runFilter(Object objData, HttpServletRequest request,
        HttpServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        Rewriter rewriter;
        try {
            rewriter = (Rewriter)ResourceEngineFilter.class.getClassLoader().
                loadClass(_clsRewriter).newInstance();
        }
        catch( Exception ex) {
            return;
        }
        _router.WriteRedirection( rewriter, request);
    }
}
```

In the example class `ResourceEngineFilter`, the methods `isTrigger` and `runFilter` are implemented, as required by `TriggerFilter`. The method `init` is used to initialize the filter and retrieve the filter configuration information, and specifically the base-directory that is used by the class `FilterRouter` or the `Router` interface instance.

In the implementation of `init`, the default `Router` instance `_router` is instantiated by using the configuration declaration item `router`. In contrast, in the Permutations pattern implementation, the instantiation of `Router` was hard-coded. Regardless of how the `Router` interface instance is instantiated, in the example of `ResourceEngineFilter`, the `Router` interface instance must be stateless with respect to the HTTP request. The statelessness is required because the `Router` instance is associated with the `ResourceEngineFilter`, which is also stateless. What is not stateless, but is instantiated with every triggered filter request, is the `Rewriter` interface instance. This is because the implementations of the `Rewriter` will require multiple calls, and the calls will reference some state generated by the HTTP request.

The statelessness results in a modified version of the `Router` interface that is defined as follows:

```
public interface Router {
    public void setConfiguration( String key, String value);
    public boolean IsResource(HttpServletRequest request);
    public void WriteRedirection( Rewriter rewriter, HttpServletRequest request);
}
```

The modification of the interface involves the addition of the `setConfiguration` method, which assigns the configuration information. The configuration information is used by the `Router` interface implementation when figuring out whether a request is a resource or a specific representation. The method `WriteRedirection` has been modified to include the parameter `rewriter`. As the configuration information is passed to the `Router` interface instance, having the parameter `rewriter` may not seem necessary. It is necessary because otherwise a hidden dependency in the implementation of the interfaces is created, complicating the development of modular code.

The implementations of the `Rewriter` and `Router` interfaces remain as illustrated in the Permutations pattern. The resulting implementation is a prototype example for the server side that can be used to filter implementations. When implementing the Decorator pattern, the filters should be stacked by using the HTTP filter mechanism. What is important is the ordering of the filters, because some HTTP filter implementations have an ordering dependency.

Implementing the State Layer

In Figure 9-12, the Resource to Representation filter appears after the State filter, which is important so that not all requests need to be processed. For example, when retrieving the associated state of a resource, it is not necessary to execute a handler. The State layer captures the associated state request and processes it directly.

Managing the State Calls

As per the previous discussion, the filter needs to implement two functionalities: storing the state and retrieving the state. The State layer will extend the `TriggerFilter` class, and the implementation will be outlined in four pieces.

The first piece is the filter initialization:

```
public class StateFilter extends TriggerFilter {
    private FilterConfig _filterConfig;
    private StateManager _stateManager;
    private String _resourceStateContentType;
    private String _XPageState;
    private String _XPageWindowName;
    private String _URLStateIdentifier;
    private int _URLStateIdentifierLength;
    private String _XPageOriginalURL;
```

```
public void init(FilterConfig filterConfig) throws ServletException {
    _filterConfig = filterConfig;
    _resourceStateContentType = filterConfig.getInitParameter(
         "resource-state-content-type");
    _XPageState = filterConfig.getInitParameter( "page-state-header");
    _XPageWindowName = filterConfig.getInitParameter( "page-window-name");
    _URLStateIdentifier = filterConfig.getInitParameter(
        "url-state-identifier");
    _URLStateIdentifierLength = _URLStateIdentifier.length();
    _XPageOriginalURL = filterConfig.getInitParameter( "page-original-url");
    try {
        String strClass =
            filterConfig.getInitParameter("state-manager");
        _stateManager = (StateManager)
            StateFilter.class.getClassLoader().loadClass(
                filterConfig.getInitParameter(
                "state-manager")).newInstance();
    }
    catch (Exception e) {
        throw new ServletException( "Could not instantiate _stateManager", e);
    }
}
```

In the implementation of `StateFilter`, the data member assignments are dynamic and can be specified in the HTTP server configuration file. Not all data members will be explained because that would be too lengthy and redundant. The data members `_resourceStateContentType` and `_XPageState` are the counterparts to the client-side-defined `StateController.constResourceStateContentType` and `StateController.constPageStateHeader` data members, respectively. The data member `_stateManager` is the state manager implementation. The idea is that the filter manages the state retrieval and storage calls, whereas `_stateManager` is the implementation of the retrieval and storage of the state. By separating the actual doing from the calling functionality, the doing can determine which persistence medium is used. For the scope of this book, the persistence medium is the memory, but could also be implemented to use a database or hard disk.

The second piece of code relates to the object used to manage state and resource reference information that is created on a per request instance and is passed to the `isTrigger` and `runFilter` routines. The implementations for `initializeRequest` and `destroyRequest` are as follows:

```
private class Data {
    public String _method;
    public String _stateHeader;
    public String _windowName;
    public int _operation;
    public String _path;
    public void reset() {
        _method = null;
        _stateHeader = null;
        _operation = OP_NONE;
```

```
            _path = null;
            _windowName = null;
        }
    }
    public Object initializeRequest() {
        return new Data();
    }
    public void destroyRequest( Object objData) {
    }
```

The class Data is declared as a private class and is used only in the scope of the StateFilter class. Five publicly declared data members reference the HTTP method, HTTP state header, window name, path representing the URL, and locally defined operation type. In the implementation of initializeRequest, a new instance of Data is returned. There is no implementation for destroyRequest because it is not necessary to do anything when the object is destroyed.

The third piece of the State filter is the code to test whether the request or post is related to manage the server-side state:

```
    private static final int OP_NONE = 0;
    private static final int OP_RETRIEVE = 1;
    private static final int OP_POST = 2;

    public boolean isTrigger( Object inpdata, HttpServletRequest httprequest,
       HttpServletResponse httpresponse) {
        String tail = httprequest.getRequestURI().substring(
            httprequest.getRequestURI().length() - _URLStateIdentifierLength);
        String stateHeader = httprequest.getHeader( _XPageState);
        Data data = (Data)inpdata;

        if( tail.compareTo( _URLStateIdentifier) == 0) {
            data._path = httprequest.getRequestURI().substring( 0,
                httprequest.getRequestURI().length() - _URLStateIdentifierLength);
        }
        else {
            if( stateHeader == null) {
                return false;
            }
            data._path = httprequest.getRequestURI();
        }

        data._method = httprequest.getMethod();
        data._stateHeader = stateHeader;
        data._operation = OP_NONE;
        data._windowName = httprequest.getHeader( _XPageWindowName);
        if( data._method.compareTo( "GET") == 0) {
            data._operation = OP_RETRIEVE;
            return true;
        }
```

```
        else if( data._method.compareTo( "PUT") == 0 ||
            data._method.compareTo( "POST") == 0) {
            if( _resourceStateContentType.compareTo(
                httprequest.getContentType()) == 0) {
                data._path = httprequest.getHeader( _XPageOriginalURL);
                data._operation = OP_POST;
                return true;
            }
        }
        data.reset();
        return false;
    }
```

The method `isTrigger` is used to determine whether the method `runFilter` should execute, and if so, `isTrigger` populates the `Data` type instance. That way, if `runFilter` executes, `runFilter` will not need to organize the details of the state or resource call. For the method `isTrigger`, the first parameter `inpdata` is the object instantiated by the method `initializeRequest`. Hence the first step of `isTrigger` is to typecast the parameter to the type `Data` and assign the instance to the variable `data`.

The variable `tail` is the end of the URL and is used to test whether the state identifier `/state` is present. If the identifier does exist as per the decision (`if( tail.compareTo( _URLStateIdentifier...)`, the URL assigned to `data._path` must not contain the `state` keyword. If the URL does not contain the `state` keyword, a test is made to see whether the state header (`stateHeader`) exists. If the state header does not exist, the request is not a State Navigation request. If the state header does exist, the URL assigned to `data._path` is the same as the input URL. The example illustrates testing for two conditions, but it is possible to test for only a single condition and make a decision. The example of two conditions was shown to illustrate the code for each condition.

If the code after the initial decision block is reached, we are assured the request is a State Navigation request and the standard variables can be assigned. The variables `data._method`, `data._stateHeader`, and `data._windowName` are assigned to the HTTP method, HTTP header, and window name, respectively, so that they may be used by the `runFilter` method.

The last decision block in the implementation of `isTrigger` tests which State filter operation is being executed. The operation can be one of two values: HTTP `GET` or HTTP `POST`. Support is added for the HTTP `PUT`, which is classified as an HTTP `POST`. If either decision block returns a `true` value, the data member `data_operation` is assigned to `OP_RETRIEVE` or `OP_POST`.

Having `isTrigger` return a `true` value will cause the `runFilter` method to be executed, which is implemented as follows:

```
    public void runFilter(Object inpdata, HttpServletRequest httprequest,
        HttpServletResponse httpresponse, FilterChain chain)
            throws IOException, ServletException {
        Data data = (Data)inpdata;
        if( data._operation == OP_RETRIEVE) {
            State state;
            if( data._stateHeader.compareTo( "none") == 0) {
                state = _stateManager.getEmptyState( data._path, data._windowName);
            }
```

```
            else {
                state = _stateManager.copyState( data._stateHeader, data._path,
                    data._windowName);
            }
            httpresponse.setContentType( _resourceStateContentType);
            httpresponse.setHeader( _XPageState, state.getStateIdentifier());
            httpresponse.setStatus( 200, "success");
            PrintWriter out = httpresponse.getWriter();
            out.print( state.getBuffer());
            return;
        }
        else if( data._operation == OP_POST) {
            ServletInputStream input = httprequest.getInputStream();
            byte[] bytearray = new byte[ httprequest.getContentLength()];
            input.read( bytearray);
            State state = _stateManager.copyState( data._stateHeader, data._path,
                data._windowName);
            state.setBuffer( new String( bytearray).toString());
            httpresponse.addHeader( _XPageState, state.getStateIdentifier());
            chain.doFilter( httprequest, httpresponse);
            return;
        }
    }
```

In the implementation of `runFilter`, the first parameter is the object instance allocated by the method `initializeRequest`. And as with `isTrigger`, a typecast is made to convert the type and assign it to the variable `data`. From there, the decision blocks are based on the data members of the variable data that were assigned in `isTrigger`.

There are two state operations: retrieve state and post state. The first decision block (`== OP_RETRIEVE`) tests whether the operation is a state retrieval, and the second decision block (`== OP_POST`) tests whether the operation is a post. If the operation is a state retrieval and the asked-for state is `none` (indicating that the client has not associated a state with an HTML page), an empty state is created. A new empty state is created by using the method `getEmptyStateHashcode()`, and the method `getHashcode()` retrieves the hash code of a state. By default, when creating an empty state, a hash code will automatically be created. Using the method `copyState` copies the old state to a new state, and is explained shortly. After calling the `copyState` method, various methods on the `httpresponse` variable are called to generate the response.

If the operation is a posting, the posted stream is retrieved from the request by using the method `input.read`. The read buffer is stored in the variable `bytearray`, which happens to be an array of bytes. As when a state is retrieved, the state is copied by using the method `copyState`, and then assigned by using the method `state.setBuffer`. The state is copied from the original reference, and the new data overwrites the old. By copying a state, the state manager can create a trail of dependencies and associations that could be used by the state manager for optimization purposes. In the response, the newly generated state header is added by using the method `addHeader`.

The last and very important step is to call the method `chain.doFilter` because that allows the posting to be processed by a handler. This raises the question, "If the state is stored, why

process it?" Let's say that I buy a ticket and fill out the form. When I click the Submit button, I want to buy the ticket; but when I click Back, I want to know the form details used to buy the ticket. Knowing the details, I can click Forward, and a ticket will not be bought twice—which would have happened if I had to click Submit. Therefore, to buy the ticket, some handler has to process the posted data, thus requiring the State filter to store the data and to let the handler process the data.

An Example State Manager Handler

In the `StateFilter` implementation, the variable `_stateManager` references the type `StateManager`. The type `StateManager` is an interface and manages the state that is posted and retrieved. Using interfaces makes it possible to separate intention from implementation as per the Bridge pattern.

The `State` interface is defined as follows:

```
public interface State {
    public String getURL();
    public void setURL( String URL);
    public String getWindowName();
    public void setWindowName( String windowname);
    public String getBuffer();
    public void setBuffer( String buffer);
    public String getStateIdentifier();
    public void setStateIdentifier( String hashcode);
}
```

The interface is based on four properties (`URL`, `WindowName`, `Buffer`, and `StateIdentifier`) that are implemented as getters and setters. The property `Buffer` is used to assign and retrieve the state sent by the client. The property `StateIdentifier` is used to assign and retrieve the state identifier of an HTML page. The property `URL` is the URL of the state, and finally `WindowName` is the associated window name. A minimal implementation of the `State` interface would define four private data members of the type `String`, `String`, `String`, and `String`.

What is more complicated is the implementation of the `StateManager` interface. An advanced implementation is beyond the scope of this book and depends on the context of the problem. The `StateManager` interface is important in the overall architecture because it is meant to be shared by servlets and external processes. A servlet could be used to manage and accumulate the state, whereas a J2EE server could be used to execute the transaction on the accumulated state. The idea is to implement the State filter and let the architecture manage the state. The `StateManager` interface is defined as follows:

```
public interface StateManager {
    public State getEmptyState( String url, String windowName);
    public State copyState( String stateIdentifier, String url, String windowName);
    public State[] getStateWindowName( String windowName);
}
```

The method `getStateWindowName` is used to retrieve an array `State` interface instance based on the name of a window. In the `StateFilter` class implementation, the method is not used because the method is intended to be used by some other processor carrying out some application logic. The method `getEmptyState` returns an empty `State` instance based on the URL and window name. The method `copyState` is used to transfer the state of one state instance to another. The method `copyState` might do a physical copy from one `State` instance to another `State` instance. Or the method `copyState` might do an in-place copy. It depends on the implementation of `StateManager` and is kept flexible for diversity purposes.

Pattern Highlights

The State Navigation pattern is used to solve the web application usability problem associated with HTTP `POST` and with the inconsistencies of running a web application using multiple web browsers. Using the State Navigation pattern, you can separate the state of an HTML page from the HTML page. With a separation, it is simpler to manage and accumulate state that can be used by a process to execute a single transaction. The State Navigation pattern requires active participation by the programmer to make everything work and as such could be prone to problems.

The following are the important highlights of the State Navigation pattern:

- The pattern is used to associate a state with an HTML page.
- The associated state is in most cases nonbinding, and therefore, if lost, will not cause an application malfunction. In the worst case, a lost state results in the user having to reenter the data.
- HTML frames, when used extensively, may pose a problem for the pattern because the way that the browser manages navigation is modified and typically frames are given a name. Normally frames are not problematic, but if HTML frames are used, you should build a prototype so that there are no surprises.
- The pattern makes it possible to build applications that are transaction friendly because the state is cumulated by a state manager and can later be referenced as single action.
- The pattern provides a consistent user interface because posting the data is a separate step that is not part of the web browser's history. This solves the problem of posting data again when navigating HTML pages based on the history.
- The window name is a physical window name but could be used as an application grouping. For example, if a window is popped up, a window name could be reused, creating a relation between two separate HTML windows without sacrificing the ability to try out permutations of a form.