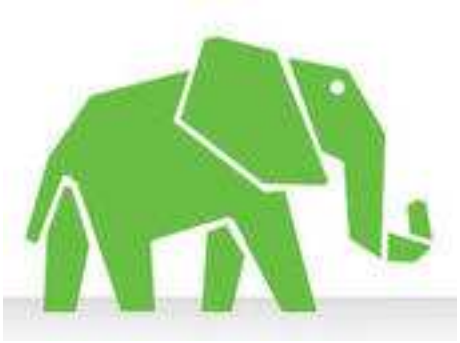


An Internet State of Mind
Part 1 – General Concepts

Session Number

Barbara Peisch
Peisch Custom Software, Inc.
3138 Roosevelt St. Suite O
Carlsbad, CA 92008
Voice: 760-729-9607
Fax: 760-729-9608
Email: Barbara@peisch.com

Overview

You've got a VFP application that's been running on your LAN. Now your boss wants you to put it on the web. Where do you start? Developing applications for the Internet is a completely different environment from developing applications for the desktop or LAN. In this session, we'll cover some of the concepts that seem to cause developers the most difficulty when first learning how to create web applications, **and we'll compare some typical design decisions you would make for a desktop application versus making those decisions for a web application.**

Caveats

In order to run the examples that come with this session as is, you'll need a copy of Web Connection. The demo version of Web Connection should work just fine. If you're using one of the other tools, you may need to make some minor changes to the code. Please refer to the instructions with the tool of your choice to determine how to configure and run an application in that environment.

For this demo, you'll need to define Login.htm as a document in IIS for the virtual site In the source code that comes with this session, the "WebContent" subdirectory is the path for the virtual site, and all files in that subdirectory belong in the virtual site.

I have not included the wc.dll file with the source code because that file is licensed. It would go in the WC subdirectory under the WebContent subdirectory. The application also expects a script map, defined in IIS, called ISM which points to this wc.dll file.

The path set in the Config.FPW file assumes that your copy of Web Connection is in a subdirectory of the application called WCONNECT. Be sure to change the path in the CONFIG.FPW file to reflect your environment.

Scope of This Topic

True web applications are a different animal from LAN applications. **Although there are ways to take a LAN application as is, and run it across the Internet, such as using Terminal Server** or Citrix Server, this session deals only with applications that were truly created to run on the Internet. Running LAN applications across the Internet using one of these server packages has its own set of challenges, and these are outside the scope of this session.

Also outside the scope of this session are the topics of setting up a web server, security settings and a comparison of the various web development tools available.

What we will cover are differences in concepts and design between LAN applications and web applications.

The More Things Change, the More They Stay the Same

Remember the days when "cross-platform" meant you could run a program on Windows, Mac or Unix? That never worked out as smoothly as promised, did it? You've probably also heard that the web is truly cross-platform. Not really. We've just traded our problems with differences in operating systems for problems with differences in browsers. With a web application, the primary way a user will interact with your application is through a web browser. And guess, what! There are lots of different "flavors" of web browsers. Not only that, you may have a whole different set of problems from one version of the same browser to the next.

What this means is that you really need to know your audience when you design a web application. Do you have the ability to dictate the browser and version that will be used? Is this an application that is available to the general public, or do you have a limited audience? Chances are, if your application is meant for a limited audience, you'll be able to state browser requirements. That means you'll have more options for using features specific to that browser.

If your application will be used by the general public, or a wide group of people, you'll probably have to allow for more browser variation, and will be more restricted in the features you use. The key is to know which features the various browsers support, and which they don't. Check the References section at the end of this paper for some useful links, including a link to a chart of which features are supported by which browsers.

One mitigating factor in all of this is the World Wide Web Committee (W3C for short). This group governs the standards for the web, something sorely lacking in the world of operating systems. This does make the web a more likely cross-platform environment than what we've seen with different operating systems.

Some Things Have Changed

You've probably heard the term "stateless" used when referring to the Internet. You may not really know what this means. When you have workstations on a LAN, each workstation has a real-time connection to the server for the duration of the time the user is logged on. This means you can easily keep track of things like how many users are on your system at any given point. This isn't the case with the Internet**. In a stateless environment, you don't have a permanent connection to a web server.** You're only connected for as long as it takes to send a request to the server, and receive a response. Each request or "hit" is disconnected. Combine the "stateless" concept with the fact that with an Internet application, VFP is only running on your server, not on each workstation, and you have a situation where your application doesn't know from one request to the next which user is making a request, what tables they have open, or where any of the record pointers are. Even memory variables and application properties can't be relied upon from one hit to the next to hold a value you can trust as relevant to the current user.

As you can imagine, this can present some challenges you haven't had to deal with before.

Why Create a Web Application?

Given these new challenges, you may be asking yourself, "Why should I write a web application instead of sticking to a LAN application?" **Good question. One answer is "connectivity". Another is "ease of distribution".**

Connectivity

Once upon a time, when the PC was a new invention, computers didn't talk to each other. If you wanted to share information between computers, you either duplicated your efforts, and typed it into each machine, or maybe you could copy the data to a floppy and copy it from the floppy to the other machines.

Then along came the Local Area Network (LAN), and our world changed. Building multi-user applications presented some challenges we hadn't had to deal with when there was only one computer using the application. But what made it all worth it was the ease of sharing data. You could have a single program that allowed an entire office to enter data into a common directory. And customer service representatives could use that same common directory when a customer called in, and have all the data for that customer at their fingertips. (This isn't too unlike the old mainframe days.)

But what about the building that the company expanded into that's down the block? For that, the Wide Area Network (WAN) was invented. Suddenly, everyone didn't have to be in the same building to share the same network anymore.

But what about our sales team that travels around town, or around the country, or around the world? They can't get the status of orders and inventory while at a client site, can they? With the Internet, they can!

If you have customers who phone-in orders, or send orders by fax, they can enter those orders themselves through a web application.

If you have customers who want to know the status of an order they've already placed, or who want information about your company, they can look it up on the web.

All this means that you don't need someone on the phone dealing with customer or sales support nearly as much as you used to, and you can put your employees to work on other tasks instead—a very important edge in today's tight economy.

Ease of Distribution

If your company distributes software by CD or other hardware, you're aware of the distribution costs in terms of time and materials every time any changes need to be sent out. If there's a major bug in what you ship, your costs are instantly doubled because you have to correct the error and send out a new version right away.

Even if your application isn't a full web application, imagine if there were some mechanism in your software that checked a web site for updates when the application is started. You could post the new version on a web server, and the next time each of your customers started their program, they'd get the update automatically. **No CDs to burn. No shipping costs**.

If your application is a full web application, it doesn't even have to check for a new version at startup because users don't have the application on their computers—it's all on the server. **You update the application on the server, and that's it!**

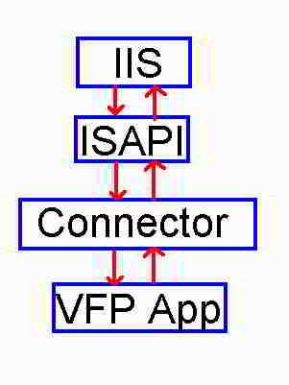
How Does it All Work?

If you're running a Windows machine, you need to **have IIS** installed (Internet Information Server). Your web application must be able to talk **to ISAPI (the Internet Server API**). To accomplish this, you need an ISAPI connector, written in an ISAPI compliant language such as C++. **FoxPro is a language which is not "ISAPI compliant" by nature, which means it's not easy to create a DLL that can talk directly to ISAPI.**[1]

So, IIS receives incoming web requests, and communicates with ISAPI. ISAPI communicates with an ISAPI connector. The ISAPI connector communicates with your application.

---

[1] There is an ISAPI connector called FoxISAPI.dll, which ships with VFP, along with samples of how to use it. Unlike the third party web tools, there is no framework or supporting functions that come along with FoxISAPI, which means you'll have to spend a lot of time creating these yourself or, at a minimum, modifying the sample code to suit your needs.



The ISAPI connector I'll be using in the examples for this session is called wc.dll.

Who's Doing What

As I mentioned already, for a web application, VFP is running on the server only, not on any workstations (clients). In the case of a web application, the clients are running internet browsers. This is probably easier to understand with some pictures.

Workstation Server

VFP ←------- ----------→Data

(data records)

As shown in the diagram above, on a LAN your VFP application talks directly to the user. It gets its data remotely, and transparently, over the LAN. **Here's a web version:**

Anybody←---Web---->OS web server←--------->App Server
(IIS and ISAPI) (VFP)

Data

On the web, your VFP application talks to the OS web server via ISAPI. It may get its data directly or remotely, but the users are also remote and transparent.

Where the Pieces Fit

One big change you'll have to get used to is the idea of a virtual directory versus your application directory. This subject delves into the area of server setup, which I mentioned was outside the scope of this session, but I would like to introduce the concepts because they are so different from the way a LAN application is setup.

A virtual directory is a path you've defined in IIS as the home page for your application. This directory, and everything underneath it is accessible through the web to the outside world. The kinds of files you'll keep in the virtual directory and it's subdirectories are any HTML files, image files or cascading style sheets your site uses.

Your application directory holds the executable for your application. You may also choose to put data under your application directory, but you may put it elsewhere, as long as it's not under the virtual directory. This is to prevent access to your data through means other than through your application.

The ISAPI connector may go anywhere, but be aware that the Internet Guest Account will need execute rights to the connector, so this may affect your decision as to where to put it. It is common to keep the ISAPI connector somewhere under the virtual directory.

The Players

There is one important user account on your system you may not be aware of. This is the Internet Guest Account, and is called **IUSR_NameOfYourMachine**. This account is created when IIS is installed. The "NameOfYourMachine" part of that name will be whatever your machine is called. For example, if my machine's name we WebServer, the Internet Guest Account will be IUSR_WebServer. The Internet Guest Account must have read access to the virtual directory, and execute access to the ISAPI connector.

The user who executes your application **can** be any user account, but shouldn't be the Internet Guest Account.

Comparison of Design Decisions

Let's compare the differences in the types of decisions you're likely to make for a LAN application versus a web application.

Security

With a LAN application, there's only one way a user can get into your application—by running the executable. Generally, you don't provide users with a command window, but even when you do, unless they're really familiar with your application or your programming style, they're not going to try to do something like call a function in the middle of your program from the command window. Even if they did, the whole thing would probably come crashing down in a heap because the environment wasn't setup properly.

What this means is you can provide a login screen at the start of your application, and you know who that user is for the rest of the time they're running the application. This is not the case in a web application. Often, function names in the middle of your application are displayed for full view in the address bar of the browser. A user may even try to bookmark a screen in the middle of your application. If your application is one that requires a login, you must make sure a user is logged in for every hit you receive!

You also must carefully screen input from users of a web application. Although this does fall under the category of security, we'll discuss this more fully in part 2, under the topics of URL Hacking and SQL Injection.

Screens

If you've been used to creating forms in VFP, you're used to working in a rich environment. *HTML and the web is going to seem like a giant step backwards*.

In VFP forms, we're used to being able to specify an input mask for fields. If a field has a ControlSource that is a Date type, that field will only accept valid dates. You've also gotten used to being able to do all kinds of validation on fields as soon as they lose focus.

As you might guess, this isn't the case with HTML forms. You may be able to use some client-side scripting with JavaScript for some validation, but what if the user has JavaScript turned off in their browser? What if they found some other way to send a request to your server, other than the means you intended, i.e. by hacking? (We'll cover this more in part 2.) Although formatting may be possible with JavaScript or an ActiveX control, there's no guarantee this worked on the client**. You must perform all validation on the server for the incoming data**, regardless of what you think the client may have done.

If you're used to using VFP data in your applications, you probably have gotten used to being able to display the entire contents of a table in a grid, so the user can browse the table and make a selection. This isn't a good idea in a web application. (It isn't in a client/server application either, for similar reasons.) In a web application, you don't have a connection to the server that can handle large amounts of data anywhere as quickly as a LAN application can. If you have a cursor of any decent size to display, **chances are the user's browser will timeout before it can download the data--**and nothing will be rendered in the browser until all the data is downloaded. You need to come up with a different scheme.

There are a couple of ways you can approach this problem. One is to put up a screen where the user enters criteria to narrow the search. You can then display a much smaller set of records for the user to choose from. Another approach is to fetch all the records, but only send back a subset (a page worth) at a time. So your big file would show the first 20 or so records, and when the **user clicks "Next", you'd requery the data and send the next 20 records**. This keeps the amount of data you're sending to the client to a reasonable amount.

Reports

Reports on the web can be done in HTML or PDF format. If you want to create HTML output, you can do that directly from within your application, but just like showing records on screens, you need to be careful about the amount of data you're sending to the client. You need a similar mechanism as discussed above where you display a limited number of records until the user asks for the next screen. Another alternative is to send the entire report to an HTML file, and e-mail that report to the user. This works in cases where viewing the report isn't vital to the process of using the application. If your application requires users to get information from a report while using the application, sending the file by e-mail won't work.

Another option is to create a PDF file. This requires a special third party tool (unless you want to spend HUGE amounts of time and code it yourself). You can use Adobe Acrobat (www.adobe.com), Amyuni (www.amyuni.com), PDF995 (www.pdf995.com), The Minds Eye Report Engine (www.mindseyeinc.com) or any of a number of tools for creating PDF files. If the creation of the PDF takes an appreciable amount of time, you may need to do this in an asynchronous process (discussed in part 2). If you create PDF reports, the user must also have a PDF reader on their computer. They can download that for free from the Adobe web site, but you'll probably want to have a link on your site so users can get it easily.

How do you decide which format to use? PDF certainly has more formatting flexibility, and you can zoom in or out on a PDF document. Some of the PDF tools, like MindsEye, will even use an existing FRX to generate a PDF document. If these features are important to you, then you probably want to use PDF reports. On the other hand, if your reporting requirements are fairly simple, and don't require the capabilities of a PDF reader, then consider HTML. It may come down to a matter of what you're more comfortable with.

4 Standard Web Objects

Regardless of which of the VFP web tools you use, or even if you use ASP, **there are four standard objects you'll be using in your application**. These may have slightly different names in different tools (e.g. Active VFP uses an "o" in front of the object name in most cases), but the purpose of the objects are the same. You'll see examples of how most of these objects are used in the code examples which follow this section.

Server Object

Before I explain what the server object is, I need to clarify the term "server". Unfortunately, the term "server" is used to mean three different things in the web world. First, there's the physical computer used to store your files. I like to refer to this as the "physical server". Then there's the web server that's tied into the operating system (OS). These are things like IIS (Internet Information Server) if your OS is Windows, or Apache if your OS is Linux. I'll refer to this as the "OS server". The third type of server is the one called the "Server Object" here, and the one we'll be referring to the most in these sessions. **This is your VFP application that runs on the physical server, and waits for hits to your web site**.

Session Object

The Session object allows you to restore state in the web's stateless environment. It provides an interface to a persistent place for data to be stored for the user between hits. The session object acts like a coat-check, you give it a coat, it gives you back a ticket. This ticket is given to the user (usually in the form of a cookie). On the next hit, the user presents this ticket and he gets his coat back. The "coat" in this case would be information that tells your app what the user was doing on his last hit and guides your application in serving the next one.

Request Object

The Request object is the object you will use to read any variables sent to your server in a hit.

Response Object

The Response object is the object you will use to send output back to the user.

Basic Application Functions – Login/Add/Edit/Delete/Reporting

Let's take some simple examples of basic functions you would normally need in an application. These examples are meant to show basic concepts only and are by no means complete or ready for production.

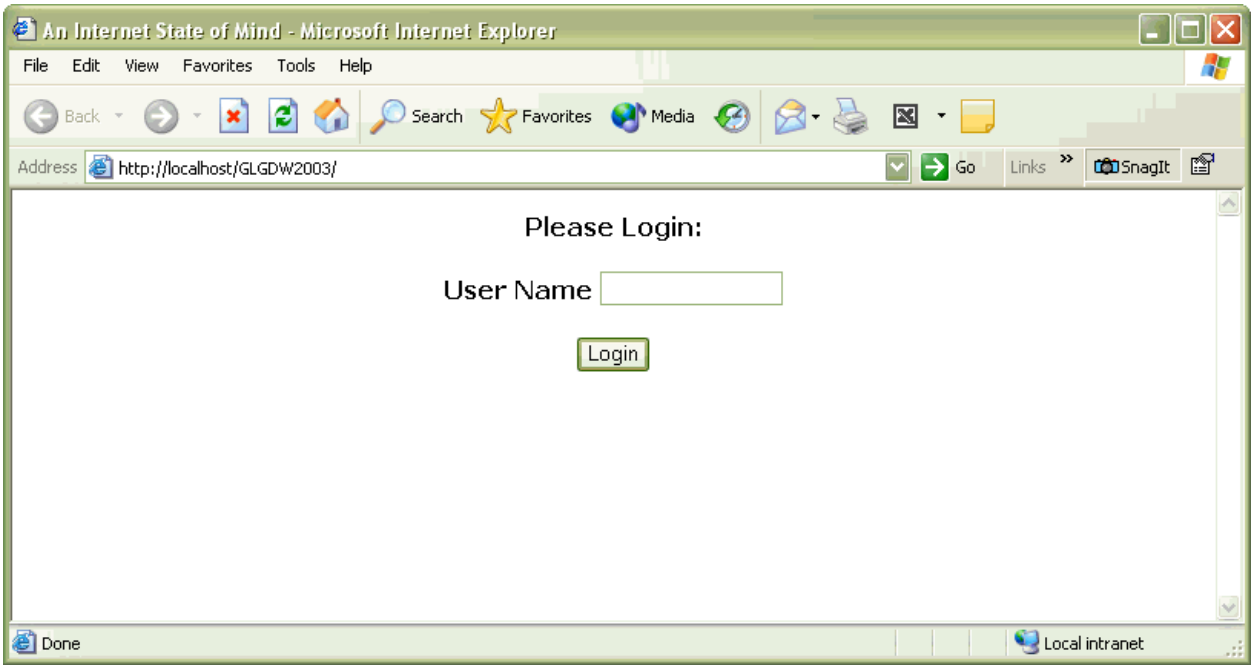
This is a simple application for entering, editing and reporting on invoices. Each invoice has an "owner", which happens to be the name of the person logged in when the invoice was created.

I'm using Web Connection for all my examples because that's what I know and use. All of the concepts will translate to other tools fairly seamlessly. Web Connection has a fifth object not mentioned above, called the Process class, which is the "central processing method" mentioned in the next section.. In Web Connection, all the code you create is in or is called by the process class. All references to "This" in the code shown below is referring to the process class.

Although I don't show it explicitly in the examples shown below, the top of my process class checks to see if the user has created a session, and creates a session for the user if one doesn't already exist.

Login

When the application first starts, you are greeted by a login screen that looks like this:



This screen is created by code in a separate file called Login.htm. The code looks like this:

```
<html>

<head>
<title>An Internet State of Mind</title>
</head>

<body>

<form method="post" id=frmLogin name=form1 action=login.ism>
  <p>
  <CENTER><font face="Verdana"><b>Please Login: </b>
         </font></CENTER> 
  <CENTER><FONT face=Verdana><b>User Name </b>
     <input type="textbox" name="txtUserName" size="16"></CENTER></FONT> 
  <CENTER><INPUT id=submit1 name=btnLogin type=submit value=Login>
      </CENTER> 
</form>

</body>

</html>
```

I'll get more into the detail of how an HTML form works shortly. Just note that I've specified the name of the textbox to be txtUserName. It's not a coincidence that this looks a lot like naming syntax in VFP. I do this because this is the variable name I'll use in my VFP application when I want to determine what name the user entered. This naming standard is easy for most VFP developers to understand.

Also note that in the <form> tag, there is a clause in that tag that says, "action=login.ism". This tells our VFP application to look for the login method. I'll explain this further with the next code example.

You must make sure your user is logged in on each hit. You'll need some kind of central processing method that looks at the incoming request and figures out which method to call. Most of the tools mentioned at the end of this paper provide this mechanism, and you just need to add a call to your login method somewhere in this central processing method. Your login method will probably look something like this:

```
FUNCTION Login
LOCAL lcUserName

IF TYPE('Session') = 'O' AND NOT EMPTY(Session.GetSessionVar('UserName'))
      RETURN .T.
ENDIF

IF NOT This.OpenTable('Users')
      RETURN .F.
ENDIF

lcUserName = Request.Form("txtUserName")
```

```
IF EMPTY(lcUserName)
      Response.ErrorMsg('No user name','You must log in',,3,"Login.htm")
      RETURN .F.
ENDIF

LOCATE FOR UPPER(LoginName) = UPPER(lcUserName)
IF EOF()
      Response.ErrorMsg('User Name not found', ;
         'The user name you entered is not in our database.')
      RETURN .F.
ENDIF

Session.SetSessionVar('UserName',lcUserName)

This.DemoOptions()      && Show the demo options

RETURN .T.

ENDFUNC
```

You'll notice at the top of this method, we check if we've created a session object, and if so, we use the session object to check if there's a session variable called "UserName" that has been created. If both of these cases are true, we can assume the user is already logged in. For this application, we use a VFP table to store session information, including the ID of the session, which is passed to us in the form of a cookie in the HTTP header of the request, and which we send back in the response to the user in the HTTP header. Our central processing method, which is called for each hit, will look for this ID, and will either create a new session if there's no ID or no session record, or it will re-establish the session object if there is already an ID. This ID is stored in the session table, and gives us a place to store variables for this user, like the UserName, so we don't have to ask for it on each hit.

We must return .F. from this function if the login is not successful. You must also check for this return value in your central processing method after the call to the login method. If you see the login method has returned .F., you must not continue to process the request, but instead, return from the method. The error messages generated from within the Login method will provide a response to the user, so it's not necessary for your central processing method to generate output to send back to the user.

When a user logs in successfully, I send them directly to the DemoOptions method. This method displays the page of hyperlinks for the various options for the demo. This isn't really a good technique because the login method should just have the user login and return .T. or .F., but to simply things in this demo, I'm having the login method send the user to the main screen.

OpenTable is a function I wrote, and is available in the source code with this session. You cannot rely on a table being open or not from one hit to the next, so before using any table, I call the OpenTable method. This method essentially looks to see if the table is already opened. If so, it selects the table, if not, it opens the table.

Note the use of the Request object to retrieve form variable sent by the user. I'm using a method of the Request object called "Form" for retrieving form variables. There are similar methods in all the web tools.

Also note the use of the Response object to create messages that will be sent back to the user, in case the login is not successful.

Finally, in the case where the login is successful, note the use of the Session object to create a session variable called “UserName” that will save the name of the user for future hits.

The Invoice Form

In the sample application provided with this session, the invoice processing form is stored in a separate file called Invoices.wc. This file, shown below, contains the HTML for the browser to display, and uses ASP style tags for displaying field values. (These are the tags with <%= … % > in them.) This file is a plain ASCII file, and can be edited with any text editor, like NotePad, or the VFP modify file command.

```
<html>

<head>
<title>Invoices</title>
</head>

<body>

<script language=JavaScript>

function DelConfirm()
{
  if (confirm("Delete this invoice?"))
     frmInvoice.submit();
  else
     alert("No invoice was deleted");
}
</script>

<h1 align="center">Invoices</h1>
<center>
<form method="POST" action="ProcessInv.ism" name="frmInvoice">
  <p>Invoice#: <input type="text" name="txtInvNo"
      value="<%= iif(TYPE("oInvoices")='O',oInvoices.InvNo,'0') %>"
              size="20"></p>
  <p>Owner: <input type="text" name="txtOwner"
      value="<%= iif(TYPE("oInvoices")='O',oInvoices.OwnerName,'') %>"
              size="20"></p>
  <p>Date: <input type="text" name="txtInvDate"
      value="<%= iif(TYPE("oInvoices")='O',oInvoices.InvDate,'  /  /    ') %>”
              size="20"></p>
  <p>Total: <input type="text" name="txtInvTotal"
      value="<%= iif(TYPE("oInvoices")='O',oInvoices.InvTotal,'0') %>"
             size="20"></p>
  <p><input type="submit" value="Add Invoice" name="cmdSubmit">
     <input type="submit" value="Find" name="cmdSubmit">
     <input type="button" value="Delete Invoice" name="cmdDelete"
onclick="DelConfirm()">
     <input type="submit" value="Report" name="cmdSubmit">
     <input type="submit" value="Save" name="cmdSubmit">
     <input type="submit" value="Cancel" name="cmdSubmit">
     <input type="submit" value="Return to options" name="cmdSubmit"></p>
```

```
</form>
</center>

</body>

</html>
```

Don't worry about the script tag shown near the beginning of the code. We'll get to that a bit later. The important things to note about this code are fields for input, identified with a <input type="text"> tag, and the ASP style tags.

First, let's address the input fields. You'll notice that each of these has a "name=" clause. The value we assign to this clause determines what variable name we need to look for when the form is submitted to the server. In the code above, the value of the first input field will be in a variable called txtInvNo, the value of the second field will be in a variable called txtOwner, etc.

Next, let's look at the ASP style tags. These are used in conjunction with a "value=" clause. What we are doing is calling VFP code to determine what to display in these input fields. This file isn't displayed directly by the browser. If you double-click on it in Windows Explorer, and say to you want to use Internet Explorer to open the file, Internet Explorer won't know what to do with the ASP style tags. Instead, this file is processed through your server, and the ASP style tags are interpreted, and the resulting values are inserted in place of the tags. Then the output is sent to the browser. All of the VFP tools listed here have a mechanism for handling scripting pages in this way.

Using ASP style tags is one way of showing data from VFP tables in a browser. We'll look at another technique shortly.

Another form of ASP style tags allow you to run multiple lines of VFP code from within an HTML file, such as looping and branching constructs. We won't be using any examples of these kinds of scripts in this session, but it's good for you to know they exist.

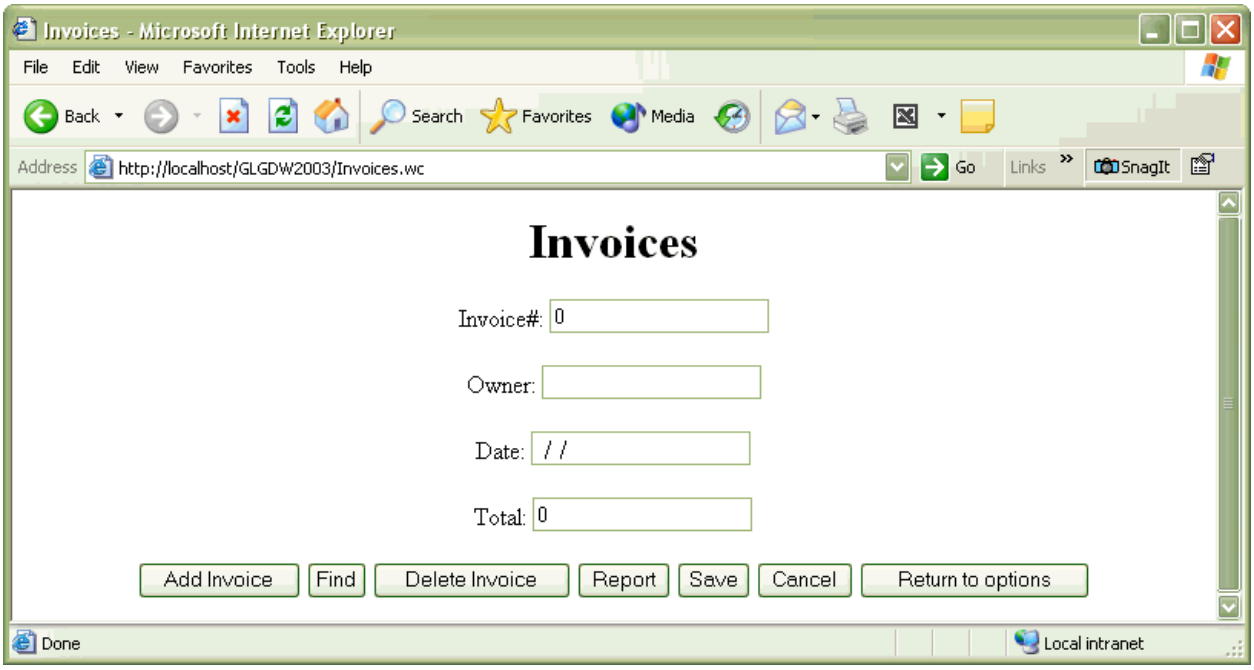
Notice that there's a form tag in the code shown above, which specified we'll be using a POST method for submitting the form. This is the method you'll probably use 90% of the time to submit a form. Only variables between the <form> and </form> tags will be sent to the server when this form is submitted. Also notice that there's an "action=" clause in the form tag, which specifies a file name. This tells the server what method to run when the form is submitted.

The file name of the action clause is the name of the method or function we want to run. Our central processing method on the server will look for this in the parameters it receives, and will call this method or function.

The extension of the file name in the action is mapped to what's called a "script map" in IIS. A script map is a shortcut that tells ISAPI where and what your ISAPI connector for the web site is. In this case, I've defined a script map called ISM (for Internet State of Mind), and it's pointing to the wc.dll file I use for the web site. The wc.dll file sends the request on to my server application. The exact syntax for calling your code differs slightly with web each tool. Please refer to the documentation for your tool of choice for the syntax you should use.

The ProcessInv method called by the action clause is simply a method that retrieves the value of the cmdSubmit variable. (It will be the caption of the button that was pressed.) It then has a case statement to call the appropriate method, based on the button pressed.

This code will produce the following screen on the user's browser:



Adding

In this application, I'm adding a record as soon as the Add button is pressed. I'm using VFP 8's autoincrement feature to assign the invoice number to the InvNo field. The UserName used to login to the application is what I use to identify the owner of the invoice.

```
FUNCTION AddInv
LOCAL lcOwner

IF not this.OpenTable("Invoices")
      return
ENDIF

lcOwner = Session.GetSessionVar("Owner")
INSERT into Invoices (OwnerName) values (lcOwner)
PRIVATE oInvoices
SCATTER name oInvoices

Response.ExpandTemplate(ADDBS(This.HTMLPath)+"Invoices.wc")

ENDFUNC
```

The oInvoices object created by this function is the same oInvoice object referenced in the Invoices.wc file for displaying data. I then call the Web Connection function, ExpandTemplate, to interpret the ASP tags in the Invoices.wc file, and send the output to the browser. The form displayed is the same form we saw originally, only with data in the fields.

Editing

In order to edit a record, we have to find it first. In this application, we find a record by entering the invoice number we want to edit in the Invoice# field and clicking on the "Find" button. The FindInv method will be called by the ProcessInv method.

```
FUNCTION FindInv
LOCAL lnInvNo

IF not this.OpenTable("Invoices")
      RETURN
ENDIF

lnInvNo = VAL(Request.Form('txtInvNo'))

LOCATE for InvNo = lnInvNo
IF UPPER(OwnerName) <> UPPER(Session.GetSessionVar("Owner")) AND NOT EOF()
      Response.StandardPage("Not your invoice", ;
       "Sorry, but this invoice number doesn't belong to you" ;
       ,,5,"DemoOptions.ism")
ELSE
      IF EOF()
            Response.StandardPage("Locate error","I couldn't find invoice " ;
           +TRANSFORM(lnInvNo)+" and I'm not sure what to do about it")
      ELSE
            PRIVATE oInvoices
            SCATTER name oInvoices
            Response.ExpandTemplate(ADDBS(This.HTMLPath)+"Invoices.wc")
      ENDIF
ENDIF

ENDFUNC
```

This function isn't too different from ones we've already seen. It's using the Request object to get the invoice number entered. Note that it uses the VAL() of the invoice number retrieved from Request.Form. This is because everything you get back from a browser is in string format. It doesn't know about numeric or date formats.

Once the invoice is found, the function creates the oInvoice data object used by the Invoices.wc template, then calls the Response.ExpandTemplate, which is a method to interpret the ASP tags and create the output.

The StandardPage method used is a Web Connection method for quickly displaying output on the browser. The first parameter is the page heading, the second is the message to display. The third parameter is optional, and not used here. The fourth parameter is an optional timeout. This parameter will create a META refresh tag, which is an HTML tag for specifying a timeout period, after which the page will send you to a URL specified by the fifth parameter.

Once the invoice has been found and displayed, the user is free to edit it. They must then press the Save button to save their changes.

```
FUNCTION SaveInv
LOCAL lnInvNo, lcOwnerName, ldInvDate, lnInvTotal

lnInvNo     = VAL(Request.Form('txtInvNo'))
IF EMPTY(lnInvNo)
   lnInvNo = VAL(Request.QueryString("InvNo"))
ENDIF

lcOwnerName = Request.Form('txtOwner')
ldInvDate   = CTOD(Request.Form('txtInvDate'))
lnInvTotal  = VAL(Request.Form('txtInvTotal'))

IF not this.OpenTable("Invoices")
      RETURN
ENDIF

LOCATE for InvNo = lnInvNo
IF UPPER(OwnerName) <> UPPER(Session.GetSessionVar("Owner")) AND NOT EOF()
      this.StandardPage("Not your invoice", ;
      "Sorry, but this invoice number doesn't belong to you" ;
      ,,5,"DemoOptions.ism")
ELSE
      IF EOF()
            this.StandardPage("Locate error", ;
            "I couldn't find invoice "+TRANSFORM(lnInvNo) ;
            +" and I'm not sure what to do about it")
      ELSE
            replace InvDate with ldInvDate, ;
                    Invtotal with lnInvTotal

            this.StandardPage("Changes Saved", ;
            "Your changes are saved!",,3,"DemoOptions.ism")
      ENDIF
ENDIF

ENDFUNC
```

Note that we don’t assume we’re on the correct invoice record before saving. This is an important point. You can’t be sure from one hit to the next that a given table is open or not, is the currently selected work area, or that it’s pointed to any specific record.

Theoretically, the user shouldn’t be able to view an invoice that isn’t theirs, and therefore, wouldn’t be able to save an invoice that isn’t theirs. But I check that the current user is the owner of the invoice anyway. Remember what I said about checking all input from the user? Here’s an example of doing that. We’ll look at cases where this becomes very important.

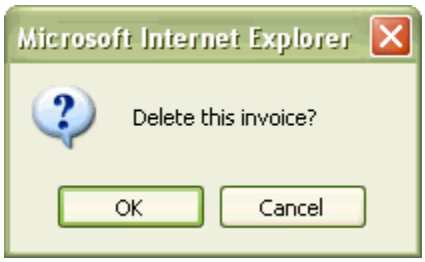
Deleting

You always want to confirm with the user before deleting anything. There are two techniques for doing this. One would be when the user clicks the Delete button, send the request to the server,

have the server send back a confirmation form, have the user answer that, then have the server respond appropriately depending on the answer to the confirmation.

Remember the script code that we saw in the Invoices.wc file? This is used for the second way of performing the confirmation. This technique performs the confirmation without leaving the form. Only if the user answers "OK" to the confirmation is a request sent to the server. As you can imagine, this is a more efficient technique.

The way this works is the delete button is not defined as a submit button as the other buttons are. It's simply defined as a button. This means that the form won't be submitted as soon as the button is pressed. Instead, there's an "onclick=" clause in the definition of the button. This is a Javascript event, and the definition says to call a function called DelConfirm. The DelConfirm method is defined in the script section of the Invoices.wc file. The DelConfirm function calls another Javascript function called Confirm. This will display the dialog shown below.



If the user clicks "OK" on the confirm screen, only then is the form submitted to the server.

Once we've confirmed the user's request to delete an invoice, the code for deleting is similar to other functions we've seen.

```
FUNCTION DelInv

IF not this.OpenTable("Invoices")
      RETURN
ENDIF

lnInvNo = VAL(Request.Form('txtInvNo'))
IF EMPTY(lnInvNo)
   lnInvNo = VAL(Request.QueryString("InvNo"))
ENDIF

LOCATE for InvNo = lnInvNo
IF OwnerName <> Session.GetSessionVar("Owner") AND NOT EOF()
      Response.ErrorMsg("Hacker Alert!", ;
      "You hacked the URL, <br><b>you bad person, you!</b>" ;
       ,,15,"DemoOptions.ism")
ELSE
      IF EOF()
            Response.StandardPage("Locate error", ;
            "I couldn't find invoice "+TRANSFORM(lnInvNo) ;
            +" and I'm not sure what to do about it")
      ELSE
            DELETE
            Response.StandardPage("Deleted", ;
            "Invoice "+TRANSFORM(lnInvNo) ;
            +" has been deleted!",,3,"DemoOptions.ism")
```

```
      ENDIF
ENDIF

ENDFUNC
```

The code above tests that the owner for the invoice to delete is the currently logged in user. If not, we're displaying a message that indicates we've got a hacker. This method can be called either of two ways, which we'll see in part 2, and the second way opens us up for some easy hacking.

Reporting

With reporting, you have to decide if you want to create your reports in HTML or PDF. If you choose PDF, you'll need to purchase a utility that will create PDF output. Generally, these products will install a new printer driver on your system which, when selected, will create PDF output from whatever you send to it. This process may be more tricky than it would seem. Adobe creates your output in a temporary file, and doesn't always copy the results to the file and directory of your choosing. Also, if you create PDF output, users must have a PDF reader on their computer. The PDF reader can be downloaded for free from the Adobe web site (http://www.adobe.com/products/acrobat/readstep2.html) and you may want to provide this link in your application so users who don't already have a reader installed can get one easily.

The example for this session uses HTML output. Reporting in HTML format is really a two-part process. Part 1 creates a cursor of the records to be displayed, and part 2 is the output mechanism. Chances are, you'll have more records to display than will fit in a single browser screen. You really don't want to send more than that all at once to the browser. You need a way for the user to indicate that they're ready for the next page or want to go back to the previous page, and a way to find that section of the report and send it to the browser.

This code in the example below assumes that the resulting table for your report can be created in a reasonably short amount of time. If creating the table (or PDF output) takes a noticeable amount of time, then you need to run it as an asynchronous process, which is discussed in part two of this topic.

```
FUNCTION DoReport
* This function creates the cursor of all the output
LOCAL lcOwner

lcOwner = Session.GetSessionVar("Owner")
SELECT * ;
      FROM Invoices ;
      WHERE OwnerName = lcOwner ;
      INTO TABLE (lcOwner+"Invoices") ;
      ORDER BY InvNo

this.ShowRpt(1)

ENDFUNC
*--------------------------------------------------------------------------
FUNCTION ShowRpt
* This function displays the output in reasonable sized chunks
```

```
LPARAMETERS tnRecord
LOCAL lcOwner, lcCmdSubmit

IF EMPTY(tnRecord)
      tnRecord = VAL(Request.Form("txtRecord"))
ENDIF
lcOwner = Session.GetSessionVar("Owner")


IF NOT This.OpenTable(lcOwner+"Invoices")
      Response.ErrorMsg("Error opening table", ;
            "Could not open invoices report table",,5,"DemoOptions.ism")
      RETURN
ENDIF

lcCmdSubmit = UPPER(Request.Form("cmdSubmit"))
DO case
CASE EMPTY(lcCmdSubmit) OR lcCmdSubmit = "NEXT"
      IF tnRecord > RECCOUNT()
            GOTO bottom
      ELSE
            GOTO tnRecord
      ENDIF
CASE lcCmdSubmit = "PREVIOUS"
      tnRecord = tnRecord-41
      IF tnRecord-41 < 1
            tnRecord = 1
      ENDIF
      GOTO tnRecord
CASE lcCmdSubmit = "RETURN"
      this.DemoOptions()
      return
ENDCASE

WITH Response
      .writeln(this.StdHeader())

      .writeln([<form method="POST" action="ShowRpt.ism">])
      .writeln([   <center>])
      .writeln([   <font size="3"><b>Invoices for Owner: ;
             ]+ALLTRIM(OwnerName)+[</b></font><p></p>])
      .writeln([   <table>])
      .writeln([   <th>Invoice#</th><th>Date</th><th>Total</th>])

      SCAN WHILE RECNO() <= tnRecord+20
            .writeln([      <tr>])
            .writeln([         <td>]+TRANSFORM(InvNo) ;
                        +[</td><td>]+DTOC(InvDate) ;
                        +[</td><td align="right">] ;
                       +STR(InvTotal,10,2)+[</td>])
            .writeln([      </tr>])
      ENDSCAN

      .writeln([   </table>])
      .writeln([   <input type="hidden" name="txtRecord"] ;
                  +[ value="]+TRANSFORM(tnRecord+21)+[">])

      .writeln([   <p><input type="submit" name="cmdSubmit" value="Next">])
      .writeln([      <input type="submit" name="cmdSubmit"] ;
                  +[ value="Previous">])
      .writeln([      <input type="submit" name="cmdSubmit"] ;
```

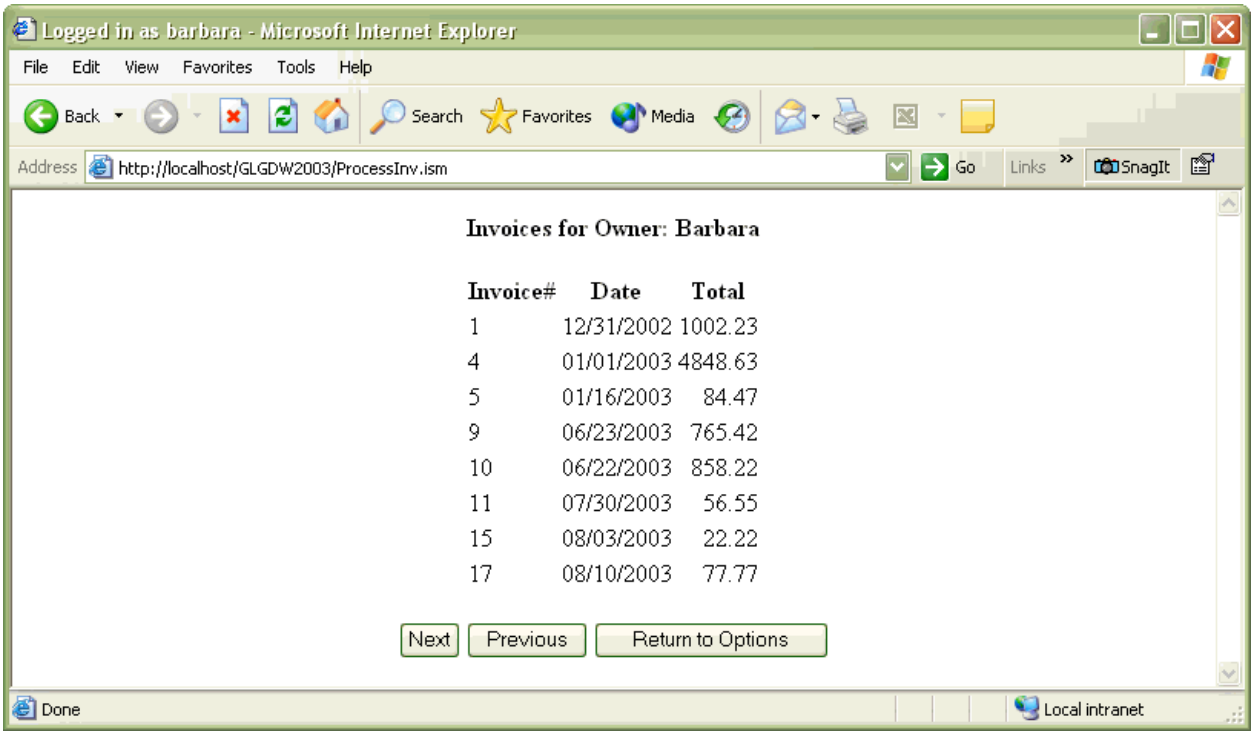
```
                    +[ value="Return to Options"></p>])

      .writeln([   </center>])
      .writeln([</form>])

      .writeln(this.StdFooter())
ENDWITH

ENDFUNC
```

This code will produce the following output on the browser:



The code above uses an alternative method for sending output to a browser. Previously, we saw an example where we had a separate file, with HTML and ASP style tags that specified what the browser displayed. Here, we are using the Response Object's WRITELN method to send output directly to the browser from within our VFP code. (Other tools have similar techniques for doing this.) No separate file is needed for this technique. The technique you choose depends on your specific circumstances. Having a separate file makes it easier to install changes to your screens without having to shutdown the server. Separate files are also easier to hand off to a graphic designer. Separate files are also create more pieces to keep track of, and must be in the virtual directory instead of your application directory, which means they are more accessible to the world.

There are a few choices you have with creating a cursor or table for your report. The most important thing is that you uniquely identify the output for the current user, so that if another user tries to run the same report, they don't overwrite the first user's results. Remember that if you

use a cursor, it is not unique for the user because VFP is running on the server, not the workstation. I like to use a combination of the user's login name and the report name.

One problem with using cursors is knowing when to close them. Remember this is a stateless system, so the user may request a report and then go off to another site, or close their browser. You could end up with a lot of cursors open this way. One way around this problem is to create a session variable that is a string of cursor names the user has open. After a reasonable period, you need to delete the session, but first, check the list of cursor names, and close all those cursors.

A second technique is to create tables instead of cursors, and write these tables to a temporary directory. You would then probably want some kind of maintenance routine that periodically runs on the server, and deletes all files in the temporary area that are over a certain age.

Conclusion

As you can see, developing web applications requires that you think of things in a somewhat different light. **The good news is that VFP is well-suited to creating web applications if you have the right tools.** Most of your existing code can be used, with only the front-end being replaced.

References

Here are some links that will help you as you venture into web development with VFP:

Commonly Available Web Tools

The following tools are the commonly available tools for creating web applications with VFP:

- Web Connection (www.west-wind.com)
- Active FoxPro Pages (www.afpages.com)
- ActiveVFP (www.dotcomsolution.com)
- FoxWeb (www.foxweb.com)

Info and Other Tools

- The W3C site is the site of the World Wide Web Consortium, the standards committee for everything related to the web. This site has many useful tools (like an HTML validator), and specifications: http://www.w3.org/
- Accessible web page design: http://www.makoa.org/web-design.htm
- Cross-browser compatibility testing: http://www.netmechanic.com/browser-photo/tour/BrowserList.htm
- and: http://www.browsercam.com/
- A chart of browser features: http://hotwired.lycos.com/webmonkey/reference/browser_chart/ (This site has some other useful tools, like an HTML quick reference and a color chart.)

Sites for PDF writers

- Adobe: http://www.adobe.com/
- Amyuni: http://amyuni.com/index.php
- PDF995: http://pdf995.com/
- Minds Eye Report Engine: http://www.mindseyeinc.com/ReportEngine/index.htm

An Internet State of Mind
Part 2 – Show me the Code

Session Number

Barbara Peisch
Peisch Custom Software, Inc.
3138 Roosevelt St. Suite O
Carlsbad, CA 92008
Voice: 760-729-9607
Fax: 760-729-9608
Email: barbara@peisch.com

Overview

You need to move a VFP application to the web. You know in theory what you're supposed to do, but you've never done this before. What are the problems you're likely to run into? How do you debug them? What security issues are you likely to face? How will you host this application? This session will address all these issues and more.

Caveats

In order to run the examples that come with this session as is, you'll need a copy of Web Connection. The demo version of Web Connection should work just fine. If you're using one of the other tools, you may need to make some minor changes to the code. Please refer to the instructions with the tool of your choice to determine how to configure and run an application in that environment.

For this demo, you'll need to define Login.htm as a document in IIS for the virtual site. In the source code that comes with this session, the "WebContent" subdirectory is the path for the virtual site, and all files in that subdirectory belong in the virtual site.

I have not included the wc.dll file with the source code because that file is licensed. It would go in the WC subdirectory under the WebContent subdirectory. The application also expects a script map defined in IIS, called ISM, which points to this wc.dll file.

The path set in the Config.FPW file assumes that your copy of Web Connection is in a subdirectory of the application called WCONNECT. Be sure to change the path in the CONFIG.FPW file to reflect your environment.

Before your application can run…

There are a number of things that can go wrong before you even get your application to run. These are not necessarily programming problems, but problems with setting up your development environment.

You may start up your application, start up your browser and try to access the web site. You may find that the server isn't even seeing requests from the browser. This could be the result of any of a number of problems:

1. You have to make sure you've got IIS installed on your development machine, and that you've setup a virtual directory for the new application, with the appropriate permissions set. If you're using any script maps in your application, you need to make sure those are properly configured in IIS as well. That would include your virtual directory and possibly a temporary directory, but NOT your application directory.
2. If there are any directories used by the IUSR_machine account (the Internet guest account on your machine), you need to give the appropriate permissions to the IUSR_machine account for those directories. You should NOT give IUSR_machine any access to your application directory or data directory.
3. You may need to specify a path in your application and/or your ISAPI connector, which must match. Check that they do.
4. Depending on the mechanism you're using for messaging, other problems may arise, such as COM configuration. You may need to consult with others using your tool of choice to determine what your problem may be.

Once your server is seeing requests from users, you may also find that users aren't seeing the response from the server. This isn't as common as the server not seeing the request, and this one is usually a programming problem of some kind. Most often, your application isn't sending the output you think it is, and tracing through the code will often reveal where this problem occurs.

Debugging Web Applications

The good news is, tracking down programming problems in a VFP web application doesn't have to be any more difficult than tracking down programming problems in a LAN application. In fact, it's often much easier because testing your changes are often just a browser "refresh" away. As with your LAN application, you want to set a flag that tells the application whether or not you're running in development mode. If you are running in development mode, any errors encountered while running your application will be handled by VFP in the same way it does with a LAN application—by displaying an error dialog and offering the choices of "Cancel", "Ignore" or "Suspend". You can suspend and open the VFP debugger, and see all the same kinds of things you're used to seeing when you debug. You can even put a SET STEP ON anywhere in your code to bring up the debugger at any time. Just keep in mind that you probably want to click on

the “Stop” icon in your browser, so the browser doesn’t timeout while you’re looking around in the debugger. The “back” button often won’t work, depending on the browser’s configuration.

One really nice feature of web apps is that your server application doesn’t have to be running between hits from the browser. (Just try that with a LAN app!) You can leave the browser up on a screen you’re ready to submit, make some changes in your application, re-start the server application, and hit the submit button in the browser.

Typical Programming Errors

Aside from not sending output back to the user, here’s a list of errors a new web developer is likely to make.

Using Variables or Properties to Store Info Between Hits

Unlike the typical LAN application, where VFP is running on each workstation, in a web application, VFP is only running on the server. In a LAN application, it’s common practice to store all kinds of settings in an application property or even global variables. In a LAN application, those are unique to the workstation. Because VFP is only running on the server, for a web application, this is a very bad practice. You may set a variable or property to something for one user, and the next request could be from a different user, who then sees the value set by the previous user.

To run the example that demonstrates this problem, first login as Barbara. Click on the “Public Variable” link and make note of the owner shown. Open a second instance of your browser and login as Tom. Click the “Public Variable” link and note that owner in this case is “Tom”. Go back to the first instance of your browser, and click on the “Public Variable” link again, and note that the owner that had shown, “Barbara” for this user is now showing “Tom”.

In the code example for this session, I set a global variable to the name of the user who logs in, during the Login function. When the next user logs in, that variable is changed. When the first user comes back and queries the contents of that variable, the value is incorrect for that user. The Login function also creates a session variable called “Owner”, and the ShowOwner method in the code has a section commented out that shows the correct way to deal with this situation.

Using the Same Cursor Name for All Users

In a LAN application, when we want to display a subset of information, we typically use a cursor. Cursors are unique to the workstation in a LAN application. Once again, because we are only running VFP on the server, this is not the case in a web application.

To demonstrate this problem with the sample code, click on the “Two users one cursor” link from the options, while logged in as “Barbara”. Step through the invoices shown using the “Next” and “Previous” keys, and note that the owner shown for all invoices is “Barbara”. From the instance of your browser logged in as “Tom”, click on the “Two users one cursor” link. Step through the invoices shown and note that all the invoices are for the owner, “Tom”. Now go back to the instance logged in as “Barbara”, and step through the invoices. All the invoices now say the owner is “Tom”! What happened is that after “Barbara” created the cursor for her

invoices, “Tom” chose the same option and overwrote the “Barbara” cursor with invoices for “Tom”.

This code actually demonstrates bad practice in a couple of ways. Not only does it use one cursor that can be overwritten by other users, but it also makes assumptions about the current record pointer. When you click on “Next” or “Previous”, it simply does a SKIP or a SKIP -1. It would be better to send the current record number or key back to the browser as a hidden variable, then read that on the next hit and position the record pointer accordingly.

So, what do you do to avoid the problem of interfering data sets? Let’s look at some alternatives.

Requery on each Hit

The best option, if the query doesn’t take too long to run is to re-run the query for each hit. Send this output to a cursor (the name can be the same for each hit), format the output, then close the cursor. The advantage to this technique is there are no cleanup issues, and it’s fairly scalable. If queries take too long to run, however, this option is not feasible.

Using Cursors

You can use cursor for temporary data, but you must use a naming scheme where users won’t interfere, or even multiple instances of the same user running different queries won’t interfere. Something that may work in your situation is to name the cursor for the user and function.

But remember, this is a stateless environment. What if the user makes a request of the server and goes off to another site before the results are complete? What if they only view some of the results. How do you know when they’re done? You could be left with an awful lot of open cursors hanging around, without knowing if it’s safe to close them.

One alternative to this could be to create a session variable that stores the names of all cursors that user has open. You should periodically review open sessions, and delete any sessions that are over a certain age. (That age will vary widely with the type of application.) During this process, you could retrieve the list of cursors open for any session you’re about to delete, and close those cursors.

The biggest problem with this technique is that it’s the least scalable. If you need to run more than one instance of your server, which most apps require, using cursors won’t work. A cursor exists on a specific server, and there’s no guarantee which server will get the user’s next hit.

Using Tables

Temporary tables are a much cleaner approach than cursors, but should be used sparingly because of cleanup issues. Again, you need a naming scheme that won’t interfere with other users or other sessions. You could use SYS(2015) or some other naming convention. You can safely close the table after creating the response for the user, then reopen it again for the next request. Reopening a recently closed table is fairly quick on Windows 2000 Server and later

servers because of caching.[2] Your routine maintenance would then include a function that scanned through all the tables in the temporary area, deleting any beyond a certain age.

Error on the Server

In your LAN applications, you probably have some technique for handling errors that involves storing information about the error, then terminates what it was doing when the error occurred. You may try to RETURN TO MASTER, or THISFORM.RELEASE(), or maybe you just quit the application completely. There isn't really a "MASTER" to return to in a web application. Nor is there a "Form" to be released. You may want to return to a "home page", if your application has one. There are two things you must do in a web application when an error is encountered:

1. Send a display to the user that there was an error of some kind, optionally giving information the user can give to the web master
2. Keep the server running!

You don't want to put up any kind of display on the server that requires human intervention. This will effectively hang your server! That means no WAIT WINDOWs, unless there's a very short TIMEOUT clause, and no MESSAGEBOXes, also unless there's a very short timeout parameter. Chances are, no one will see these anyway, so it's best not to use them at all.

Put up some kind of display, letting the user know there was a problem. You can let the user navigate to another place in the application, or send the user somewhere else after a short timeout, but do let the user know what happened.

In the code sample, the method called "ServerError" shows what happens when an error goes unhandled on the server. You can run this method from the "Error on the server" link from the demo options. You'll see how the error is displayed on the server, and waits for human response.

The code sample also has a section which is commented out that shows a better way to handle an error. A message is sent back to the user, and then the user is sent back to the demo options screen after a 10 second timeout.

Status or Progress Messages on the Server

A similar mistake to the error handling situation discussed above is the display of status or progress messages on the server. Although these don't hang the server, the user may think the server is hung. Using typical LAN techniques for displaying progress or status messages in a web application will only display on the server, and the user won't have a clue about what's happening. In fact, if the process takes any appreciable time, the user's browser may even timeout waiting for a response. In addition to that, the time the server is taking to run this process, it's unavailable to handle any other hits.

---

[2] Thanks to Mike Cummings for making this observation.

To see an example of this problem, click on the "Status message on the server" option in the example program.

One solution could be to send these requests to another server, which generates results and sends them by e-mail to the user. This is fine in some situations, but not always.

What if the process being performed is crucial to the current online session—the user can't leave and get their results later? The answer to this situation is not simple. The problem is that in a stateless system, once a response has been sent to a user, there's no mechanism to keep the server running a process, or an automatic way for the browser to get the results back later.

The solution is to use an asynchronous process to run the request. For this session, I use an example created by J. Randy Pearson for the book, "WebRAD: Building Database Applications on the Web with Visual FoxPro and Web Connection"[3] which has been modified so that the application with the book is not required.

To run the example, find the "StatusMessage" method in the code and comment out the code running there now. Remove the comment from the call to "This.SlowTasks()", and run the "Status message on the server" option of the demo again.

How does this function work? There are three pieces required:

1. The front end (the user running a browser)
2. The current server application
3. A second server application for processing slower requests—called the "Async Server"

The process goes like this:

- The front end makes a request of the server
- The server identifies this task as one that takes some time, and creates a record in an asynchronous request table (the "async table"), with a unique ID for the job.
- The server sends a status message back to the browser, with a timeout that triggers the browser to query the server again in a short while, sending back the job ID with the request. (This is done in HTML with a <METAHTTP-EQUIV="Refresh" CONTENT="…"> tag that specifies the timeout period.) Note that after this, the main server is ready to receive requests from other users.
- A server running in the background is dedicated to running these longer tasks. It periodically checks the async table to see if any new requests have been added. When a new record is found, it makes a note in the record that the process has been started. Periodically, this server will update the status in the record with new information. When the process is completed, the status is changed to reflect this, and the results are put into the job's record.
- In the meantime, the front end sends a request about the job status to the server again.

---

[3] Available from www.hentzenwerke.com.

- The server checks on the status of the job in the async table, and returns the latest progress information to the browser, with another trigger to requery the server again at a later time.
- Eventually, when the browser queries the server, the server finds the job has been completed, and send the results that are found in the async table back to the browser.

As you can see, this is one way to handle longer requests without causing the browser to timeout, and keeping the user informed of the progress.

Displaying Anything on the Server Requiring Human Intervention

I can't stress this point enough. Anything that displays something on the server that requires a response from a person will hang your server. This includes the dreaded open file dialog, which can't be trapped with error handlers. I have a method called OpenTable that I always call before attempting to do anything with an alias on each hit. You must use some form of this mechanism to protect yourself from the open file dialog.

There is a command called SYS(2335), which sets unattended server mode in an application. Be careful of how you use this! This command is only intended for situations where you are running your application as a COM server. Turning on unattended server mode in an EXE that you run normally will cause your application to crash when it's not started from the VFP command window!

URL Hacking

URL Hacking is when someone alters the contents of the address bar in their browser to make calls to your application in ways you didn't intend. This idea is probably completely foreign to you if you've only been dealing with LAN applications, but it's a very real issue with web applications. This is just one more reason why you have to carefully check all input from the user.

To run the URL Hacking example, choose the "URL Hacking" link from the demo options. What you'll see is a slightly different version of the Invoices screen we used in Part 1 of this session. Here's the code for this version of the Invoices screen:

```
<html>
<head>
<title>Invoices</title>
</head>
<body>
<h1 align="center">Invoices</h1>
<center>
  <p>Invoice#: <input type="text" name="txtInvNo"
       value="<%= iif(TYPE("oInvoices")='O',oInvoices.InvNo,'0') %>"
       size="20"></p>
  <p>Owner: <input type="text" name="txtOwner"
       value="<%= iif(TYPE("oInvoices")='O',oInvoices.OwnerName,'') %>"
       size="20"></p>
```

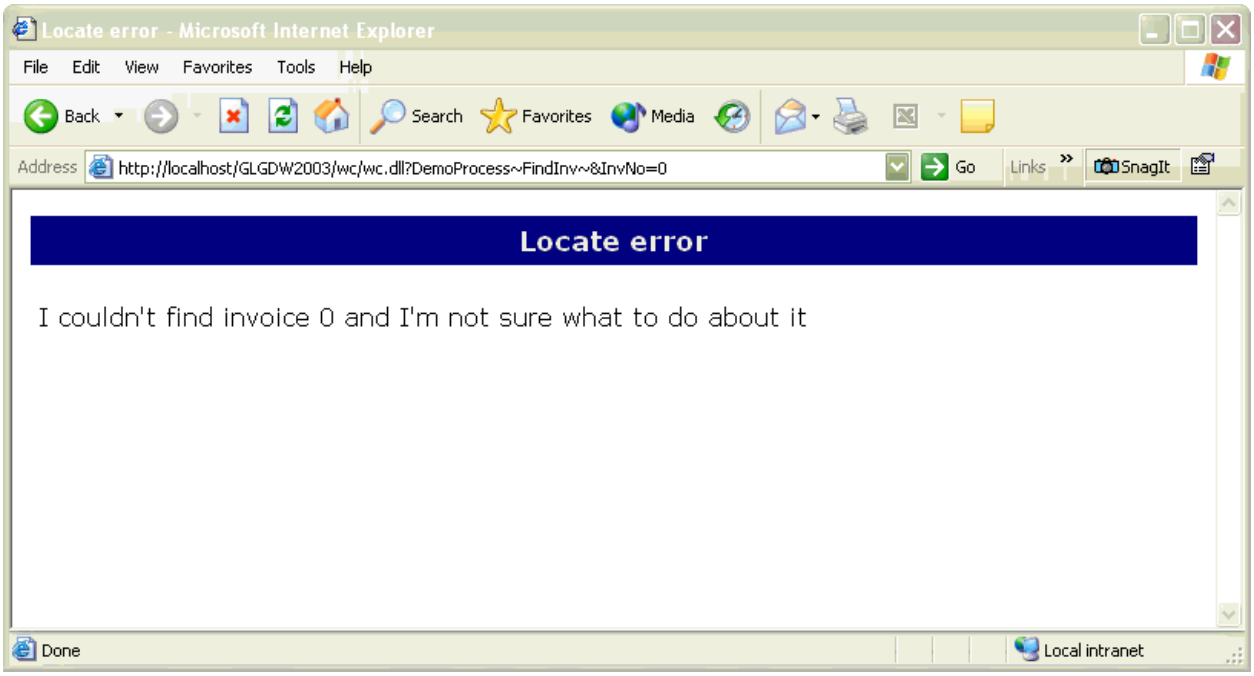
```
  <p>Date: <input type="text" name="txtInvDate"
       value="<%= iif(TYPE("oInvoices")='O',oInvoices.InvDate,'  /  /    ')%>"
       size="20"></p>
  <p>Total: <input type="text" name="txtInvTotal"
       value="<%= iif(TYPE("oInvoices")='O',oInvoices.InvTotal,'0') %>"
       size="20"></p>
  <p>
  <a href="FindInv.ism?InvNo=0">Find</a>
   
  <a href="DelInv.ism?InvNo=
      <%=iif(TYPE("oInvoices")='O',oInvoices.InvNo,'0') %>">Delete</a>
   
  <a href="SaveInv.ism?InvNo=
      <%= iif(TYPE("oInvoices")='O',oInvoices.InvNo,'0') %>">Save</a>
   <a href="DemoOptions.ism">Return to Options</a>
  </p>
</center>
</body>
</html>
```

And here's the screen this code will show:

The difference between this form and the one we used in part 1 is that instead of buttons across the bottom, we have hyperlinks. The hyperlinks call exactly the same functions we used for the buttons in part 1, except that with the hyperlinks, you can see what those functions are. When I took this screen shot, I had the mouse cursor over the "Find" hyperlink. **Notice that the status bar at the bottom of the screen shows exactly where this hyperlink will take you, and with which parameters.** Note that the exact syntax used to call your code from a hyperlink may

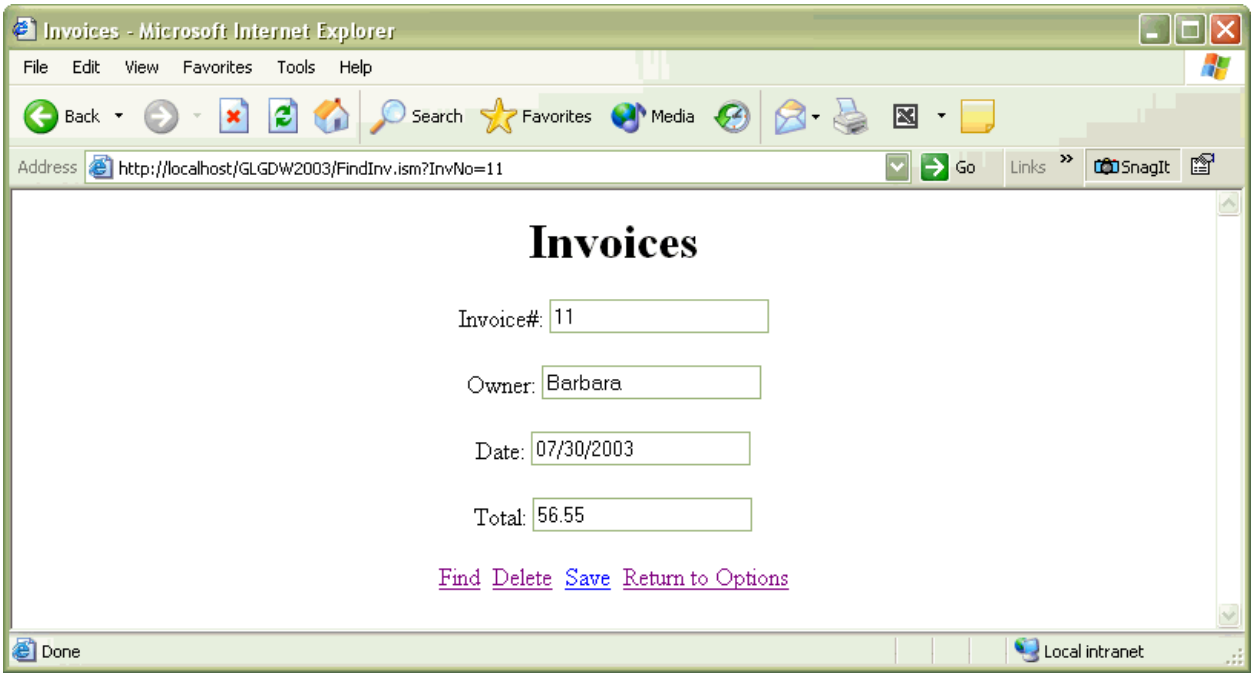
differ, depending on which web tool you are using. Please refer to the documentation for your tool of choice for the syntax you must use.

There's a deliberate bug in the "Find" hyperlink of this application. When I try to enter an invoice number to find in the "Invoice#" field and click on the "Find" link, I get an error message, as shown below.



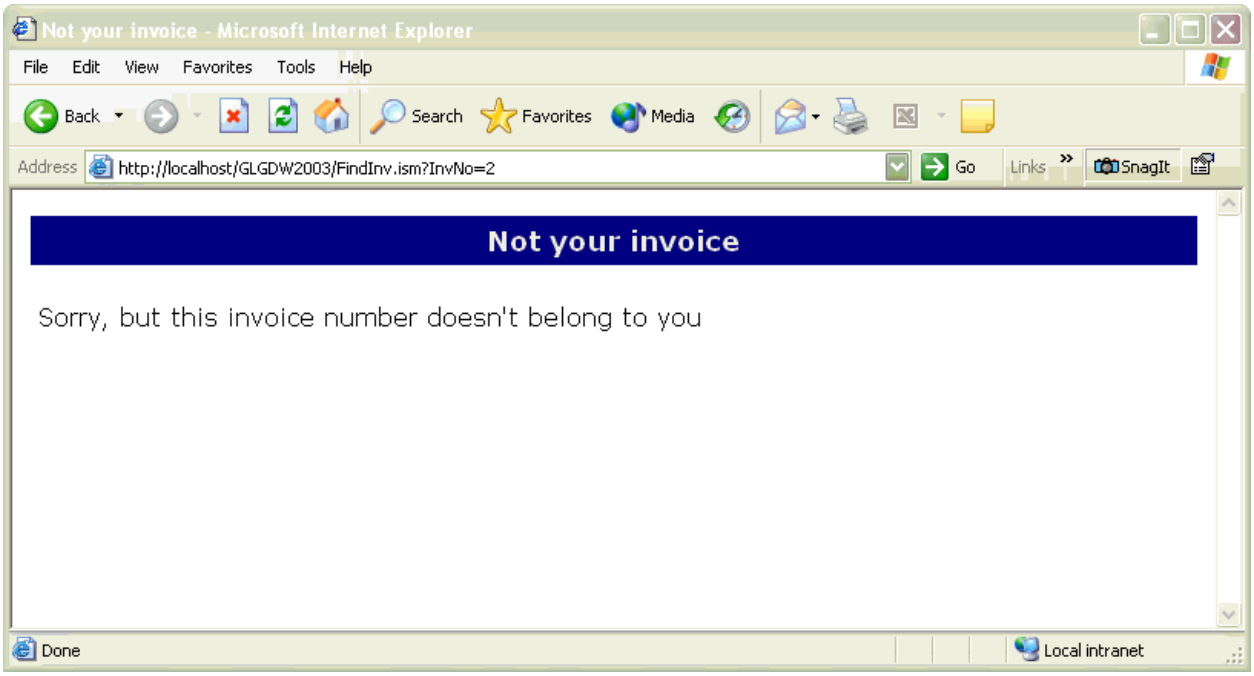
You may see this message and say to yourself, "Wait a minute! I entered invoice number 11, not 0! Why is it trying to find invoice 0? But wait! Look at the address bar! It's showing &InvNo=0. What if I just change the 0 to 11 and press the Go button?"

Good question. Go ahead and try it. Guess what? You get the invoice screen for invoice number 11!



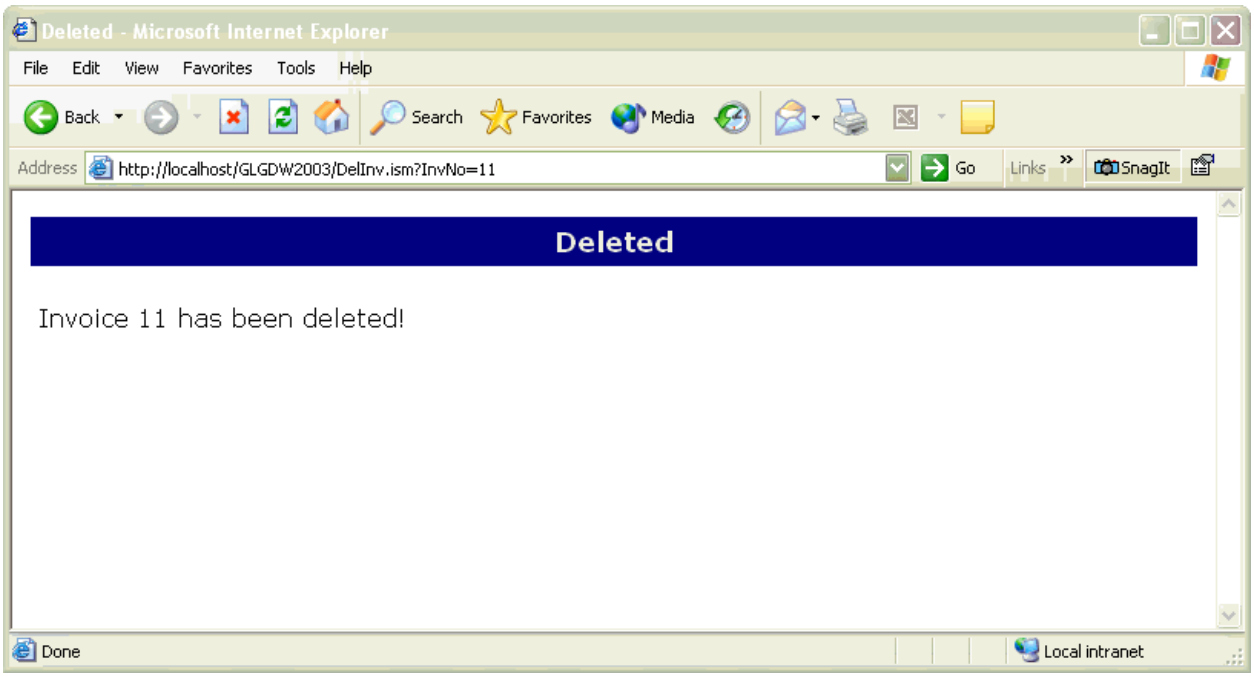
This URL is now also recorded in your browser's history, so you can use the dropdown list to easily get back to the URL at any time. This is probably not something you want your users doing, but there's nothing you can do to prevent it.

Now, let's try something else. Let's try to find invoice 2 using the same hacking technique we used to find invoice 11. While viewing invoice 11, change the address bar so that the 11 is now a 2, and click the browser's Go button.

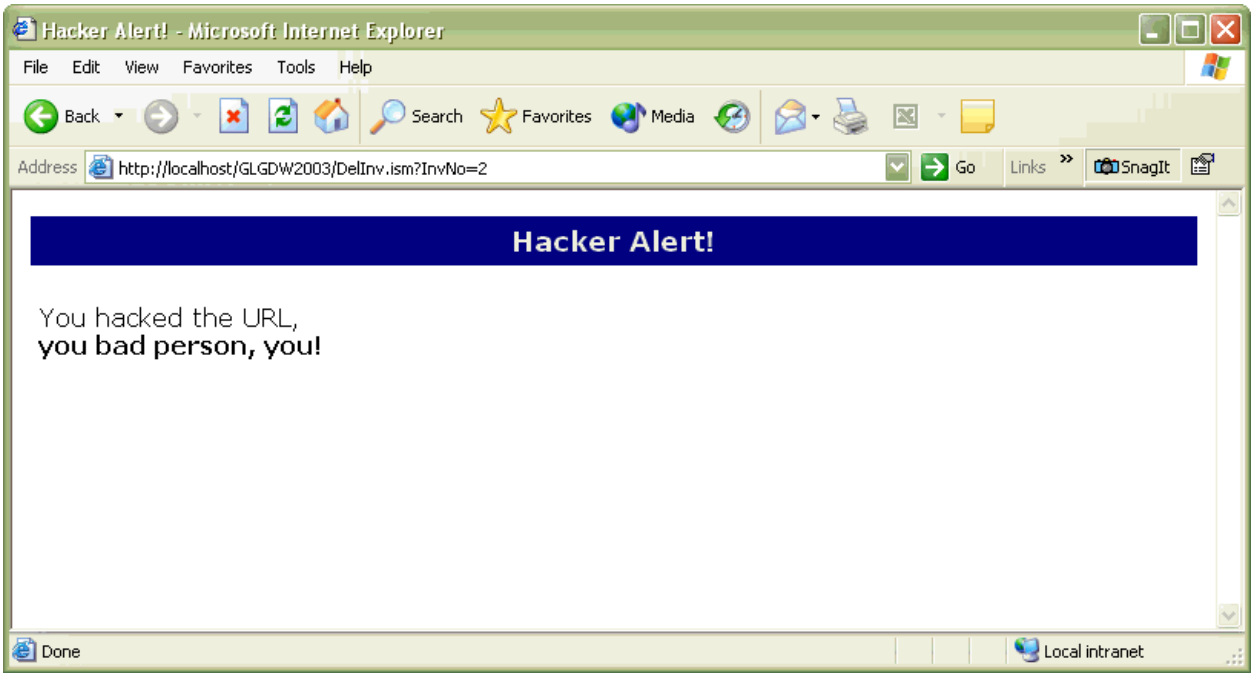


We check the owner of the invoice in the Find function, and in this case, invoice 2 is for a different owner. After 5 seconds, this message will send you back to the demo options. From there, click on the "URL Hacking" link again.

Use the history drop down list of your browser to go back to viewing invoice 11, and click on the "Delete" link. You'll see the following confirmation that the invoice was deleted.



Take note of the URL in the address bar. After 5 seconds, this message will return to the demo options. Enter the same address in the address bar as is shown above, except change the 11 to 2, then click the "Go" button. You'll get the following message.



Remember that when we tried to find invoice 2, it said that invoice didn't belong to us? If you use the program properly, you should never be able to see invoice 2, and therefore, wouldn't be able to delete it by clicking on the "Delete" link. If I had programmed with that assumption in mind, I probably wouldn't have bothered to check the owner in the delete function. This is another case where you must check input from the user, even if you think there are restrictions on what they can do. The only way you could possibly be trying to delete an invoice that doesn't belong to you is if you hacked the URL.

Let's take a closer look at the code in the DelInv function.

```
FUNCTION DelInv

IF not this.OpenTable("Invoices")
      RETURN
ENDIF
lnInvNo = VAL(Request.Form('txtInvNo'))
IF EMPTY(lnInvNo)
   lnInvNo = VAL(Request.QueryString("InvNo"))
ENDIF
LOCATE for InvNo = lnInvNo
IF OwnerName <> Session.GetSessionVar("Owner") AND NOT EOF()
      this.ErrorMsg("Hacker Alert!",
       "You hacked the URL, <br><b>you bad person, you!</b>",
         ,15,"DemoOptions.ism")
ELSE
   IF EOF()
      this.StandardPage("Locate error",
          "I couldn't find invoice "+ ;
          TRANSFORM(lnInvNo)+" and I'm not sure what to do about it",
          ,5,"DemoOptions.ism")
   ELSE
```

```
      DELETE
      this.StandardPage("Deleted", ;
      "Invoice "+TRANSFORM(lnInvNo)+ ;
      " has been deleted!",,3,"DemoOptions.ism")
   ENDIF
ENDIF

ENDFUNC
```

The first thing that may strike you as odd is that this function calls both the Request.Form method and the Request.QueryString method to find the invoice number we're looking for. This is because exactly the same code is used for both the invoice form with the buttons and with the hyperlinks. You must use Request.Form to retrieve the value of txtInvNo, which is submitted with the form, but when the code is called from a hyperlink, no form has been submitted. All the information available must be sent to the function in the hyperlink itself. Use the Request.QueryString method to retrieve values from the hyperlink. You may recall that the end of the hyperlink had "?InvNo=2". The question mark indicates the start of parameters in the query string, and "InvNo" is the first parameter named. When using named parameters, as we are here, you must pass the name of the parameter whose value you want to retrieve when you call the Request.QueryString method.

Looking a little further down in the code, you'll see that once again, the code is checking if the owner of the invoice matches the user who is logged in. If not, the "Hacker Alert" message is displayed.

Protecting Your Application from URL Hacking

What can you do to protect you application from this kind of hacking? To start with, don't use hyperlinks within your application unless you have to. Remember, hyperlinks reveal more than you want the user to know about your application with what it displays on the status bar and the address bar. If you do use hyperlinks, you should consider using some kind of encoding technique so that paths into your application aren't quite as obvious.

The second thing you must do is to program defensively. Don't make any assumptions about the conditions under which a user got to any point in your code.

When you do use links, send an absolute minimum of information in the query string. Any information about the user or session should be stored as session variables that are saved on the server, not sent in the query string. This is an argument why the use of cookies can be more secure than not using them. The use of cookies allows you to store information in session variables on the server instead of having to pass the information to the server on each hit, where it could be intercepted by a hacker.

If a user hacked the URL to run something they would have legitimate rights to run through normal means, you may not be able to detect the difference from within your code, but it shouldn't matter in this case.

Does this mean that by using a form's submit and POST operation that you're safe from this kind of hacking? Certainly not! You must always be defensive and check input! The next section drives home the reason why.

SQL Injection

SQL Injection is a term used to describe a technique for entering commands into places in your application that accept user input, but which create effects you didn't intend. This is best described with an example.

Create a file called Junk.txt in the application directory in which the demo for this session is running. The contents of this file are irrelevant. We just want the file to be around. Open Windows Explorer and display your application directory. You want to be able to see the Junk.txt file.

Run the demo and select the "SQL Injection" link.

When the screen comes up that asks for the owner name to display, enter this:

```
barbara] and trans(execscript("ERASE JUNK.TXT"))=[
```

Press the Submit button, and the listing of invoices will display. But if you're watching the Junk.txt file in Windows Explorer, you'll see that it disappears. Be aware, the junk.txt file that disappeared was in our application directory, not the virtual directory. This is a directory that the Internet guest account has no rights to, and access is completely controlled through the application. Yet, a user on the web was able to erase a file in this directory.

How did this happen?

Let's take a look at the code that gets executed when you click the submit button from this screen.

```
FUNCTION SQLInjection
LOCAL lcOwner, lcOutput, lcWhere

lcOwner = UPPER(Request.Form("txtOwner"))

IF EMPTY(lcOwner)
      lcWhere = []
ELSE
      lcWhere = "where UPPER(OwnerName) = [" + lcOwner + "]"
ENDIF

SELECT * ;
      FROM Invoices ;
      &lcWhere ;
      INTO CURSOR TmpInvoices

IF _tally > 0
      lcOutput = this.StdHeader()
      lcOutput = lcOutput + [<center>] + CRLF
      lcOutput = lcOutput + [<br>Invoices for ] + lcOwner + [:] + CRLF
      lcOutput = lcOutput + [<br><table>] + CRLF
      lcOutput = lcOutput+[<th>Invoice</th><th>Date</th><th>Amount</th>]+CRLF

      SCAN
            lcOutput = lcOutput + [<tr>] + CRLF
            lcOutput = lcOutput + [  <td>] + ;
```

```
                  TRANSFORM(TmpInvoices.InvNo) + [</td>] + CRLF
            lcOutput = lcOutput + [  <td>] + DTOC(TmpInvoices.InvDate) + ;
                  [</td>] + CRLF
            lcOutput = lcOutput + [  <td align="right">] + ;
                  STR(TmpInvoices.InvTotal,10,2) + [</td>] + CRLF
            lcOutput = lcOutput + [</tr>] + CRLF
      ENDSCAN

      lcOutput = lcOutput + [</table>] + CRLF
      lcOutput = lcOutput + [</center>] + CRLF
      lcOutput = lcOutput+[<form method="POST" action="DemoOptions.ism">]+CRLF
      lcOutput = lcOutput + [<p align="center">] + ;
            <input type="submit" value="Done"></p>] + CRLF
      lcOutput = lcOutput + [</form>] + CRLF
      lcOutput = lcOutput + this.StdFooter()
      Response.Write(lcOutput)
ELSE
      this.StandardPage("No invoices found", ;
      "There are no invoices for the owner entered",,5,"DemoOptions.ism")
ENDIF

ENDFUNC
```

In the code above, pay close attention to how the user input is handled. Notice that we just accept the user's input unchecked and unfiltered, and incorporate it into the WHERE clause used in a SQL SELECT statement. When this expression is macro-expanded, any command the user (or should I say "hacker") has typed in gets executed. In this example, the WHERE clause ended up like this:

```
"where UPPER(OwnerName) = [" +barbara] and trans(execscript("ERASE
JUNK.TXT"))=[ + "]"
```

Now, just imagine if instead of "ERASE JUNK.TXT", this command had contained "ERASE *.*" or "FORMAT C:". That should scare you!

Note that this is not a VFP problem, nor a problem with the web tool. It's a programming problem. Your LAN apps are subject to this kind of abuse too, but you probably haven't run into it because your application isn't exposed to such a wide audience, and therefore isn't susceptible to hackers.

There are a couple of things you can do to protect yourself from this kind of thing.

This wouldn't happen if you didn't include the "WHERE UPPER(OwnerName) =" part of the command in the variable. In other words, if your SQL Statement looked like the example below, you wouldn't have this problem.

```
SELECT * ;
      FROM Invoices ;
      WHERE UPPER(OwnerName) = lcOwner ;
      INTO CURSOR TmpInvoices
```

But this is a simple case, and you may need to build more complex and dynamic WHERE clauses that require macro expansion. Macro expansion in any form, or use of the EVALUATE function,

could open you up for problems if you don't check the contents. Even with the original syntax, you could stay out of trouble by doing a couple of checks on the input.

First, in this application, the owner name is a 10 character field. There is absolutely no reason the input should exceed that length. So, the first thing we should do is check the length of what's been entered. If it's greater than the maximum length of the field we're matching it to, we must reject the input.

The second thing we should do is see if there are any characters in the input that are invalid for the field. For example, there's no reason our owner field should have characters like *&%^$#@, etc.

With these additions, the start of our function now looks like this:

```
FUNCTION SQLInjection
LOCAL lcOwner, lcOutput, lcWhere

lcOwner = UPPER(Request.Form("txtOwner"))

DO case
CASE LEN(lcOwner) > 10
      this.ErrorMsg("Invalid entry","Invalid length for owner")
      RETURN
CASE CHRTRAN(lcOwner,"<>][?~\!@#$%^&*()+=_'","") <> lcOwner
      this.ErrorMsg("Invalid entry","Invalid characters in owner")
      RETURN
CASE CHRTRAN(lcOwner,["],[]) <> lcOwner
      this.ErrorMsg("Invalid entry","Invalid characters in owner")
ENDCASE

IF EMPTY(lcOwner)
      lcWhere = []
ELSE
      lcWhere = "where UPPER(OwnerName) = [" + lcOwner + "]"
ENDIF
```

The case statement that's been added starts by making sure the input isn't longer than 10 characters. Depending on the sensitivity of your application, you may not even want to let the user know that the input is an invalid length, but simply reject it as "Invalid input".

The second case in the statement checks for most of the invalid characters that could be entered. We must delimit the string of invalid characters somehow, and here we're using double quotes to delimit the string. But the double quote character is also one we don't want in the input, therefore, we have a third case that's looking specifically for the double quotes in the string, using an alternate delimiter.

You probably don't want to have to repeat code like this everywhere you need to check the input, so it's best to create a centralized routine you can call.

For a much more involved discussion of SQL Injection and how to prevent it, read the article at http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf.

Microsoft has created a lockdown tool for IIS which you can read about at http://www.sans.org/rr/papers/65/301.pdf. You need to be aware of the issues discussed in this paper, whether you use the lockdown tool or not to secure your web server.

Conclusion

In this session, I've discussed some common mistakes when creating web applications as well as some easily missed precautions you should take for web applications. I hope I've made you realize that developing applications for the web requires a very different mindset from the kind of application you're probably used to writing. I've barely scratched the surface here, when it comes to security issues, but you should now have an idea of the some of the kinds of issues you'll be facing and how to deal with these.