In this chapter we use the Flight Information application to learn how Seaside saves user data as part of a web application. But first, some background on the problem…

The HTTP protocol (which is used by web browsers to communicate with web servers) is by design very light-weight and intentionally avoids keeping any information that would tie one page request to another page request. This worked well when the web served static documents (the web was initially designed for the academic research community). Unfortunately, this does not work so well for complex applications such as e-commerce or travel reservation where we need to keep track of items added to a shopping cart or flights selected.

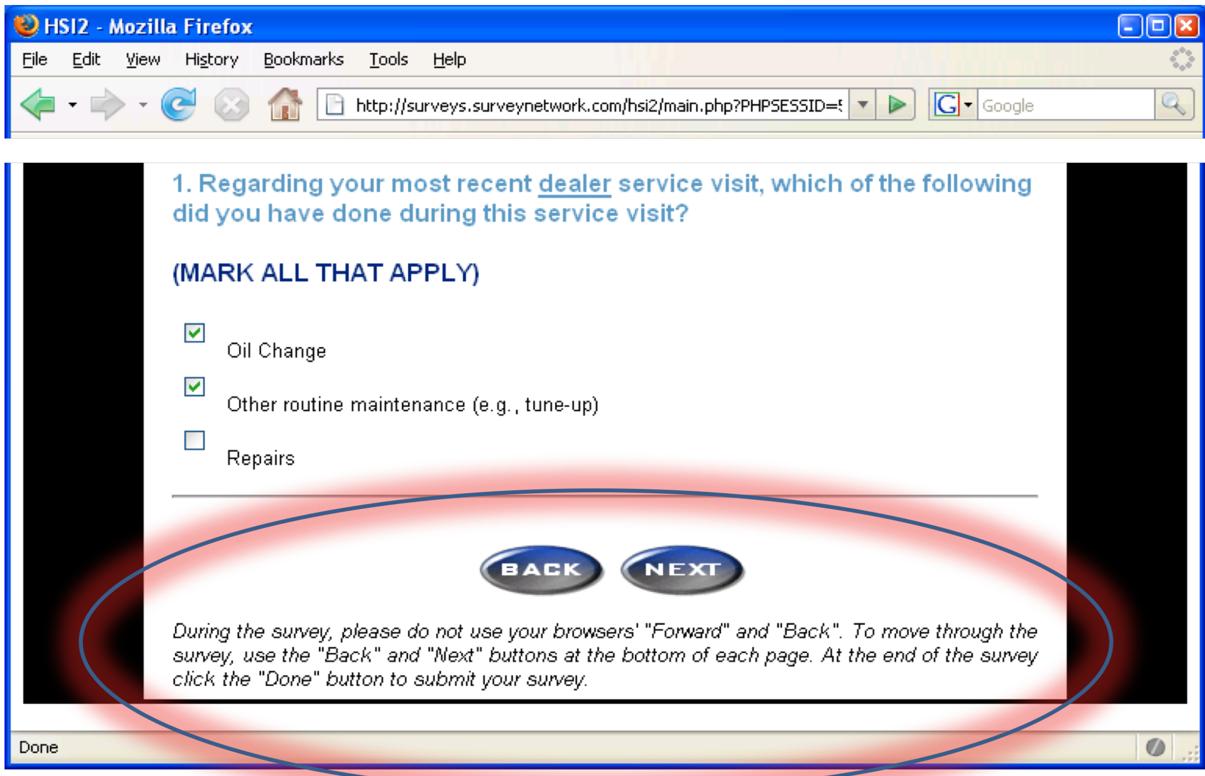
Various strategies are available to preserve state from one page to another. One approach is to add hidden fields to forms. When the form is submitted to the server, the hidden fields can contain saved data. This is okay if a form will be submitted, but doesn't work if the user clicks on a link (technically, the hidden field approach works for POST requests, but not for GET requests). Another approach is to use cookies. This works for both GET and POST as long as the user has not disabled cookies in the web browser. Also, the same cookie is sent for every request to the same server, so if the user has two tabs in the same browser, any changes in one will be visible to the other. A third approach is similar to "URL Rewriting" but instead of making the URL simpler or prettier, data is added to the URL (making it uglier).

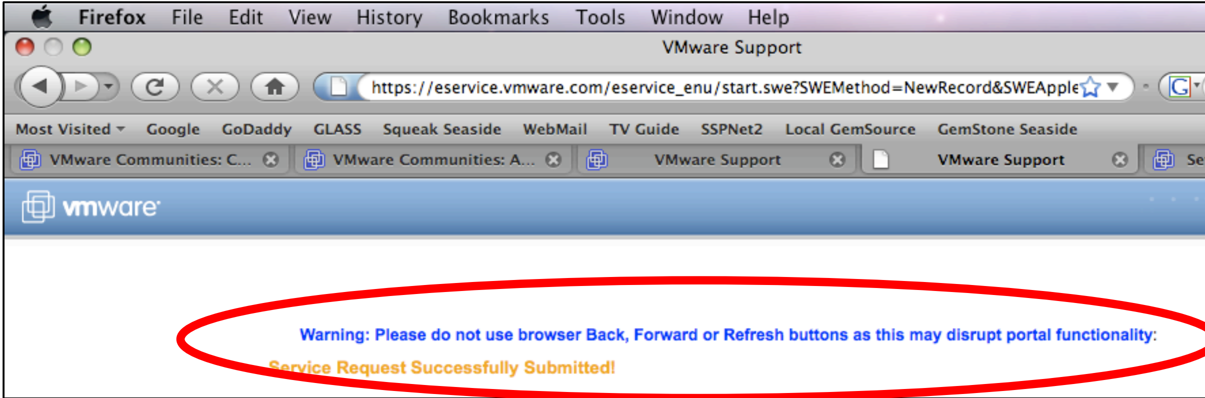
In any case, one needs to consider what data is sent to the browser to be returned by the client (whether in a hidden field, in a cookie, or in the URL). If actual user data is encoded then nothing needs to be remembered on the server. This scales very nicely, but opens the application to hacking. (See the PayPal example below.)

The more secure alternative is to store user data on the server and send a *session key* to the client to be returned with subsequent requests to the server. With user data on the server it should be difficult for a hacker to modify server data other than through the web application. If the session key is hack resistant (say, a large random number), then there is little risk of someone hijacking another user's session. The down side of this approach is that the server must store data on each session and if there are a lot of sessions then there can be a lot of data. Further complicating this is that most users who start interacting with a web application will not complete the process (it is common to add a product to a shopping cart and not complete the purchase). The application then needs to decide when to expire stale sessions. In order to handle a lot of user sessions, the application needs to be able to store and quickly retrieve a lot of data.

 Copyright © 2016 by GemTalk Systems

One significant problem of storing user data on the server with a session key on the client is that the user could use the browser's back button to change the displayed data without communicating to the server that the user is looking at different data. Following are a couple screen shots of commercial web applications that instruct the user not to use the browser's back button. This seems like a rather clumsy solution to the problem.





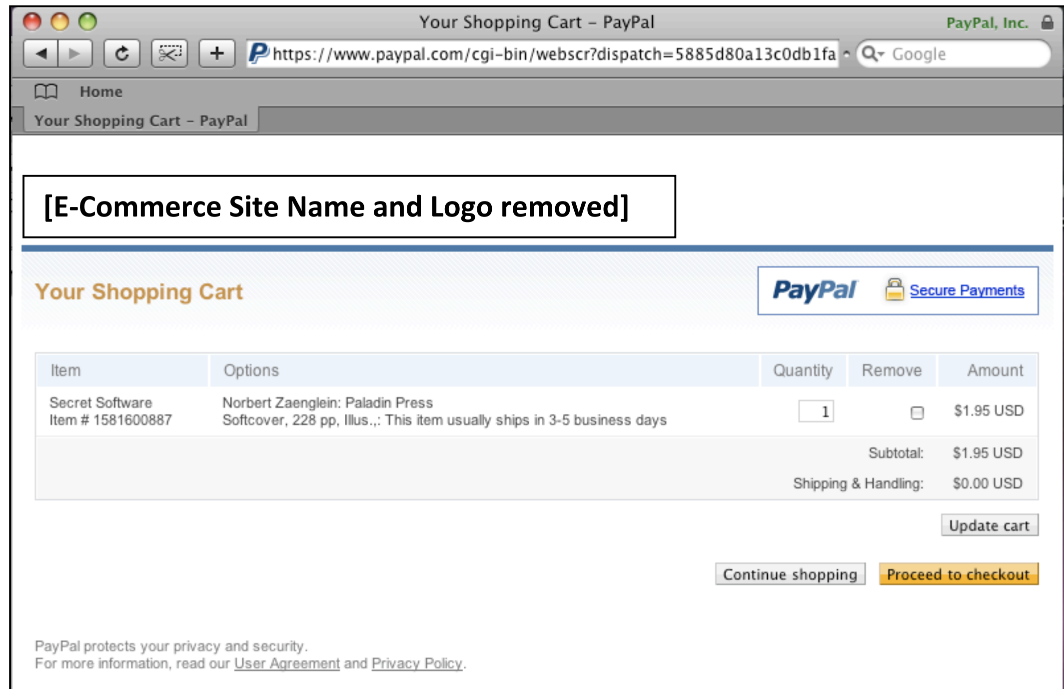
We will continue our Flight Information application to examine this problem and how Seaside addresses it.

 Copyright © 2016 by GemTalk Systems

A particularly striking example of sending critical data in an easily hackable manner is found in “Advanced Professional Web Design” by Clint Eccher (Charles River Media, 2007). Chapter 3 is titled “Understanding E-Commerce Functionality” and on page 72 describes how to submit a shopping cart to PayPal using a URL with encoded data. The example link is

```
<a href="https://www.paypal.com/cart/add=1&business=...&
item_name=...&item_number=...&amount=17.95&..."
target="_new"><img src="add_to_cart.gif"></a>
```

When I first read Eccher’s book I tried the link with different values for the amount and there was every indication that PayPal would complete the order with the revised amount. Following is a screen shot of the shopping cart with a price of $1.95 instead of $17.95:



Note that the problem displayed here is not with Eccher’s description or his sample code, but with the credit card processing design he describes. (If using this approach it would be important for the seller to check the price paid by the customer before shipping the merchandise.)

Copyright © 2016 by GemTalk Systems

1. At this point our Flight Information application allows a very trivial way of searching for flights but does not offer any way to select a displayed flight. Before we begin the following exercise, make sure you have the Seaside One-Click Experience running and that you can browse the Flight Information application in a web browser.

2. First, the method FlightInfoComponent>>#renderContentOn: (this is a typical Smalltalk way of identifying a class and method) is getting a bit long. We will start with copying most of the code to a new method and adding a line break after the earlier/later links. Next we will add a new anchor to report the selected flight.

```
renderChangeTimeLinksOn: html

   html anchor
      callback: [model addHours: -30];
      with: '<- Earlier'.
   html space.
   html anchor
      callback: [self later];
      with: 'Later ->'.
   html break.
```

```
renderContentOn: html

   html heading: model.
   self renderChangeTimeLinksOn: html.
   html anchor
      callback: [self inform: 'You selected ' , model printString];
      with: 'Book flight for ' , model printString.
```

3. In your web browser refresh the FlightInfo page to get the new elements, click the 'Later ->' link a few times, and then click on the 'Book flight' link. Your web browser should show a page showing the date, time, and price for the flight. Clicking the OK button takes you back to the flight information page.

4. Now click the 'Later ->' link a few times, note the price, click the browser's back button once, note the price, and then click the 'Book flight' link. Compare the price displayed on the inform page with the two prices. Because data is saved on the server, and because the server did not know that you clicked the back button, the 'Book Flight' link showed you the data saved on the server, not the data displayed in the browser.

5. We can see the same problem if we have two browsers or tabs. In your current web browser, right-click on the 'Later ->' link and select the menu option to open in a new window. In the second window click the 'Later ->' link a few times. Then switch back to the first window, note the displayed date/time/price, and then click the 'Book flight' link. Note that the 'wrong' flight was booked. Again, this happened because there is one FlightInfo instance that is kept in the 'model' instance variable of the one FlightInfoComponent being displayed on multiple windows.

Copyright © 2016 by GemTalk Systems

6. To address this problem, Seaside gives you a way to identify objects for which the state at the time a page is rendered should be preserved and associated with that page. When a user interacts with that page in the future the state of the object is restored to how it was when the page was originally rendered. Thus, while there is still only one FlightInfo instance, its state (when & price) can be preserved by Seaside and restored if needed for a future request. To see this simply add a method to FlightInfoComponent:

```
states

   ^Array with: model.
```

7. After saving the new 'states' method, return to your web browser, click the 'Later ->' link a few times, click the browser's back button, note the displayed date/time/price, and then click the 'Book flight' link. Note that the correct flight is displayed.

   Seaside's approach of saving state on the server is powerful and easy, but it does come at a price. Now we are keeping information on the server not just for every user (or session), but for every page served to every user. This is a manifestation of the old adage that "There's no such thing as a free lunch." The nice thing is that it is available, and you can choose to use it if your application would benefit from this help. Also, keep in mind that until you measure performance (either space or speed) it is probably a mistake to be overly concerned.

8. Save your Pharo image before going on to the next chapter.

 Copyright © 2016 by GemTalk Systems