Stage Driver: a design pattern for multistage forms

Pablo Montesinos
pmontesi@cs.uiuc.edu
Department of Computer Science, University of Illinois at Urbana Champaign
201 N Goodwin Ave, Urbana, Illinois, 61801, USA

Abstract

The Stage Driver pattern simplifies the creation of multistage forms in JavaServer Pages (JSP). Stage Driver introduces an extra control page that handles and validates inputs from the user and chooses which stage the user should be presented with. By clearly specifiying the role of each compononent in the system, Stage Driver also helps determining where the transaction bounds are.

Introduction

Many web applications require users to register for an account in order to use them. If the registration process collects little information, a single form is enough. For example, on-line news sites ask their users to register so their comments are not marked as anonymous. Creating an account in on-line news sites [3] is usually straightforward: users are only asked for their nickname, a valid email address and their time zone.

Other web applications require much more information from the user. The data is partitioned so that related pieces are presented in the same page or stage. The benefits are twofold: validation of connected data can be done simultaneously (e.g. zip code and address) and the user does not need to remember what he answered a couple of stages ago (e.g. Did I enter my offices zip code or my homes?).

For example, a free, web-based email provider [1] could divide the registration process into four stages:

- *Account information stage*: the user would be asked for the email address he wants to use, a password, a confirmation for the password and a password reminder question.
- *Personal information stage*: the user would need to fill out his name, last name, gender, etc.
- *Subscriptions stage*: because the free email provider needs to pay the bills, the user would be asked to choose to which online magazines he would like to be subscribed to.

- *Contract Stage*: the systems would show the user a contract, and the user would be required to click the *I accept the conditions* box.

Figure 1 shows that process. While some of the fields can be validated at the client (e.g. do the password and the confirmation match?) each stage might need to access a database to validate some of the users inputs (e.g. is there any user with that name already?). In each stage, if the system detects an error in the input data, the user is not allowed to continue until he fixes it. Finally, once the user accepts the terms of use, all the information is stored in the database, and the user is shown a welcome page.

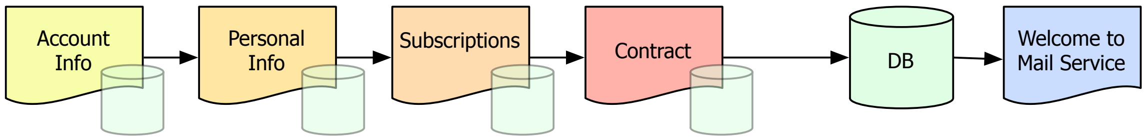

Figure 1: Multistage form process

Context

Web applications that use multistage forms to acquire data from the user.

Problem

Error validation and presentation are key issues in multistage forms. They are tightly related because the system has to present the errors to the user as soon as they occur (users cannot proceed to the next stage until the current one is error-free). When an error in a form is detected, the system could ask the user to input all the data for the stage again. Or it could just show an error page and ask the user to press the browsers back button. A more interesting solution (and less frustrating for the user) is to show the same form with all the data the user entered, except for those fields that were wrong.

Presenting the data the user introduced in the first time is very easy if the system uses client-side validation; after all, no request leaves the client until the local checks are completed. Client-side validation is fast, as there is no need to wait for the data to go to the server, be processed and return. Its greatest strength is that it reduces the server load by performing some checks before the data arrives at the server. For example, the client could check that the zip code the user entered is 5 digits long. However, it would nott be able to check if the zip code 61801 belongs to Urbana, Illinois (unless we make all the clients that intelligent, which is not a good solution either). In addition to accuracy, client-side validation also lacks security: it could be possible for a user to modify the form for a stage, removing its validation code. He could then use the modified form to post malicious or malformed data back to the server. Although server-side error validation/presentation provides higher security and accuracy, is more cumbersome because it requires a more intelligent state machine that will make the system slower and harder to develop.

Since the information acquired during the process is not considered complete until the last stage is validated, multistage forms need to be atomic. Either the user completes them or not. Consider the email service from the example section. Keeping the registration process temporary data for longer than is necessary can be expensive if the site is very popular. Another reason why temporary data should not be allowed to persist is to avoid denial of service. For example, the first stage of the mail provider asks for a username. If a user in the third stage decides to abandon the registration process, no other user would be able to use that username until the system discards the temporary data. Therefore, it is necessary to prevent the users from bookmarking any intermediate page because temporary data needed to resume from that point might no longer exist.

Finally, concurrency and session control are important issues. In the case of single-stage forms, there is no problem when two users register at the same time. However, problems can arise if two users try to complete a multistage registration process at the same time. Thus, some form of session control must be implemented.

Forces

Therefore, when using multistage forms in web applications, the following forces need to be resolved:

- *Client-side validation*. Clients might validate some information, so it is filtered by the time it gets to the server.
- *Server-side validation*. The server must validate (at least) the critical data.
- *Error presentation*. When there is an error in the data the user entered, the system must show the same page with the error fields cleared and a message indicating what the error was.
- *Data Persistence*. Already validated, temporary session data should not be retained any longer than necessary.
- *Separation of control logic from the structure of the pages.*
- *Simplicity*. The control code should be simple to minimize application complexity.

Solution

The Stage Driver pattern eases the management of the data entered by the user through multistage forms by creating a single file for each stage in the process (the stages) and an extra file (the root) that will understand in

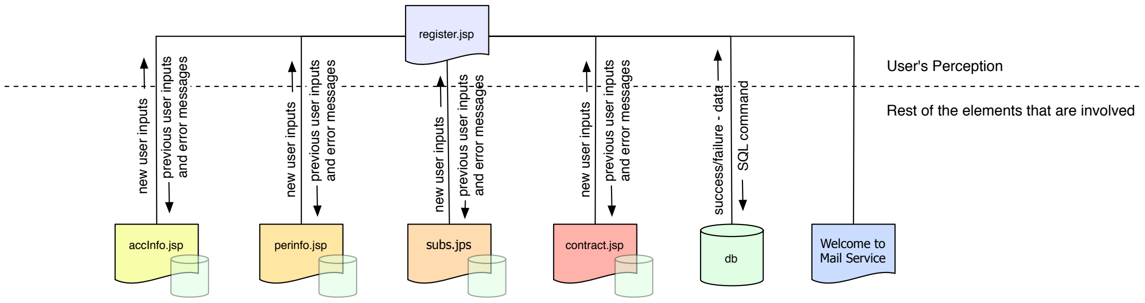
register.jsp
User's Perception
Rest of the elements that are involved
new user inputs
previous user inputs and error messages
new user inputs
previous user inputs and error messages
new user inputs
previous user inputs and error messages
new user inputs
previous user inputs and error messages
success/failure - data
SQL command
accInfo.jsp
perinfo.jsp
subs.jps
contract.jsp
db
Welcome to Mail Service

Figure 2: Data flow between the stages and the root.

which part of the process the user is and will display the appropriate stage.

The stages are very simple files; they display a form and, in case of error, the error message that the root passes to them. Any local data validation code should go in the stage files. The target for the form is the root page.

The root page is responsible for driving the user through the sequence of files that are part of the process. Figure 1 shows a linear multistage process although it is possible to have a non-linear one. However, the root does not scale very well if the graph of the possible paths the user might follow is very complex. The root page also takes care of any server validation. If the server returns a success, it moves on to the next stage. If not, it loads the same stage file and asks it to display an error message as well as the data in the fields (except the one that caused the error). Figure 2 shows the data flow between the stages and the root.

Stage Driver does not impose any method for temporary data storage. Developers are free to choose which method to employ. However, beans have been used successfully with this pattern and they will be used in the example later in the paper. Also, Stage Driver does not impose any session control method. The example stores temporary data in the session object for simplicity.

Implementation

To implement the Stage Driver pattern, carry out the following steps:

1. Create the root page. The root will not generate any output, but will handle the state depending on the data in the session object.

2. Create a different page for each stage. Name each of them with a unique identifier so the root page can distinguish which form submitted the request. Make the root page the target for the form.

3. Write, in each stage, the validation code for those fields that can be locally validated.

4. Write, in the root page, the stage selection code. The stage selection code will examine the request to determine

which form initiated it.

5. Write, in the root page, the server validation code for each stage. Once the selection code deduces which stage executed the request, the validation code is executed. If it finds no error in the data, then the root loads the following stage. If there is an error in any of the fields, it reloads the same stage and shows an error message indicating which fields were wrong. Those fields that were correct should contain the data the user wrote.

6. When the last stage is considered valid, proceed with the insertion of the data in the database.

Sample Code

We now show an example for a two-stage registration process using JSP. The example consists of two stage files (called stage1.jsp and stage2.jsp) and a root (root.jsp). Figures 3 and 4 show the source code for the first and the second stage, respectively.

```
1  <html>
2   <head><title>Account information</title></head>
3    <body>
4      <%=(request.getParameter("error")!=null)?
5           request.getParameter("error") :"Fill this:"%> <br/>
6      <form name="formstage1" action="root.jsp" method="post">
7       <!-- form declaration-->
8       <input type=submit value="continue!"> <!-- the button -->
9      </form>
10   </body>
11 </html>
```

Figure 3: Source Code for stage1.jsp

```
1  <html>
2   <head><title>Credit Card Information</title></head>
3    <body>
4     <%=(request.getParameter("error")!=null)?
5          request.getParameter("error"):"We want your money:"%> <br/>
6      <form name="formstage2" action="root.jsp" method="post">
7       <!-- form declaration-->
8       <input type="submit" value="continue!"> <!-- the button -->
9      </form>
10   </body>
11 </html>
```

Figure 4: Source Code for stage2.jsp

The sample code for the first stage is straightforward. Lines 4 and 5 manage the error message that the root passes to the stage: if there is an error, the stage shows it. Otherwise, it just asks the user to fill in and submit the form. Figure 4 shows the code for the second stage, which is very similar to the first one. Each stage is uniqely identified by a name (formstage1 and formstage2, respectively), so that the root can know from which stage the requests comes from. Finally, it is often a good idea to change the title of each page so it reflects the purpose of the stage. Recall that the user only sees the root's address.

```
1 <% if (request.getParameter("form") == null)  { %>
2     // It's the first time we load it
3      <jsp:include page="stage1.jsp" flush="true" />
4 <% } else {%>
5     boolean isValid = true;
6     String error = "";
7     if (request.getParameter("form").equals("formstage1")) {
8          // validate stage1 fields
9          if (isValid) { //we load the next form
10           %><jsp:include page = "stage2.jsp" flush = "true" />
11         <% } else { %>
12            //the user did something wrong.
13            //We present the same page
14            <jsp:include page = "stage1.jsp" flush = "true" >
15            <jsp:param   name = "error" value = "<%=error%>" />
16            </jsp:include>
17         <% }
18     } else
19       if (request.getParameter("form").equals("formstage2")) {
20          //We check the second form
21          if (isValid) { %> //everything is ok
22             <jsp:include page = "done.html" flush = "true" />
23       <% }else { %>
24         <jsp:include page = "stage2.jsp" flush = "true" >
25         <jsp:param   name = "error" value = "<%=error%>"/>
26        </jsp:include>
27       <% }
28      }
29    }
30 %>
```

Figure 5: Source Code for stage2.jsp

Figure 5 shows the source code for the root page. As expected, it is more complex than the code for the stages. Lines 1 – 3 detect whether the user is requesting the root page for the first time. If that is the case, the root begins the multistage process by loading the first stage. Otherwise, line 7 checks whether the request came from stage1.jsp. If it did, the root validates its data in line 8. Notice that we do not show any code for data verification because this pattern does not impose any restrictions on how to store the input data. For example, it is possible to use a bean or just save the field values in the session object.

Line 10 displays the second stage if the validation process for the first stage succeeds. If not, lines 12 – 16 display stage2.jps and pass the error message as a parameter. The rest of the code is straightforward. Figure 6 gives an overall view of the logic in the root page.

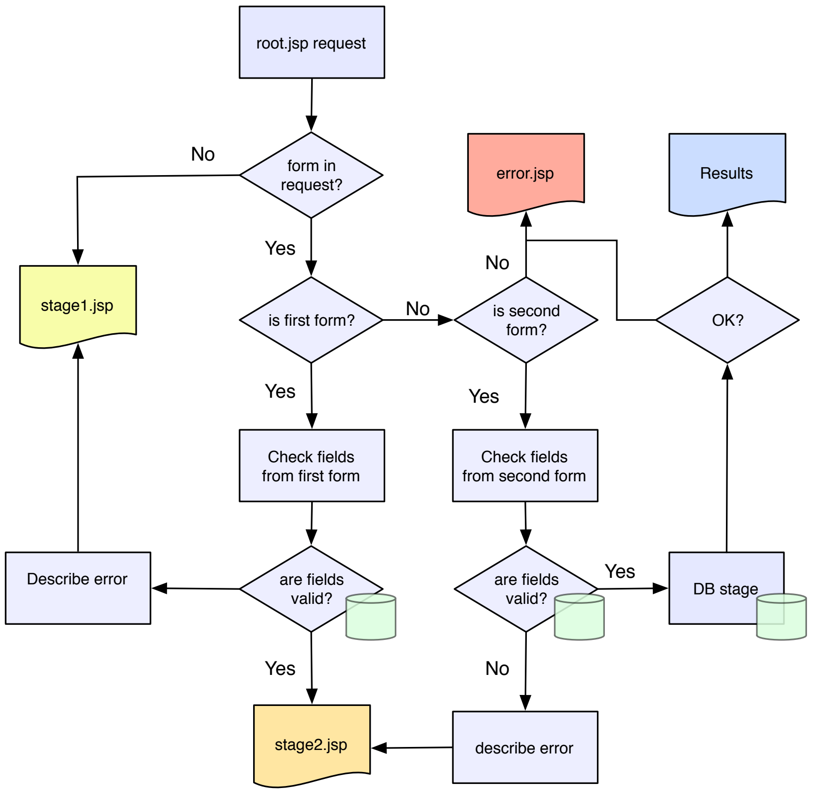

Figure 6: Multistage form process

Consequences

The Stage Driver pattern has several benefits. First, it hides the steps of a multi-stage form from the user. As a result, users are forced to follow the stages sequentially. Second, multistage processes become atomic. Either the user completes it (and all the required information has been already validated) or not. Third, it separates logic and presentation. In the Stage Driver pattern, the logic is kept in the root page, well isolated from the presentation code, which stays in the stages. Finally, by removing the control code from the stages, it simplifies their development and maintenance.

However, Stage Driver also imposes some liabilities. For example, an excessive use of this pattern can increase the distances of your web applications. Some user interface patterns proposed in [4, 5], like *Clear entry Points*, persuade developers to keep distances short. That is, multistage forms should have as few forms as possible to prevent users from abandoning the process without completing it. Another liability is the scalability and complexity of the root. As long as the state machine is relatively simple, the simple root page proposed by Stage Driver is enough. Large multi-stage forms or complex-non-linear paths may require a different approach, — maybe a hierarchy of roots.

See Also

Some of the patters in [2] are related to Stage Driver. For example, Echo Back could be used in each stage. Some of the fields in the stages that cant be validated until the very end will follow the Deferred Validation pattern. Session [6] should be used to prevent the data from one user to interfere with other users data.

Acknowledgments

The author would like to express his thanks to Brian Greskamp for his valuable comments and Federico Balaguer for his great job during the shepherding process.

References

[1] M. corp. Hotmail. http://www.hotmail.com.

[2] W. Cunningham. The checks pattern language of information integrity. pages 145–155, 1995.

[3] O. S. T. Group. Slashdot: News for nerds, stuff that matters. http://www.slashdot.org.

[4] J. Tidwell. Ui patterns and techniques. http://time-tripper.com/uipatterns/.

[5] J. Tidwell. *Designing Interfaces*. O'Reilly, 2005.

[6] J. Yoder and J. Barcalow. Architectural patterns for enabling application security, 1997.