ADOBE® CREATIVE CLOUD®

JAVASCRIPT TOOLS GUIDE

Adobe

6 External Communication Tools

ExtendScript offers tools for communicating with other computers or the Internet using standard protocols.

- The Socket object supports low-level TCP connections. It is available in the following applications:
  - Adobe Bridge CS5
  - Adobe InDesign CS5
  - Adobe InCopy® CS5
  - Adobe After Effects® CS5
  - Adobe Photoshop CS5

Socket object

TCP connections are the basic transport layer of the Internet. Every time your Web browser connects to a server and requests a new page, it opens a TCP connection to handle the request as well as the server's reply. The JavaScript `Socket` object lets you connect to any server on the Internet and to exchange data with this server.

The `Socket` object provides basic functionality to connect to a remote computer over a TCP/IP network or the Internet. It provides calls like `open()` and `close()` to establish or to terminate a connection, and `read()` or `write()` to transfer data. The `listen()` method establishes a simple Internet server; the server uses the method `poll()` to check for incoming connections.

Many of these connections are based on simple data exchange of ASCII data, while other protocols, like the FTP protocol, are more complex and involve binary data. One of the simplest protocols is the HTTP protocol. The following sample TCP/IP client connects to a WWW server (which listens on port 80); it then sends a very simple HTTP GET request to obtain the home page of the WWW server, and then it reads the reply, which is the home page together with a HTTP response header.

```
reply = "";
conn = new Socket;
// access Adobe's home page
if (conn.open ("www.adobe.com:80")) {
    // send a HTTP GET request
    conn.write ("GET /index.html HTTP/1.0\n\n");
    // and read the server's reply
    reply = conn.read(999999);
    conn.close();
}
```

After executing this code, the variable `reply` contains the contents of the Adobe home page together with an HTTP response header.

Establishing an Internet server is a bit more complicated. A typical server program sits and waits for incoming connections, which it then processes. Usually, you would not want your application to run in an endless loop, waiting for any incoming connection request. Therefore, you can ask a `Socket` object for an incoming connection by calling the `poll()` method of a `Socket` object. This call would just check the

incoming connections and then return immediately. If there is a connection request, the call to `poll()` would return another `Socket` object containing the brand new connection. Use this connection object to talk to the calling client; when finished, close the connection and discard the connection object.

Before a `Socket` object is able to check for an incoming connection, it must be told to listen on a specific port, like port 80 for HTTP requests. Do this by calling the `listen()` method instead of the `open()` method.

The following example is a very simple Web server. It listens on port 80, waiting until it detects an incoming request. The HTTP header is discarded, and a dummy HTML page is transmitted to the caller.

```
conn = new Socket;
// listen on port 80
if (conn.listen (80)) {
    // wait forever for a connection
    var incoming;
    do incoming = conn.poll();
    while (incoming == null);
    // discard the request
    conn.read();
    // Reply with a HTTP header
    incoming.writeln ("HTTP/1.0 200 OK");
    incoming.writeln ("Content-Type: text/html");
    incoming.writeln();
    // Transmit a dummy homepage
    incoming.writeln ("<html><body><h1>Homepage</h1></body></html>");
    // done!
    incoming.close();
    delete incoming;
}
```

Often, the remote endpoint terminates the connection after transmitting data. Therefore, there is a connected property that contains true as long as the connection still exists. If the connected property returns false, the connection is closed automatically.

On errors, the `error` property of the `Socket` object contains a short message describing the type of the error.

The `Socket` object lets you easily implement software that talks to each other via the Internet. You could, for example, let two Adobe applications exchange documents and data simply by writing and executing JavaScript programs.

Chat server sample

The following sample code implements a very simple chat server. A chat client may connect to the chat server, who is listening on port number 1234. The server responds with a welcome message and waits for one line of input from the client. The client types some text and transmits it to the server who displays the text and lets the user at the server computer type a line of text, which the client computer again displays. This goes back and forth until either the server or the client computer types the word “bye”.

```
// A simple Chat server on port 1234
function chatServer() {
  var tcp = new Socket;
  // listen on port 1234
  writeln ("Chat server listening on port 1234");
  if (tcp.listen (1234)) {
    for (;;) {
```

```
      // poll for a new connection
      var connection = tcp.poll();
      if (connection != null) {
        writeln ("Connection from " + connection.host);
        // we have a new connection, so welcome and chat
        // until client terminates the session
        connection.writeln ("Welcome to a little chat!");
        chat (connection);
        connection.writeln ( "*** Goodbye ***");
        connection.close();
        delete connection;
        writeln ("Connection closed");
      }
    }
  }
}
function chatClient() {
    var connection = new Socket;
    // connect to sample server
    if (connection.open ("remote-pc.corp.adobe.com:1234")) {
        // then chat with server
        chat (connection);
        connection.close();
        delete connection;
    }
}
function chat (c) {
    // select a long timeout
    c.timeout=1000;
    while (true) {
        // get one line and echo it
        writeln (c.read());
        // stop if the connection is broken
        if (!c.connected)
            break;
        // read a line of text
        write ("chat: ");
        var text = readln();
        if (text == "bye")
            // stop conversation if the user entered "bye"
            break;
        else
            // otherwise transmit to server
            c.writeln (text);
    }
}
```

Socket object reference

This section provides details of the object's properties and methods.

Socket object constructor

```
[new] Socket ();
```

Creates and returns a new `Socket` object.

Socket object properties

`connected`	Boolean	When true, the connection is active. Read only.
`encoding`	String	Sets or retrieves the name of the encoding used to transmit data. Typical values are "ASCII," "BINARY," or "UTF-8."
`eof`	Boolean	When true, the receive buffer is empty. Read only.
`error`	String	A message describing the most recent error. Setting this value clears any error message.
`host`	String	The name of the remote computer when a connection is established. If the connection is shut down or does not exist, the property contains the empty string. Read only.
`timeout`	Number	The timeout in seconds to be applied to read or write operations. Default is 10.

Socket object functions

`close()`

```
socketObj.close ();
```

Terminates the open connection. Deleting the object also closes the connection, but not until JavaScript garbage-collects the object. The connection might stay open longer than you wish if you do not close it explicitly.

Returns true if the connection was closed, false on I/O errors.

listen()
socketObj.listen (*port* [, *encoding*]);

port Number. The TCP/IP port number to listen on. Valid port numbers are 1 to 65535. Typical values are 80 for a Web server, 23 for a Telnet server and so on.

encoding Optional. String. The encoding to be used for the connection. Typical values are "ASCII," "binary," or "UTF-8." Default is "ASCII."

Instructs the object to start listening for an incoming connection.

The call to `open()` and the call to `listen()` are mutually exclusive. Call one function or the other, not both.

Returns true on success.

open()
socketObj.open (*host* [, *encoding*]);

host String. The name or IP address of the remote computer, followed by a colon and the port number to connect to. The port number is required. Valid computer names are, for example, "www.adobe.com:80" or "192.150.14.12:80".

encoding Optional. String. The encoding to be used for the connection. Typical values are "ASCII," "binary," or "UTF-8." Default is "ASCII."

Opens the connection for subsequent read/write operations.

The call to `open()` and the call to `listen()` are mutually exclusive. Call one function or the other, not both.

Returns true on success.

poll()
socketObj.poll ();

Checks a listening object for a new incoming connection. If a connection request was detected, the method returns a new `Socket` object that wraps the new connection. Use this connection object to communicate with the remote computer. After use, close the connection and delete the JavaScript object. If no new connection request was detected, the method returns `null`.

Returns a `Socket` object or `null`.

read()
socketObj.read ([*count*]);

count Optional. Number. The number of characters to read; default is 0. If negative, the call is equivalent to `readln()`

Reads up to the specified number of characters from the connection, waiting if necessary. Ignores CR characters unless encoding is set to `BINARY`.

Returns a string that contains up to the number of characters that were supposed to be read, or the number of characters read before the connection closed or timed out.

readln()
socketObj.readln ();

Reads one line of text up to the next line feed. Line feeds are recognized as LF or CRLF pairs. CR characters are ignored.

Returns a string.

write()
socketObj.write (*text*[, *text*...]);

text String. Any number of string values. All arguments are concatenated to form the string to be written.

Concatenates all arguments into a single string and writes that string to the connection. CRLF sequences are converted to LFs unless encoding is set to BINARY.

Returns true on success.

writeln()
socketObj.write (*text*[, *text*...]);

text String. Any number of string values. All arguments are concatenated to form the string to be written.

Concatenates all arguments into a single string, appends a Line Feed character, and writes that string to the connection.

Returns true on success.