Controlling Page Changes in AWS Applications

1 Document History

Date	Version	Change Description	
8/4/09	1.0	Initial Entry	
8/4/09	1.1	Initial edits	
8/5/09	1.2	Added application description content	
8/6/09	1.3	More edits added Glossary	

2 Table of Contents

3 Introduction

This document describes methods for controlling all page transitions within a NET+OS AWS-based application. The solutions and examples described herein use a combination of JavaScript and Advanced Web Server (AWS) Toolkit function calls.

3.1 Problem Solved

For many simple embedded web applications, based on NET+OS's Advanced Web Server, giving the AWS engine decision-making power over the next page to which the application transitions is inadequate. On the other hand, there can be times where the default page flow is not adequate for your application. For example, you might want your application to catch non-secure accesses (http:) and relay those accesses to secure accesses (https:). In another case, suppose while processing a web request, an error occurs (can't open a file, can't allocate memory….) how do you alert the user of this pathological case? The contents of this paper and its associated sample application, explain handling these and other cases with web page transitions controlled by your application.

3.2 Audience

The audience for this white paper is users with both web development experience and NET+OS development environment and specifically NET+OS AWS development experience. The amount of JavaScript used in this document will be small, meaning that extensive JavaScript knowledge is not required.

3.3 Assumptions

This paper assumes the reader has access to a version of the NET+OS development environment of at least V7.0. More recent versions, in the area of V7.3 and later are more desirable but not required. The techniques shown here may be compatible with the NET+OS development environment V6.3 but we have not specifically tested these techniques in that environment.

3.4 Scope

- The scope of this paper is to describe methods in html, JavaScript and AWS Toolkit function calls for transitioning web page flow. That is altering the "normal" flow of web pages within a web application.
- In the sample application, there is also extensive use of NET+OS's file system using C library calls. Though the application uses these calls, there will be no description of usage models for these calls in this paper.

The following subjects are not within the scope of this white paper:

- An extensive discussion of JavaScript coding techniques or general uses of JavaScript
- Any discussion of Ajax techniques using JavaScript
- Any discussion of Ajax techniques using XML

- A general discussion of developing AWS applications in a NET+OS development environment
- A general discussion of developing applications in a NET+OS development environment
- Any discussion of C coding techniques
- Any discussion of TCPIP
- Any discussion of NET+OS's file system and its usage except where it is used to clarify the attached sample application
- Any discussion of web page development and/or html
- Any discussion of the PBuilder utility

3.5 Theory of Operation

There are four reasons (that I can think of at this writing) why I might want my application to more directly control the flow of web pages within a web application. Specifically, within a NET+OS AWS web application. These reasons are as follows:

- Redirecting a browser request from unsecure (http – port 80) to secure (https – port 443).
- Displaying a "successful outcome achieved" page on the conclusion of an operation.
- Displaying an "unsuccessful outcome achieved" page on the unsuccessful conclusion of some operation.
- Closing the browser window after a user hits an "exit" button.

The first page flow change can be achieved by checking the "protocol" of the browser request and either redirecting or not redirecting the request, based on the needs of the application. The method to be described for performing this operation is the JavaScript method window.redirect().

The second and third page flow changes can be achieved in the application's stub function, supporting the web page. If a get operation is successful, the web page to which AWS is returning, presumably, will display some additional information that was not available before the get operation was completed. This might be an IP address, an account balance or a name, for example. If a put operation is complete, there is nothing returned by the AWS engine, to the browser. A get operation might immediately follow the set operation. This get operation might update some data that might instruct the user that something good has occurred. On the other hand, if a set operation fails, it is infinitely more difficult to alert the user of the failure. To facilitate alerting users of the success and failure of operations, this paper describes and demonstrates the use of the RpSetNextPage() and the RpSetRedirect() NET+OS AWS toolkit function calls. There is also an RpSetNextFilePage() function call that assumes the page contents are in the NET+OS file system as opposed to being built into the AWS application. Though, RpSetNextFilePage() is not discussed in this paper, I bring it up to make the reader aware of its existence. Please see the Advanced Web Server Toolkit Users' Guide for more information about the function call RpSetNextFilePage().

The final page flow is simply performed by calling the JavaScript method window.exit(). The user will be informed, by the browser that the application wishes to close the browser window. The user is then asked if he'd like to proceed. Hitting the yes button causes the browser to close. For applications where security is important having the user close the browser window ensures that a non-authorized user can not gain access to the system, as a login is required.

4 Basics

The following section describes, with a little more detail, the use of the methods and functions described above. Later, in this white paper, actual code examples in the sample application will be referenced.

4.1 Catching and redirecting unsecure browser connections

In this case, my application accepts both unsecure (http:) and secure (https:) accesses. I do this for the convenience of my users. In this way, if they accidently type in http://my_device_id, they do not get some potentially useless message. Instead my code redirects their browser to the secure port.

The JavaScript method window.location, provides the URL of the browser access. The method window.location.protocol provides the first component of the URL, that being http:, vs. https:, vs. file:.

So in JavaScript you might provide some code to do something like this:

```
switch(window.language.protocol)
{
   case "http:"
      // redirect to https:
      break;
   case "https:"
      // do nothing
      break;
    default:
      // maybe put out an error message

}
```

At this point, there are two things you might need to know. One is how do I get to the JavaScript function, the other is how do I perform the redirect.

To get to the JavaScript code, we recommend using an onLoad attribute in the body tag of the first web page of your AWS application. The onLoad attribute directs the browser to run the JavaScript function as soon as the page is loaded but before actually doing anything, such as calling AWS stub functions. We'll dive into this further when we study the sample application.

To perform the redirection, you'll need to do two things. First you'll need to change the protocol portion of the browser's URL from http: to https:. Using string manipulation and methods in the window.location object, this is doable. Once you have created your updated URL, use the JavaScript method window.location.replace(). The parameter placed between the parentheses is the newly created URL. Again, I'll go into details when we look at the sample application.

4.2 Redirecting web page flow in NET+OS application code (C code)

Suppose your NET+OS AWS application web page has directed the user to a function associated with a web page, or a web page's components. You attempt to open a file and the file open function fails. Or your function attempts to allocate some temporary (or permanent) memory, but the malloc call fails. Or you are given a user name and password for accessing the system access database. You make the call but the return code indicates that the user/password pair does not exist. How do you alert the user back at the browser? It is not easy as the options you have for returns are very limited.

In your application code, when you get to a code path that is to manage an error, you'll want to make the following function calls:

- RpSetNextPage()
- RpSetRedirect()

RpSetNextPage(). The parameters to RpSetNextPage() are the AWS data structure (the return from a call to RpHSGetServerData() followed by the page object pointer. You will find this declared in the .c file of your AWS application. This is the .c file as opposed to the _v.c file. Following the call to RpSetNextPage() you'll need a call to RpSetRedirect(). The single parameter to RpSetRedirect() is the AWS data structure. Again, that is the return value from a call to RpHSGetServerData() . When your code path returns to the AWS engine, the browser is redirected to the page pointed to in the call to RpSetNextPage().

If you wish, and this is how successful operations are handled in the attached sample application, you can also call RpSetNextPage() and RpSetRedirect() following a successful operation.

It goes without saying that the pages that you are redirecting to must be written in html, JavaScript or some related language, and must be run through the Pbuilder utility for inclusion in the application.

4.3 Handling your exit

This section presumes that you are giving your users the ability to exit the web application via a button. To do so you'll want to add a JavaScript call to method window.exit(). So the question is how do I get from my button to a page to some JavaScript? As we did before, you'll want to create a page with very little on it. It will

need, though, a body tag and that body tag must have an onLoad attribute. The onLoad attribute needs to call a JavaScript function whose only purpose is to call the JavaScript method window.close(). Upon doing this, the user gets a box, stating that the application wishes to close this window. Presumably the user clicks the OK button and the browser window closes.

5 Example Application Explanation

The sections above have provided you with a high level view of how to more closely control the web page flow in a NET+OS AWS-based application. In this section, we will look at the code in the sample application and look at how the functions and methods described above are actually used. Since most of the calls to RpSetNextPage() and RpSetRedirect() are repetitive, I will only go through enough representative samples to allow the reader to fully understand their use. Get Source Here

5.1 front_page.html

The first file we will explore is front_page.html. This is the page that your browser displays by default when browsing to http://<your device id> or https://<your device id>. To ensure that this page is the default web page in your application, ensure that this web page is the first web page listed in file list.bat or PBuilder.pbb if your are developing under the ESP integrated development environment.

The html code in this file is fairly mundane except for the fact that the body tag has an onLoad attribute. The onLoad attribute points to JavaScript function checkProtocolGotohttps(). So as soon as your browser pulls up html page front_page.html, this function will be executed. What is interesting is the JavaScript code of function checkProtocolGotohttps(). You'll notice a switch statement that is very much like switch statements in C and C++. The switch statement first checks for https: in upper and lower case. Please note that, from what I have seen, you need to check for https + ":". If your case statements look for https without the ":", you will not find them. For https:, we want to do nothing. If we find upper or lower case http + ":" or nothing, then we want to redirect the browser to the secure site. To do this, the window.location.replace() method is used. But before calling the method, we must create the new URL. To do this, you concatenate the window.location.host and window.location.pathname members together (in this order). You add in front of this https://. Please notice that I have included both a ":" and a set of "//". So to be clear, the new URL is made up of "https" + ":" + "//" + window.location.host + window.location.pathname . This whole new URL is used as a parameter to the window.location .replace method. When you leave the JavaScript function, you will see the browser redirect to this new URL.

Last, please notice that front_page.html uses an anchor html tag with a href attribute to transition to page createAccount.html.

So the page transition to the https page is accomplished using the window.location.replace method while the page transition to createPage.html is accomplished using an html anchor tag with a href attribute.

5.2 createAccountPage.html

createAccountPage.html looks like any standard html page set up with pbuilder tags and ready to be converted to C using the pbuilder utility. Its purpose is to solicit a user name and password for creating a new bank account. A question you might ask is if the account creation attempt fails, how would we know? Hang on; we'll explore this in a few pages.

5.3 optionsPage.html

The "Please log in" prompt on front_page.html is tied to an anchor tag and a href attribute. The href attribute points the application to optionsPage.html. This first transition to optionsPage.html will solicit a challenge screen. The question is why. If you look at the very top of file options_page.html you'll see an RpPageHeader AWS toolkit tag with an RpAccess attribute whose value is Realm1. This tag and attribute combination means that users must belong to Realm1 in order to access this page. In this sample application, there is only one realm and all users belong to it. In addition, there are two issues that adding this tag and attribute present. First, all pages that might be transitioned to must also have this tag and attribute set. One reason for this is that some of the stub functions that make up this application rely on the AWS function call RpGetCurrentUser. RpGetCurrentUser only returns authenticated users. If you transition to a page without an RpPageHeader and an RpAccess pair, the new page will have no authenticated users. This may break your application. You need to keep this in mind when designing your application. You'll notice that front_page.html, createAccountPage.html and accountCreated.html do not have this RpPageHeader tag. This is because, these pages are accessed before the user can log in (before he has an account or before he has logged in with his account). Once the application transitions to optionsPage.html, it and all other pages have an RpPageHeader tag with an RpAccess attribute.

5.4 Stub functions – front_page_v.c

The remainder of the html files are fairly mundane. If you have done any AWS development, you'll recognize the AWS toolkit tags and the references to stub functions. So we'll now explore the stub functions in file front_page_v.c. Ensure that you are looking at the version located in directory \pbuilder in your application.

The first thing to notice is that at the top of the file, there are five extern references to rpObjectDescription objects. These refer to page objects. The page objects are actually declared in front_page.c in the \pbuilder\html directory. In calls to RpSetNextPage you do not present URLs as parameters. Instead you use pointers to rpObjectDescription objects. So that might be &PgoverdrawnPage. Please notice the '&', as it is important. To get the rpObjectDescription object for a particular page that has been processed by the pbuilder utility, look in the front_page.c file (or the .c file for your particular application).

5.4.1 getTheAccount()

Function getTheAccount, gets the account name for use by the system. In our case, the account name is the user name of the currently authenticated user. First notice the call to **`RpHSGetServerData().`** This returns a void * that is used by almost all AWS toolkit functions. If the void * is equal to NULL, this generally means that the AWS is not running. No AWS functions will function normally, should this call return NULL.

Next notice the call to RpGetCurrentUser(). Notice that the parameter is the returned void * from the call to RpHSGetServerData(). If RpGetUserData() fails (there are no authenticated users currently available), the application makes two function calls. The first is to RpSetNextPage() and the second is to RpSetRedirect(). The parameters to RpSetNextPage are the retuned void * from RpHSGetServerData() and a pointer to an rpObjectDescription that we talked about in the last section. These two calls cause the AWS engine to force the browser to do a redirect to the page described by the pointer to rpObjectDescription. This use of RpSetNextPage and RpSetRedirect is a method used in most of the remaining stub functions.

5.4.2 SetThePassword

If the application is able to set up a username and a password for the account and is able to create a file named username.password, then the user's account is considered successfully accomplished. If the password for the account (user name) cannot be found or for some reason the file cannot be created or written to, then the operation is unsuccessful. If you look to the end of function setThePassword, you see calls to RpSetNextPage and RpSetRedirect, where the rpObjectDescription (pointer) used is & `PgaccountCreated`. This represents a transition to a page showing that the account creation was successful. All of the other such calls within setThePassword, represent transitions to pages telling the user that the account creation operation was unsuccessful. You'll remember that when we explored createAccountPage.html, I asked what happens when an operation fails. That is, how I notify the user of the failure. You use calls to RpSetNextPage and RpSetRedirect to send failure (or success) pages to the user.

5.5 exitPage.html

Since all of the other stub functions use methods that have already been described, I will not belabor this white paper with their descriptions. Instead I am going to jump to web page exitPage.html. Notice in this page the html body tag has an onLoad attribute and this attribute points to a call to JavaScript function closeThatWindow(). So when this page is loaded, this function will immediately be executed. The JavaScript function contains one call to the window.close() method. This method causes an alert window to appear from your browser announcing that the application wishes to close the window. If the user hits the ok button, the browser window closes. For a secure application, you probably want your exit button to perform such a task.

6 Conclusion

In this white we have explored different ways of performing page transitions. We have looked at using anchor tags with href attributes, JavaScript window.location.replace and the AWS toolkit set of RpSetNextPage() and RpSetRedirect(). In doing so, we have shown how to inform users of the success or failure of operations within stub functions. We have also looked at how to redirect a browser for (in our case) redirecting from an insecure URL to a secure URL.

When using RpSetNextPage, you must remember that rpObjectDescription pointers, must first be created as html files and then those html files must be run through the pbuilder utility. The name of the rpObjectDescription can be found by editing the .c file (not the _v.c file) AFTER you have run your html files through the pbuilder utility. As an example, the object description for exitPage.html looks like the following in file front_page.c of the attached sample application:

```
rpObjectDescription PgexitPage = {
      "/exitPage.html",
      PgexitPage_Items,
      (rpObjectExtensionPtr) 0,
      (Unsigned32) 0,
      kRpPageAccess_Realm1,
      eRpDataTypeHtml,
      eRpObjectTypeDynamic
};
```

Notice that the name of the object description (PgexitPage) is followed immediately by the URL for the page (/exitPage.html).

7 Appendix

7.1 Glossary of terms

- Ajax – Asynchronous JavaScript and XML – a method of developing web page applications using JavaScript and xml giving the applications a more Pc-based feeling
- AWS – Advanced Web Server – an embedded web server shipped as part of Digi International's NET+OS embedded operating system
- Html – hypertext markup language. The language used for creating web pages
- JavaScript – not to be confused with Java. A scripting language developed by Netscape. It is used by web developers to create more interactive web applications then what might be doable with html alone.
- NET+OS – An embedded operating system and development environment, available from Digi International
- tcp/ip – a suite of protocols, originally developed by DARPA. It is the basis for the internet.

