



You are here: [Home](#) > [Developers](#) > [View Source Articles](#) > JavaScript View Source Article

- Developer Program
 - [Membership](#)
 - [One-to-One Support](#)
 - [Newsgroups](#)
- Developer Publications
 - [View Source](#)
 - [Developer News](#)
- Documentation
 - [Technical Manuals](#)
 - [White Papers](#)
 - [TechNotes](#)
 - [Sample Code](#)
 - [FAQs](#)
 - [Books](#)
- Technologies
 - [CORBA](#)
 - [Directory & LDAP](#)
 - [Dynamic HTML](#)
 - [Java](#)
 - [JavaScript](#)
 - [Linux](#)
 - [Security](#)
 - [SSJS](#)
 - [XML](#)
- Developer Downloads
 - [Tools & SDKs](#)
 - [Patches](#)
- iPlanet Products
 - [Technical Resources](#)

Form-Data Handling in Client-Side JavaScript

By [Duane K. Fields](#)

[Send comments and questions](#) about this article to *View Source*.

In the *View Source* articles [CGI vs. Server-Side JavaScript for Database Applications](#) and [Form-Data Handling in Server-Side JavaScript](#), we learned how handling form data with JavaScript applications running on the server provides a more convenient, cross-platform alternative to traditional CGI applications. In this article, we'll explore a new technique for working with form data through JavaScript that operates entirely on the client.

The HTML form provides one of the most critical features of web-based applications, enabling the user to input information into the application. Traditionally, forms have been handled on the server, first by CGI applications in Perl or C, and increasingly through more efficient technologies such as server-side JavaScript (SSJS) and WAI. In some cases, however, it's possible to avoid the server altogether by processing the form data entirely **on the client**. Doing so dramatically reduces the complexity of deployment and configuration issues surrounding our application and enables us to implement interactive applications in environments that were previously inaccessible, such as when hosting through an ISP that doesn't provide sufficient access to develop server-side solutions.

We'll begin by discussing how handling form data on the client works, and why you might want to do this. We'll introduce a client-side JavaScript (CSJS) routine that you can use to emulate SSJS's request object, enabling you to handle form data on the client as easily as you could on the server or through a Perl CGI. To show the benefit of this technique, we'll present a fairly complex sample application at the end of this article.

THE CLIENT-SIDE APPROACH

In the client-side approach, we specify our form's action handler to be another HTML document instead of passing the data to a server-side script or application. As long as it's using the GET method, the HTML that contains the form definition doesn't require any changes or differences in design in order to work properly in a client-side implementation. Because of this, existing forms can easily be retrofitted to a client-side architecture, or you can use identical forms (with different actions) to perform the same duties in both server-side and client-side situations.

For example, we might see a form definition like this for a CGI:

```
<FORM ACTION="formhandler.cgi" METHOD=GET>
```

And like this for a CSJS handler:

```
<FORM ACTION="formhandler.html" METHOD=GET>
```

Since we're using the GET method of delivering data, the values from the form are then encoded as a string of value pairs appended to the action handler URL. The real work in implementing the client-side approach is in the HTML document that's the receiver of the form's action. You must extract the form data from the end of the URL and then perform whatever actions are needed on the data. Later, we'll look at a reusable JavaScript routine that makes accessing URL data a breeze.

WHY STAY ON THE CLIENT?

Why would you want to use the client-side approach to handling form data? In some situations you may have no choice. For many users, ISP access restriction may prohibit development of server-side applications. In addition, a client-side approach has a number of unique benefits. While SSJS, Perl, and CGI will always put some additional load on the server, a purely client-side application has no impact on the server beyond serving the HTML documents. A client-side approach can even be used to create an "offline" application that doesn't require a server at all, allowing a more flexible approach for users who are "on the go."

The client-side approach does have some limitations - it's not applicable in all cases. The client-side approach can't record data to a file or a database, for example. And since it uses the GET method of delivering form data, it's limited to transferring 4K of information.

In summary, the client-side approach has these advantages:

- can be used online or offline
- has the lightest possible impact on the server load
- doesn't require privileged access to the server
- reduces server configuration requirements

On the other hand, it has these disadvantages:

- doesn't enable persistent data
- can't handle large (> 4K) data transfers
- requires client to have JavaScript support
- makes program logic visible to end user

CREATING THE CLIENT-SIDE REQUEST OBJECT

A CSJS routine can be used to emulate a powerful feature of SSJS - the request object. If you're already familiar with programming in SSJS, you're halfway there. If not, let me explain. The request object is used to provide your application with information about the current transaction. The request object is initialized with a property for each named input element in an HTML form.

For example, consider the following fragment from an HTML form:

For example, consider the following fragment from an HTML form.

```
<FORM ACTION="handler.html" METHOD="GET">
Client Name:   <INPUT TYPE=TEXT NAME="name">
Company Name: <INPUT TYPE=TEXT NAME="company">
Delivery Zone: <INPUT TYPE=TEXT NAME="zone">
</FORM>
```

Once delivered to "handler.html," the request object will have three properties (name, company, and zone), which correspond to the values the user entered into the form elements of the same name. You could then use those values directly in "handler.html," as shown in this example:

```
document.write("Dear " + request.name);
document.write("Thanks for ordering...");
```

Or you could use the values for decision making:

```
document.write("Delivery to " + request.company + " will cost ");
if (request.zone == "A")
    document.write("$500");
if (request.zone == "B")
    document.write("$200");
```

The request object is automatically created by SSJS each time a new page is loaded. The request object doesn't exist in CSJS, so we must create it ourselves each time. The routine shown in Listing 1 will decode the form data stored in the URL and assign it to corresponding properties of the request object, just like SSJS. This routine must be called before you can access the form data directly. A handy way to do this is through the document body's onLoad handler.

Listing 1. The createRequestObject routine

```
// This routine returns an object similar to SSJS's
// request object. Used to process forms on the client, in a method
// similar to SSJS.
//
// Usage: request=createRequestObject();
//       document.writeln("The Name was " + request.name);
//
function createRequestObject()
{
    var request = new Object(); // Creates a new request object.
    var nameVal = ""; // Holds array for a single name-value pair.
    var inString = location.search; // Strips query string from URL.
    var separator = ","; // Character used to separate multiple values.

    // If URL contains a query string, grabs it.
    if (inString.charAt(0) == "?")
    {
        // Removes "?" character from query string.
        inString = inString.substring(1, inString.length);
        // Separates query string into name-value pairs.
        keypairs = inString.split("&");
        // Loops through name-value pairs.
```

```

    for (var i=0; i < keypairs.length; i++)
    {
        // Splits name-value into array (nameVal[0]=name, nameVal[1]=value).
        nameVal = keypairs[i].split("=");
        // Replaces "+" characters with spaces and then unescapes name-value
pair.
        for (a in nameVal)
        {
            &n bsp; nameVal[a] = nameVal[a].replace(/+/g, " ")
            &n bsp; nameVal[a] = unescape(nameVal[a]);
        }
        // Checks to see if name already exists in request object
        // (since select lists may contain multiple values).
        if (request[nameVal[0]])
        {
            &n bsp; request[nameVal[0]] += separator + nameVal[1];
        }
        else
        {
            &n bsp; request[nameVal[0]] = nameVal[1];
        }
    }
}
return request;
}

```

EXAMPLE: A SALES QUOTE GENERATOR

A common application for salespeople is to generate personalized sales quotes based on customer information provided via an HTML form such as the one shown in Figure 1. The information is used to trigger inclusion of customized chunks of text such as discount schedules or recommendations for companion products. For example, given the customer's name and order information, such an application might generate a letter like the one shown in Figure 2.

Figure 1. A sample HTML form for customer information

Acme Computers

Contact Information:

Mr. Last Name:

Products Ordered:

Model 1000 Computers:

Model 2000 Computers:

Model 3000 Computers:

Computer Mousepads:

Monitors:

Figure 2. A sample sales quote letter

Dear Mrs. Smith,

Thanks so much for your order with Acme Computer Products. Your order information is shown below:

<u>Part Number</u>	<u>Qty</u>	<u>List Price</u>	<u>Description</u>	<u>Subtotal</u>
cp2000	2	\$2500 ea.	AcmePC 200MMX	\$5000
cp3000	2	\$3500 ea.	AcmePC 300MMX	\$7000
TOTAL				\$12000

I noticed that you did not include any monitors in your order. If you would like to order monitors, please give me a call and I'll be happy to submit a new purchase order.

Your order should arrive in a few weeks. Thanks again for your business.

Automating this type of process is a boon to salespeople, because it gives them an easy way to generate a professional-looking response to the customer that also includes additional sales or other information keyed off the customer's order. While this application could be generated on the server, it would be more useful if it could be run offline from a notebook, allowing salespeople to quickly and easily create quotations wherever they are. Listing 2 shows such an application.

Listing 2. Our sample application to generate a sales quote

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
-
function createRequestObject()
{
    var request    = new Object();           // Creates a new
request object.
    var nameVal    = "";                    // Holds array
for a single name-value pair.
    var inString  = location.search;        // Strips query
```

```

// Removes "?" character from query string.
string from URL.
    var separator = ","; // Character
used to separate multiple values.

// If URL contains a query string, grabs it.
if (inString.charAt(0) == "?")
{
    // Removes "?" character from query string.
    inString = inString.substring(1, inString.length);
    // Separates query string into name-value pairs.
    keypairs = inString.split("&");
    // Loops through name-value pairs.
    for (var i=0; i < keypairs.length; i++)
    {
        // Splits name-value into array
        (nameVal[0]=name, nameVal[1]=value).
        nameVal = keypairs[i].split("=");
        // Replaces "+" characters with spaces and then
unescape name-value pair.
        for (a in nameVal)
        {
            nameVal[a] = nameVal[a].replace(/+/g, " ");
            nameVal[a] = unescape(nameVal[a]);
        }
        // Checks to see if name already exists in
request object
        // (since select lists may contain multiple
values).
        if (request[nameVal[0]])
        {
            request[nameVal[0]] += separator +
nameVal[1];
        }
        else
        {
            request[nameVal[0]] = nameVal[1];
        }
    }
}
return request;
}

// Creates a client-side table describing our inventory.
The beauty of
// this routine is that it could easily be populated
(through SSJS, for example)
// from a living database to keep the client-side version
up to date. This
// function could be kept separate from the main program
and imported by
// using something like <SCRIPT SRC="inventory.js">.
function createClientDB()
{

```

```

    {
        // Part number, description, price.
        addInventoryItem("cp1000", "AcmePC 166MHZ", 1500);
        addInventoryItem("cp2000", "AcmePC 200MHZ", 2500);
        addInventoryItem("cp3000", "AcmePC 300MHZ", 3500);
        addInventoryItem("m755", "Mouse Pad", 3500);
        addInventoryItem("crt9332", "Monitor", 3500);
    }

// Adds a row to our offline database table.
function addInventoryItem(partNum, description, price)
{
    // Creates the new inventory array that will serve as
our offline
    // version of a database table if it doesn't yet exist.
    if (typeof products == "undefined")
        products = new Array();
    // Creates a new inventory object.
    var o=new Object();
    o.partNum=partNum;
    o.description=description;
    o.price=price;
    // Adds the new object to the products table.
    products[products.length]=o;
}

// Creates a table displaying the customer's order
information.
function printOrder()
{
    var grandTotal=0; // The total value of the order.
    var subTotal=0; // The price x quantity for an item.
    var d=document; // Shortcut to save some typing.
    // Prints our column headings.
    d.writeln("<CENTER><TABLE BORDER CELLPADDING=5>");
    d.writeln("<TR ALIGN=CENTER VALIGN=CENTER>");
    d.writeln("<TD><B><TT><U>Part Number</U></TT></B></TD>");
    d.writeln("<TD><B><TT><U>Qty</U></TT></B></TD>");
    d.writeln("<TD><B><TT><U>List Price</U></TT></B></TD>");
    d.writeln("<TD><B><TT><U>Description</U></TT></B></T
D>");
    d.writeln("<TD><B><TT><U>Subtotal</U></TT></B></TD>");
    d.writeln("</TR>");

    // For each item in the inventory "database," checks for
a nonzero order amount.
    for (i=0; i < products.length; i++)
    {
        // Steps through the database item by item.
        item=products[i];
        // Makes sure that the request property exists to
avoid a JS error
        // caused by referring to nonexistent object

```

```

properties.
    if (eval("typeof request."+item.partNum) !=
"undefined")
        {
            // Sets the quantity property of our item to the
value passed
            // in from the form data. We protect ourselves
from problems
            // by running everything through parseInt.

item.quantity=eval("parseInt(request."+item.partNum+"");
    if (item.quantity > 0)
        {
            // Displays a line item for this row.
            subtotal=(item.price*item.quantity);
            d.writeln("<TR
ALIGN=CENTER><TD><TT>"+item.partNum+"</TT></TD>");

d.writeln("<TD><TT>"+item.quantity+"</TT></TD>");
            d.writeln("<TD><TT>$"+item.price+
ea.</TT></TD>");

d.writeln("<TD><TT>"+item.description+"</TT></TD>");
            d.writeln("<TD
ALIGN=LEFT><TT>$"+subtotal+"</TT></TD></TR>");
            // Tracks the total for later.
            grandTotal += subtotal;
        }
    }
    // Closes up the table and prints the grand total of all
line items.
    d.writeln("<TR
ALIGN=LEFT><TD><B><TT>TOTAL</TT></B></TD><TD> </TD>");
    d.writeln("<TD> </TD><TD> </TD><TD><TT>" );

d.writeln("$"+grandTotal+"</TT></TD></TR></TABLE></CENTER>");
}

// Builds our request object.
request=createRequestObject();
</SCRIPT>
</HEAD>

<BODY BGCOLOR="#FFFFFF">
<FONT FACE="Arial,Helvetica">
<SCRIPT LANGUAGE="JavaScript">
// Prints the greeting -- for example, "Dear Mr. Green
Jeans."
document.writeln("Dear "+request.salutation+
"+request.lname+",")
</SCRIPT>
<P>

```

Thanks so much for your order with Acme Computer Products.
Your order information is shown below:<P>

```
<SCRIPT LANGUAGE="JavaScript">
// Builds an easy-to-access table of inventory information.
createClientDB();
// Displays a line-item detail of the customer's order.
printOrder();
</SCRIPT>
```

<P>

```
<SCRIPT LANGUAGE="JavaScript">
// Checks for special situations. In this case, we want to
try to sell the
// customer monitors if none were ordered.
if (parseInt(request.crt9332) == 0)
{
    document.writeln("I noticed that you did not include any
monitors in your order. ");
    document.writeln("If you would like to order monitors
please give me a call ");
    document.writeln("and I'll be happy to submit a new
purchase order.<P>");
}
</SCRIPT>
```

Your order should arrive in a few weeks. Thanks again for
your business.

```
</FONT>
</BODY>
</HTML>
```

GOING FURTHER

We could expand this application by integrating it with a production database. An SSJS application could be used to generate new client-side data nightly, keeping the application up to date. It could even be coupled with a server-side version that speaks directly with the live database - giving you online and offline versions of your tool.

Another application that works well with the client-side technique is an expert system. For example, you ask the user a series of multiple-choice questions, then based on the input, return suggestions appropriate for the situation. To do this, your form would send the answers to the action receiving page, where they would be compared to an array of answers matched to solutions.

FURTHER RESOURCES

- Robert Husted, ["Form-Data Handling in Server-Side JavaScript." View Source](#)

- Z. Peter Lazar and Peter Holfelder, "[CGI vs. Server-Side JavaScript for Database Applications](#)," [View Source](#)

View Source wants your feedback!

[Write to us](#) and let us know
what you think of this article.

Many thanks to Robert Husted for his assistance in preparing this article.

[Duane K. Fields](#) is an Internet systems engineer with Tivoli Systems, an IBM Company, in Austin, Texas. Duane has been developing network applications in one form or another since the early days of the Internet. He can be reached at duane@deepmagic.com.

(5.98)

Related Reading

- [Manuals](#)
- [TechNotes](#)
- [Sample Code](#)
- [View Source Articles](#)
- [Newsgroup](#)

Any sample code included above is provided for your use on an "AS IS" basis, under the [Netscape License Agreement - Terms of Use](#)