Application Engineering Documentation

Visitor Management in Maytag Home Solutions (MT192003)

Contents

Objective

Maytag chose Broadvision as the server platform for their Maytag Home Solutions website, which was coded by Giant Step (MT192003). This document explains how we used Broadvision's visitor API for site membership and our approach for recognizing visitors with cookies.

The description of the Giant Step approach is preceded by some sections about the general Broadvision API. The purpose of this is both to supply an adequate introduction, but also to remedy the fact that much of Broadvision's documentation is incomplete and inadequately thorough. Thus, this document has potential usefulness for anyone at Giant Step doing Broadvision programming.

This document was written by Greg Sandell (gsandell@giantstep.com). It's home is http://eroom.giantstep.com/eroom/giantstep/White/764_36

Broadvision API: Types of Accounts

How to create them

Account type	Broadvision code
	`var v = new BVI_VisitorManager;`
Member	`visitor = v.newVisitor(user_login_name, password);`
Guest	`visitor = v.newGuestVisitor(true);`
Transient	`visitor = v.newGuestVisitor(false);`

Connecting with an existing member account can be done by:

- `visitor = v.visitor(user_login_name)`

What they mean

Account type	Meaning
Member	Has a username & password, user has persistent profile data which they connect to when they log in. User's username/password uniquely identifies a single user. Has a record in both BV_USER and BV_USER_PROFILE. User reconnects with account/profile with a call to .visitor().
Guest	No username/password. Has record in BV_USER_PROFILE, but no record in BV_USER, no ability to log in. But user's profile data is persistent, and user can reconnect with their profile data just like a member. User connects with profile data with visitorsByQuery(), matching according to some field (BV_USER_PROFILE.EMAIL is typical); but this is not a unique association (more than one user could have the same .EMAIL value).
Transient	Has no database rows, either in BV_USER or

	BV_USER_PROFILE. Any visitor properties that are written are cached and saved for the session only.

Broadvision API: Visitor properties

Visitor Properties Defined

Visitor properties can be used to store any values that need to be available across all pages, and are associated with the specific visitor. Whether these exist in cache for just the duration of a Session (like Session variables in ASP), or persist across multiple sessions depends on the type of account (member, guest, transient) and whether they are mirrored in the database. In this document we refer to two types of visitor properties as *persistent* or *cached*.

Accessing Visitor Properties

There are three ways to access a visitor property for reading or writing.

1. Through visitor.*property_name* (for example visitor.EMAIL = "a@foo.com"). This is the quickest way.
2. Through the use of the lookupProperties (for reading) and updateProperties (for writing).[1] This is more cumbersome to code, but it is more efficient if you have a list of properties to read or write (because it is executed in a single action).
3. Direct read/writes to the database tables with executeSql(). This is not advised; it is probably wiser to stick with methods using the Broadvision API (i.e. 1 & 2 above).

Persistence

A *persistent* property is an attribute in a defined database table as declared in the system's prof_spec.src file. For example BV_USER_PROFILE is defined in prof_spec.src and has a EMAIL attribute. When a user is a Member or Guest, a write to visitor.EMAIL (for example) causes the visitor's BV_USER_PROFILE field to be updated with the same value. Thus the data persists across multiple sessions. The site can be configured (using the prof_spec.src file) to include other persistent profile values, even to other tables besides BV_USER_PROFILE.[2]

A *cached* property is one declared for use within a session, which corresponds to no database field. They do not need to be declared in prof_spec.src in advance; they are declared on-the-fly. For example, visitor.enteredSweeps could be used to store a Boolean to indicate that the person filled out a sweeps page; you would want that session-based as opposed to stored in the database.

Transient visitors are unique in that there are no persistent properties, even the ones that happen to correspond to database fields. All are cached.

---

[1] See appendix for sample code using lookupProperties() and updateProperties().

[2] In MT192003, for example we declared a MT_USER_PROFILE table, with an MT_PASSWORD field, which the code can write to with visitor.MT_PASSWORD; this is is described later in this document.

Reflectivity

For the most part, the three methods reflect each other. That is if you write with one method, say visitor.*property_name*, you can read it with lookupProperties(); or if you write with UpdateProperties(), you can read it with a Sql query; and so on. Not all cases go in both directions, and this is summarized in the table below. Asterisks mark cases that don't apply to transient visitors, because there is never a corresponding database storage.

Property Type	Write method	Readable by visitor.*property_name*	Readable by lookupProperties()	Readable by executeSql() *
Cached	Visitor.*property_name*	Yes	No	No
	UpdateProperties()	Yes	No	No
Persistent	Visitor.*property_name*	Yes	Yes	Yes
	UpdateProperties()	Yes	Yes	Yes
	ExecuteSql() *	Yes	Yes	Yes

Note that UpdateProperties will allow you to write cached properties, i.e. not corresponding to database fields, while lookupProperties() will not retrieve cached properties, only persistent ones. (One of Broadvision's quirky inconsitencies.)

Session variables

An additional kind of cached, non-persistent-across-session is to simply attach an arbitrary attribute to the Session object, on the fly.[3] As explained below, these are not actually "visitor properties" at all; we include it in this section to show another means by which the code can declare and use on-the-fly variables known to all pages in the site.

An situation where you might want to use a session variable is page tracking within the user's visit, such as "Session.sawCooktops = true". Later in this document we discuss our use of session variables Session.signedIn and Session.destinationURL in MT192003. These variables are not associated with the visitor, however. They are attached to the Session object directly rather than to the visitor object, so as long as the session stays alive the value is still available. If a user logs off, then another logs in with the same browser instance, the same session is alive. We may attach a different visitor object to it, but any other data we attached to it will still be there if we don't erase it or overwrite it.

Broadvision API: Account Manipulation Through BV_USER

The user's login name and password are stored in BV_USER (in the fields USER_ALIAS and PASSWORD). These are not visitor properties, so you can't read/write either USER_ALIAS or PASSWORD using visitor.property_name, lookupProperties() or updateProperties(). However Broadvision does allow, as one of the odd quirks of its API, write access with visitor.propertyname (i.e. you can write to visitor.PASSWORD). This can be used to allow a user to change their own password. The user name can be changed with a SQL UPDATE statement to BV_USER.USER_ALIAS, which makes it possible for a user to change their login name. The following table shows what you can and can't do.

Type of Action	Attribute	
	Member login name	Password
Read using visitor.*propertyname*	No	No

[3] These are exactly like Session variables in ASP. Unfortunately, Broadvision has no equivalent to the ASP "Application Variable", because Broadvision allows for multiple Interaction Managers on one server.

Read using lookupProperties()	No	No
Read with database query using BVI_GenericDBManager.executeSql()	Yes	Yes. An encrypted string will be returned. (It is impossible to retrieve the original clear text password from this field.)
Write (change) using visitor.*property_name*	No	Yes (!). When you write a clear-text password to this field, it gets stored in encrypted form into BV_USER.PASSWORD.
Write (change) using updateProperties()	No	No
Write (change) with database query using BVI_GenericDBManager.executeSql()	Yes	Yes, but don't do it! It will write it as clear text rather than encrypted, and it will be impossible to log the user in again.

Maytag Home Solutions (MT192003) Implementation

Visitor Type (early version)

At first we used a mixture of "Guest" and "Member" visitors. "Guests" were used when a user first came to the site and was anonymous (did not create a user account). The logic behind this was that (1) Guest accounts are "lighter weight" in terms of system resources (according to Broadvision), and (2) our Broadvision license put an upper limit on the number of Member accounts we were allowed.

In this approach, members were retrieved with the visitor() command. When an anonymous visitor reappeared, we recognized him by his cookie (details below), and connected him with his profile by means of a visitorsByQuery() command (matching to the user's GUID in the cookie).[4] If an anonymous visitor chose to sign up, we would then create a Member account for that person and forget about the preexisting profile data (the BV_USER_PROFILE for that user would be orphaned).[5]

Two drawbacks of this approach were:

1. Using two types of accounts meant that converting an anonymous user to a new member required the creation of a new account, rather than changing the existing record in place. This meant a growing number of accounts on the system, including one "zombie" account (never used again) for each member who transitioned from anonymous to member.
2. Certain forms in the site collected email addresses from the user (Brochure Request and Request Manuals). If the email address corresponded to a known member, nothing special would occur. However, if they were anonymous (created by .newGuestVisitor()), the site would create a "placeholder" for them by replacing their GUID with an email address while still keeping them as anonymous users (non-members). When such a user them became a member, we would create a new user with .newVisitor(). What we neglected to realize for a

[4] The .visitor() command can only be used to retrieve a user that was created with the .newVisitor() command. Guest visitors cannot be retrieved with a .visitor() command.

[5] Note that the existing Guest Visitor account cannot be repurposed into a Member account because there is no row in BV_USER for that user.

while is that such users now had two profiles with the same user name (two rows in BV_USER_PROFILE with the same value for .EMAIL). Because we were using visitorsByQuery() to match up a user with their profile, the system would be confused as to which profile to connect to. This led to a problem we had where users had lost their clear-text copy of their user password (MT_PASSWORD) because it was filled for one row of BV_USER_PROFILE and empty for another.

Visitor Type (final version)

Later we restructured our arrangement with Broadvision to have an unlimited number of Member accounts. Since then we have chosen to use just one type of user, "Member", even for anonymous users who only visit once. When they sign up to become members, we simply change them in place rather than creating a new user. This has the advantage that Giant Step's measurement is able to do "cradle to grave" tracking of all users, not just from the moment that they become members, but from their first visit to the site.[6]

Cookies and User Profiles

Four values get stored in the cookie for cookies-enabled users:

1. GUID: at first this is a randomly generated unique identifier (for example "9b444a:e4378223a0:-7fe2"). When the user becomes a member this value is overwritten with their email address.
2. LID: a copy of the original GUID. Does not change when user becomes a member. The LID is an attribute that the measurement team uses at Giant Step to uniquely identify members.
3. autoSignIn: takes the value 1 to mean to auto login the user (if a member), 0 otherwise.
4. CCHECK: a meaningless value written as part of a diagnostic to see if the user has cookies on their machine at all.

The normal entry for the user in the system is through the site's basic url (no extra cgi or path information, e.g. http://www.maytag.com rather than http://www.maytag.com/mths/homepage.jsp etc.). This loads the index.html page from the web server, which then redirects to the first page on the Broadvision server (redirect_home.jsp), which in turns redirects to the homepage.[7] We use these three steps as an opportunity to diagnose their cookie state, read the cookie if there is one, or manage the session differently if their browser is set to disable cookies. By the time the user hits the homepage, the state is completely diagnosed.

If a user with a cookies-enabled browser has no MHS cookie on their machine, this is detected in the index.html page on the web server. Index.html then writes (with Javascript code) the CCHECK value to the cookie, and puts a value on the querystring (cs=2) to be read by the next page, redirect_home.jsp. This code tells the page to check for the value of the CCHECK cookie. For this user, it would find a value, and pass the value cs=0 on the querystring to the next page, homepage.jsp. At this point, the user has been determined to be cookies-enabled.[8]

This two-stage diagnostic mainly exists because of the possibility of a user's browser being cookies-disabled; you cannot distinguish between a user who is cookies-disabled

---

[6] This approach might merit re-evaluation as the site grows, or for more heavily-trafficked sites. For a site with 1000's of new visitors daily, for example, this might not be as practical, and it would be best to make all visitors transient until they become members.

[7] The addition of the redirect_home.jsp step was originally to satisfy the needs of the Measurement group at Giant Step.

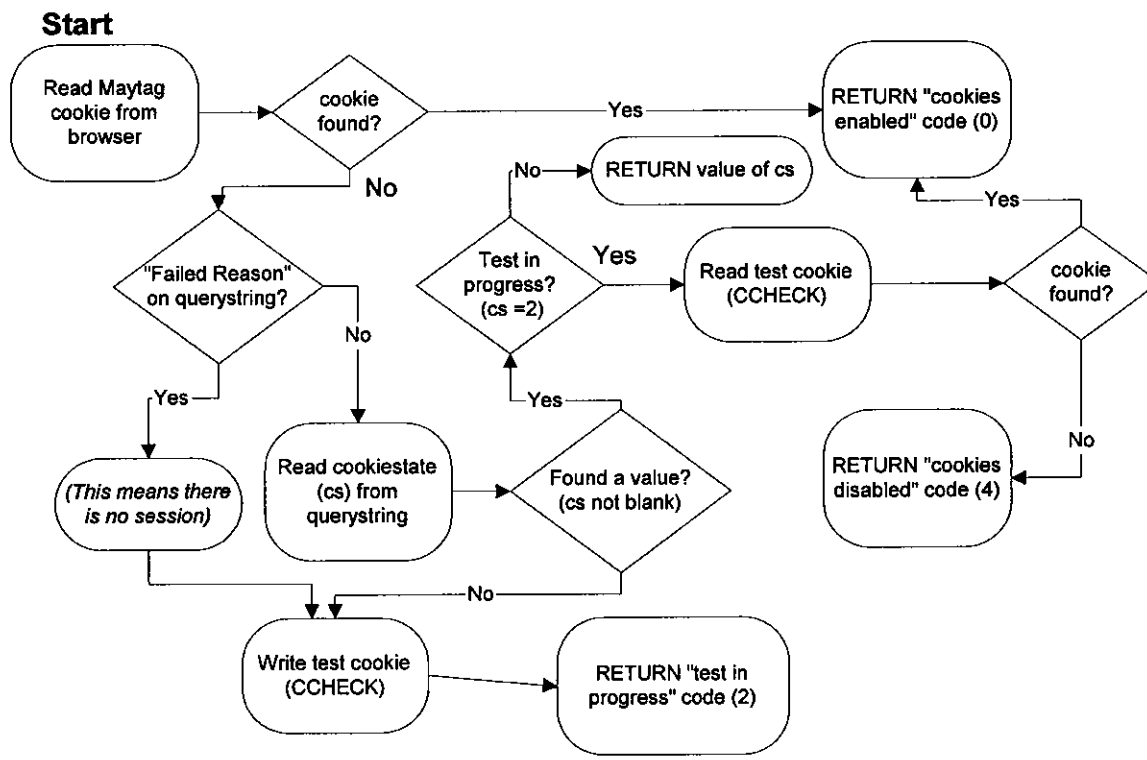
[8] The meanings of the five values that the cs argument could take on are: 0 = has cookies, 1 = no cookies, 2 = test in process, 3 = test not yet begun, 4 = maintain session on querystring

and one who simply doesn't have an MHS cookie on their machine yet. For a cookies-disabled browser, the redirect_home.jsp page fails to read the CCHECK value and passes cs=4 to homepage.jsp, which identifies the user as cookies-disabled. From that point on, the user's session is maintained on the querystring with the Broadvision values BV_SessionID and BV_EngineID.

If the user entered through some other page (e.g. through a bookmark to the products page), the same process would be handled over the course of whatever three successive pages they navigated to, by means of a function called getCookieState() (which is called every time processUserSession() is called).

When a visitor's session times out, essentially the same thing occurs: they are suddenly in the middle of the site somewhere, with an undetermined cookie state. For a bookmark, the URL carries cookie state information that was valid at the time when the cookie was created, but could be invalid later. For example, cs=0 will be incorrect for a cookies-disabled browser, so the cookie diagnostic must be restarted. In the case of both bookmarks and timeouts, the Broadvision server recognizes a non-existent session and redirects to a page which creates a session, and then back to the intended destination. Broadvision add two items on the querystring of that destination page, Failed_Page and Failed_Reason, that acts as a signal that a new session has begun. The routine getCookieState() detects that and restarts the diagnostic over again.

The complete cookie diagnostic is shown below:



For cookies-enabled members, with an MHS cookie on their browser, the site automatically connects up to their profile by (1) reading the email address from their GUID,

and using that value as an argument to the .visitor() function.[9] This user is signed in if the cookie value of autoSignIn is set to 1. A user who is signed in gets a greeting message on the homepage, and has direct access to all My Maytag pages.

We refer to "the cookie code" as the various routines that comprise the approach we are describing here. All the routines exist in the file /startup/mths.js. There is one main function called processUserSession() which is called as the first event on every page of the site. Other routines playing an important role in the cookie code are makeDevURL(), convertGuidToEmail() and getCookieState().

Visitor Session

A session exists for every page. It is created internally by Broadvision prior to the execution of the first JSP that the user visits in a new browser session, or after a timeout. The goal is to allow the user to browse through the pages of the site and have the session persist the whole way, rather than have a new session created for each page. We say that a visitor "has a session" when a valid visitor object, created by .visitor() or .newVisitor() is attached to the session object (i.e. Session.visitor). The diagnostic for whether a visitor has a session is to check if Session.visitor is null.

For cookies-enabled user, the session is maintained by a cached-based cookie that is managed by Broadvision. The Broadvision code does this by passing values called BV_SessionID and BV_EngineID on the HTTP header (invisible on the querystring, as in a form POST). We signal Broadvision that we want to do it this way by calling the function autoSetBVCookie(9). For cookies-disabled users, we maintain their session by placing BV_SessionID and BV_EngineID ourselves explicitly on the querystring. We signal Broadvision that session is being maintained that way by calling autoSetBVCookie(-9).[10] In order to create a site that can accommodate both solutions, it was necessary to set the "default-BV_UseBVCookie" config value to "No". By doing so, we must indicate the value "BV_UseBVCookie" at all times on the querystring.[11] For the cookie version that value must be "Yes", and no otherwise.

The objective of the cookie code, then, is (1) to choose the way the visitor preserves the session and (2) to attach a visitor object to that Session. During their time on the site they may change their visitor status (stay anonymous, become a member, become a different member, etc.) A variety of initial conditions can create numerous possible paths through the site:

Initial state	Initial visitor creation	Subsequent actions that changes or modifies visitor object
Cookies enabled, but no MHS cookie	▫ .newVisitor() ▫ User is anonymous (has GUID for username) ▫ Cookie is written.	▫ **Become a member (1).** The user's existing account and cookie are updated using the exisiting visitor object. ▫ **Sign in to an existing account (2).** .visitor() is called and this visitor object replaces the first one. The

[9] We experienced an interesting problem when users at Giant Step would visit different MHS servers within the Giant Step domain (development and test servers), because a single cookie was used by both servers. Therefore a GUID could refer to a user that was existent on one server but not the other. We called these "stale" cookies, and put in an exception path in the cookie code to handle it.

[10] If we did not do this kind of processing for cookies-disable users, such users would create a new session on every visit to every page on the site.

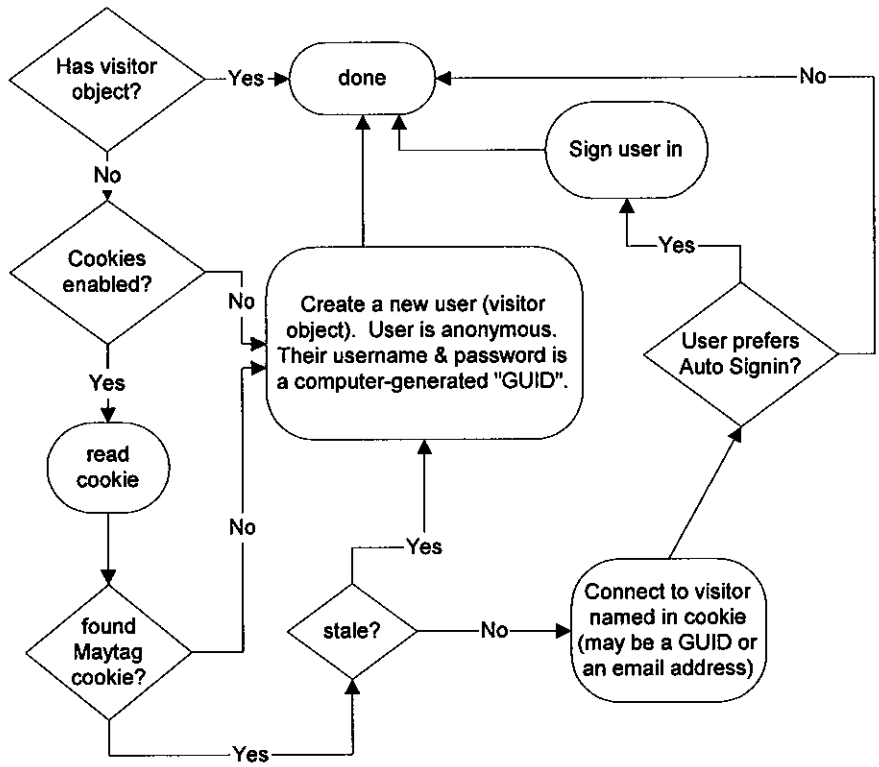
[11] On a side note: the User Experience team was concerned about appearances of any words on the querystring that might alarm users that they were being "tracked". Unfortunatley we couldn't do anything about the Broadvision-defined "BV_UseBVCookie".

		cookie is changed to refer to this user.
Cookies enabled, has MHS cookie for an anonymous user	▫ .visitor () ▫ GUID in cookie is used as username	▫ **(1)** and **(2)** above.
Cookies enabled, has MHS cookie for a member, autoSignIn is false	▫ .visitor () ▫ Email address in cookie is used as username	▫ **Sign into this existing account. (3)**. Existing visitor object doesn't change, but value Session.signedIn is set to true. ▫ **Sign in to another existing account (4)** .visitor() is called and this visitor object replaces the first one. The cookie is changed to refer to this user. ▫ **Create a new member (5)**. .newVisitor() is called and this visitor object replaces the first one. The cookie is changed to refer to this user.
Cookies enabled, has MHS cookie for a member, autoSignIn is false	▫ .visitor () ▫ Email address in cookie is used as username ▫ The user is signed in	▫ **Sign out (6)**. Set both Session.SignedIn and the autoSignIn value in the cookie to false. ▫ **(4)** above. ▫ **(5)** above.
Cookies disabled	New visitor is created with .newVisitor()[12]	**(1)** and **(2)** above (but with no cookie update in either case)

We wrote the cookie code in such a way to minimize processing on each page. Once the visitor profile is attached to the Session object, the job is mainly done. Although processUserSession() is called on every page of the site, for all but one of the pages, it will simply exit after discovering that Session.visitor is not null. The reason processUserSession() is called on every page and not just the homepage is in case people visit the site from another page (i.e. a bookmark).

The flowchart below shows a simplied version of what happens in each processUserSession() call:

[12] Note that a cookies-disabled user is creating a new anonymous visitor account on each new visit to the site. For a site with large numbers of new visitors daily, performance concerns may make it wiser to use transient visitors for these users.



The EMAIL property, a persistent property (corresponding to a field in BV_USER_PROFILE) is set by the cookie code to be the same as the USER_ALIAS field in BV_USER. It is a convient way to retrieve the current login name, and not much else. We point this out here to clarify that it is not "automatically" set by Broadvision when a member account is created.[13]

Our cookies-enabled and -disabled solution appears to have been somewhat of an innovation in the Broadvision world. Our Broadvision consultants indicated to us, and our own survey of sites confirms this, that other clients have either (1) maintained session entirely on the query string, making no use a cookies, or (2) used cookies only, and "freezed out" cookies-disabled users from member activities. Our use of separate solutions for cookies-enabled and -disabled users, sacrifices no member functionality for the cookies-disable users,[14] while allowing easy member access to cookies-enabled users.

Dynamically Generated URLs and Form data

In a system in which the querystring maintains session and information about the cookie state, it is essential that all links and forms have the correct values for cs and BV_UseBVCookie, and, for the cookies disabled case, BV_SessionID and BV_EngineID as well. In addition, the various servers that were used over the course of development, testing, staging and production varied in what needed to appear for the domain, cgi path info, and port. For example, each developer had their own Broadvision interaction manager, which meant a URL with a reference to a distinct port and cgi pathname. We created a function called makeDevURL() which accommodated both the needs of the

[13] In an earlier version of the cookie code, we would keep the EMAIL property empty until the person became a member, and then copy the USER_ALIAS to it. That was a way of distinguishing anonymous users from members, and we would log people in (i.e. connect them to their profile) by matching users to this field with a visitorsByQuery() call.

[14] The one exception is the lack of an auto Sign-in feature for cookies-disabled users.

cookie/session state on the querystring, and the server-specific domain, path and port information for URL creation. For adding information to forms we made the routine addCookieStateToForm().

Redirects

The site requirements dictated that certain processes were redirected automatically from one page to another. As mentioned above, there is a redirect necessary at the beginning of the site (redirect_home.jsp to homepage.jsp). In this case there was a cookie write involved prior to the redirect that posed problems for us. The Broadvision functions for doing redirects, Response.redirect() and Response.localRedirect() terminated page processing before the headers were written, causing the cookie write to be cancelled. After some searching we discovered that a redirect using the Javascript command "location.href = *URL*" allowed the redirect and the successful write of the cookie.

Some of the redirects that are done on the site include the following:

1. At the conclusion of Register a Product, the user is sent to the My Appliances page (where they can view their registered product).
2. If the user is not yet signed in or up, the Register a Product form acts a gateway to membership by passing the user through the signin/up page before reaching the final destination of to the My Appliances page.
3. After signing in (or up) the user from the home page the user is sent to the home page again. The same occurs after signing out, from the homepage or the My Maytag page.
4. A user who is not signed in/up who presses the My Maytag link is routed to the signin/up page before reaching the My Maytag page.
5. A user who adds a product to their wishlist must also pass through the signin/up page if they are not signed in/up.

For all of these cases except no. 2, the routing is solely through the signin/up page (signin_signup.jsp). All that was necessary was to communicate to the page that redirecting was needed, and a codeword to specify the destination. For example, redirecting to signin_signup.jsp with "redirect=true&dest=home" on the querystring was sufficient to cause a redirect to the homepage when the signin/up process completed.

Case 2 is more complicated because half of the processing occurs automatically regardless of user sign-in state. When the user successfully submits the Register a Product form, their product is registered with MCS without any need for sign-in. The next step in the process, adding the product to My Appliances, requires a sign-in, so a user who is not signed is diverted to the signin page, and then sent back to Register a Product to complete processing. This means that their registration data has to be carried as baggage through the signin/up process and returned to the Register a Product page. This is done by the Register a Product page's URL (the data baggage is on the querystring). In this case we handle this by putting the whole destination URL into a Session variable, called Session.destinationURL.

One-Way Password Encryption

Whenever a member's password is written to the database it gets written in encrypted format and is thereafter unretrievable in clear text (readable) form. This frustrated our goal of making it possible for a user to have his password emailed to him in a "forgot password" page. We solved this by creating a field called MT_PASSWORD in the database which is accessible as a visitor property. When the user account is created in the sign-up process,

the user's password is passed in the HTTP header as a normal FORM variable, so we take this opportunity to grab the clear-text password and copy it to MT_PASSWORD.

Appendix

Sample Code

A function that grabs a property:

```
function GetPropFromVisitor(visObj, prop) {
    var attr = new BVI_StringList();
    attr.append(prop);
    var propList = new BVI_Properties(attr,null);
    visObj.lookupProperties(propList,true);
    return(eval("propList." + prop);
}
```

...and one that sets a property

```
function SetVisitorProp(visObj, propname, propval)
    var  objProperties = new BVI_Properties();
    eval("objProperties." + propame + " = '" + propval +
  "'");
    visObj.updateProperties(objProperties);
}
```