



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

## Applet-to-Servlet Communication for Enterprise Applications

By [Duane K. Fields](#)

---

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

---

Java applets and servlets can be used together in the design of today's multitiered web applications. Applets provide a convenient mechanism for building powerful, dynamic interfaces to applications, while servlets give us a highly efficient means to handle requests on a web or application server. Sun's Application Programming Model, which describes the best practices for developing enterprise Java applications on the Java 2 platform, recommends using applets, HTML, and JavaServer Pages on the front end and Java servlets backed up by Enterprise JavaBeans or other components on the back end.

The key to this architecture is communication between applets on the client and servlets on the server. But due to limitations imposed on applets by the browser security model, getting data and messages into or out of an applet isn't as straightforward as it might seem. In this article I explain the restrictions facing developers in the applet-servlet architecture and explore several different communication tactics that enable data transfer between them. While it would be helpful for you to already be familiar with applet and servlet design and programming, I'll introduce both briefly in case you aren't.

### APPLETS AND SERVLETS FOR THE UNINITIATED

#### Applets

Java applets are essentially Java programs that run within a web page. They're Java classes that extend the

`java.applet.Applet` class and are embedded by reference within an HTML page, much like an image. Combined with HTML, they can make an interface more dynamic and powerful than with HTML alone. While some applets do nothing more than scroll text or play animations, they can be used in an enterprise application to view or manipulate data coming from some source on the server. For example, an applet may be used to browse and modify records in a database or control runtime aspects of some other application running on the server.

Besides the class file defining the Java applet itself, applets can use a collection of utility classes, either by themselves or archived into a JAR file. The applets and their class files are distributed through standard HTTP requests and therefore can be sent across firewalls with the rest of the web page data. Applet code is refreshed automatically each time the user revisits the hosting web site, eliminating the concern of keeping the full application up to date on each client desktop to which it's been distributed.

Thanks to Java's operating system agnosticism, applets can run in any browser with a Java virtual machine (JVM). Sun's [Java Plug-in software](#) even lets you build pages that can take advantage of the latest JVM, instead of being restricted to whatever version of the JVM your user's browser happens to have implemented.

Since applets are extensions of the Java platform, you can reuse existing Java components when you build at least a portion of your web application interface with applets. As we'll see in a later example, you can use complex Java objects developed originally for server-side applications as components of your applets. In fact, you can write Java code that can operate as either an applet or an application.

Applets have all of the functionality of a traditional Java application, including the ability to use

advanced [JFC/Swing components](#) from Sun. Applets have the full graphical and user interface capability of applications (though any secondary windows you create will be marked with "Warning, Java Applet Window" notifications). But despite their similarities, there are some key differences between applications and applets. The one that concerns us here has to do with the security constraints imposed on applets.

### Security Constraints on Applets

Applet code is served from a host web server and executed in the client's browser on the end user's machine. To prevent the proliferation of evil viral applets that could wreak havoc on unsuspecting surfers, applets are bound by security constraints that allow them to communicate only with their host server and prevent them from interacting with the end user's machine. They can't read or write from its file systems, execute programs, or examine certain sensitive environmental properties. (There actually *is* a way around this - the developer can sign the applet code with a digital signature, which will then trigger the browser to ask the user for specific privileges it otherwise wouldn't be granted - but signed applets are beyond the scope of this article.) Furthermore, applets can't create or accept foreign socket connections. *Foreign* here means a connection with anything other than the local server, which is the machine actually providing the applet's class files (as distinct from the host serving the HTML that references the applet).

Because of these restrictions, we must employ special strategies to communicate information to or from an applet. The only avenue of communication we have is the network connection between the local server on which the applet resides and the host serving the HTML that references the applet. Still, this gives us a way to build real-time interactive interfaces that can use the web as their platform.

### Servlets

Java servlets are server-side components that are analogous in many ways to CGI programs. They handle web requests, returning data or HTML programmatically rather than from a static file. They can access databases, perform calculations, and communicate with other components such as Enterprise JavaBeans. Unlike CGI programs, however, servlets are persistent - they're instantiated once and continually handle requests (usually many simultaneous requests) for the life of the web server. They're therefore operating at a much higher level of efficiency than CGI programs.

Servlets run within a servlet engine, usually on a web or application server. Both Netscape Enterprise Server 4.0 and Netscape Application Server support recent releases of the Java servlet specification. Unlike applets, servlets aren't burdened with security restrictions. Since they execute entirely on the server, they can run with all the capabilities that the operating system allows.

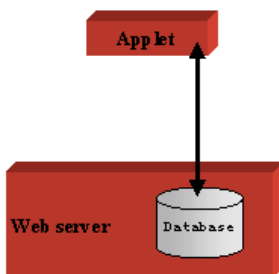
Servlets can be used to create an easily accessible interface between clients, such as applets and web browsers, and the core enterprise applications behind them. To the client, requests that go to servlets look no different from any other web request. The client contacts a URL and is given results back. As we'll see, the results can be a lot more than just HTML. Virtually any kind of data can be sent and received via the HTTP protocol.

### ARCHITECTURAL OPTIONS

An enterprise application that makes use of applets and servlets can conceivably be designed in more than one way. I'll outline three different architectural options here and describe their drawbacks and advantages.

The first option actually uses applets but no servlets, for despite the limitations imposed on applets by their security model, applets can use protocols such as JDBC and RMI to communicate directly with back-end information systems like databases, LDAP directories, or Enterprise JavaBeans components. This architectural model is diagrammed in Figure 1. Although it seems simple, this model poses a number of problems and isn't generally a good idea. First of all, this scheme requires you to embed all of your access information in the applet code directly. Database account names, passwords, server identifiers - everything must be coded into the applet, where the end user might be able to glean the information from the class files. Additionally, the database or whatever system you're accessing has to be on the same system as the web server that hosted the applet. This means your server has to do double duty as both a database and a web server. Typically, your back-end resources would be restricted by a firewall, but this isn't possible in this situation since the applet (acting from the client machine) must have direct access to the machine. Finally, this scheme makes pooling and clustering of your web servers difficult, if not impossible.

Figure 1. A two-tier application architecture

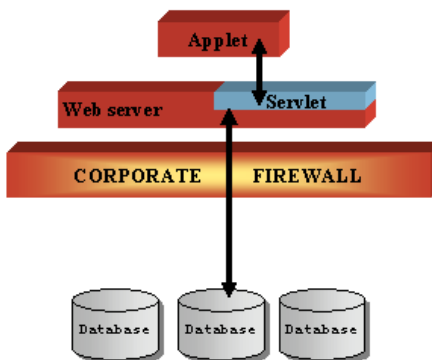


---

A better approach is to encapsulate the business of communicating with back-end resources into servlets, leaving applets to handle the front-end work only. In this architecture, sketched in Figure 2, servlets help overcome the security restrictions inherent in applets and control the applet's access to enterprise information systems and business logic. When a request comes in to a servlet, the servlet can look up information in a back-end database, perform calculations, or do whatever's necessary to get information on behalf of the applet or act on information from the applet. A big advantage here is that applet/servlet pairs can be deployed across a large pool of front-end web servers, all communicating with a single shared database on the back end. In addition, designing around servlets helps modularize the design, abstract the business logic behind the application, and plan for scalability.

---

Figure 2. A three-tier application architecture



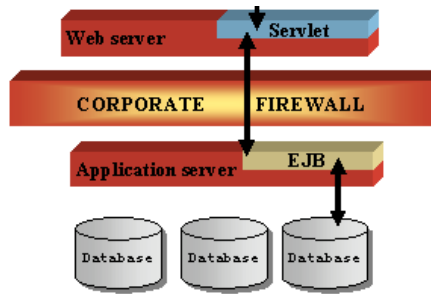
---

If you're building your application around Enterprise JavaBeans, servlets act as the go-between. The EJB components can help further abstract the business logic, moving it outside of the servlet. In this case, an applet contacts its servlet and the servlet interfaces with the EJB component, as shown in Figure 3. Building an application with a hierarchy of EJB components, servlets, and applet/HTML front ends gives you the maximum degree of flexibility and performance, though at the cost of greater complexity and expense.

---

Figure 3. An n-tier application architecture






---

## COMMUNICATION TACTICS

If you use an architecture with applets on the front end and servlets on the back end, you'll need to implement applet-to-servlet communication. Due to the limitations imposed on applets by the browser security model, we have few options when it comes to getting data and messages into or out of an applet. As mentioned above, we can't read from the client's file system or running processes, and since applets aren't running on the server, we don't have any way to access those file systems either. This pretty much leaves the network, which we can only use to create network connections to services running on our local server. Furthermore, let's not forget that for applications deployed across the public Internet, firewalls will probably restrict us to talking to servlets or other web-server modules via HTTP. In fact, since the applet itself was delivered via HTTP over the network, we know for sure that that communication option is available to us.

Given that the network connection between the applet on the client and the servlet on the server is the only communication path we can use, there's more than one way to exchange information. As you know, streams of text can be delivered back from the server via HTTP. What you may not know is that Java objects can also be sent this way. We'll look at the use of HTTP text streams and HTTP object streams in detail. In addition, we'll look briefly at ways to communicate over raw sockets, which can be a useful approach in controlled conditions, especially for applications requiring bidirectional, constant communication.

### HTTP Text Streams

▀The simplest way for an applet to exchange information with a servlet is through an HTTP text stream. Java's

`URL` and `URLConnection` classes make it easy to read data from a URL without having to worry about sockets and other normally complex issues of network programming. All we need is a server-side component that can deliver information via a URL. This is why servlets work so well here.

As an example, let's say we want to monitor the amount of memory available in the server's JVM and have an applet represent it with a simple meter. We first need to develop a servlet that when accessed via its URL will return the information we need to draw the meter. Such a servlet is shown in Listing 1.

---

#### Listing 1

```
import javax.servlet.*;
import javax.servlet.http.*;
public class ShowMemServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        res.setContentType("text/plain");
        PrintWriter out = res.getWriter();
        Runtime rt = Runtime.getRuntime();
        out.println(rt.freeMemory());
        out.println(rt.totalMemory());
    }
}
```

---

This extremely simple servlet will respond to

GET requests (through the browser directly, or, as we'll soon see, via our applet) with two lines of text. The first line of text will be the number of free bytes available to the server's JVM, and the second will be the total number of bytes the JVM has reserved.

To construct our meter applet, all we have to do is make a URL connection to the servlet, wrap its

`InputStream` in a `DataInputStream` (which reads strings and lines of text rather than raw bytes), read in the two parameters, convert them back into numbers, and update our meter. We can then have our applet implement the `Runnable` interface and run in its own thread. A few times a second, we can run a method to update the meter. This `refresh()` method, shown in Listing 2, fetches the update from the servlet.

---

### Listing 2

```
private void refresh() throws MalformedURLException, IOException {
    URL url = new URL(getCodeBase(), "/servlet/ShowMemServlet");
    URLConnection con = url.openConnection();
    con.setUseCaches(false);
    InputStream in = con.getInputStream();
    DataInputStream textStream;
    textStream = new DataInputStream(in);
    String line1 = textStream.readLine();
    String line2 = textStream.readLine();
    double freeMem = Double.parseDouble(line1);
    double totalMem = Double.parseDouble(line2);
    int usedMem = totalMem - freeMem;
    int percentUsed = (int) 100 * (usedMem / totalMem);
    meter.setLength(percentUsed);
}
```

---

As you can see, HTTP text streams are relatively straightforward and easy to use. The applet establishes a connection to a servlet back on its originating server, reads its two lines of information, and interprets them appropriately.

Using simple text streams to exchange data has one major weakness, however - the need for the applet to not only know the format of the data but also perform parsing and conversion of the data into a useful form. In our example it wasn't too hard to convert the strings to numbers, but the conversion chore could quickly get out of hand if we had to deal with more complex data and objects. There's an easier way of exchanging complex data, which we'll look at next.

### HTTP Object Streams

You may not realize that an HTTP connection can be used to transfer binary data as well as textual data, but you use it to do just that every time you see an image on the web or download a

`.zip` file. We can combine this capability with something called object serialization to pass complete Java objects from a servlet to an applet. Complex data can be passed very easily this way with no need for parsing and interpretation.

Object serialization allows us to "flatten" objects into streams of binary data that can go anywhere an

`OutputStream` can go - to disk, to the screen, or, as in our example, across an HTTP connection to an applet. Object serialization is a feature of JVM 1.1 and higher, so older browsers can't take advantage of it. But the [Java Plug-in](#) from Sun can upgrade almost any browser to support the latest releases of Java and thus to be capable of object serialization.

We use an HTTP object stream in very much the same way as we use an HTTP text stream. We initiate a URL connection back to a servlet on our local web server, then read in the resulting data. Rather than wrapping the

`InputStream` in a `DataInputStream` as before, however, we wrap it in an `ObjectInputStream` - a special

type of stream that reads objects. We then read in each object, casting it to the appropriate type.

Listing 3 shows a portion of code that exchanges an object we've designed to hold information about a record album (with artist, title, price, and other information contained in it) from a servlet. You'll note that aside from the fact that we're using

`ObjectOutputStream` rather than `PrintWriter` as in Listing 1, the basic concept is the same as in our previous example.

---

### Listing 3

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException {
    OutputStream out;
    ObjectOutputStream objStream;
    out = res.getOutputStream();
    objStream = new ObjectOutputStream(out);
    Album album = fetchNextAlbum();
    out.writeObject(album);
}
```

---

This servlet will write the serialized version of the album object it receives back from its

`fetchNextAlbum()` method as a stream of data to its output stream. If you were to visit this servlet with a web browser, you would likely see a bunch of garbage. This is because we're no longer dealing with plain text but rather with a serialized object represented by binary data and not suitable for human consumption. Note that an object must implement the `Serializable` interface in order to be serialized. If it doesn't, your compiler will complain. In addition, any object directly referenced in the object you're serializing must also implement `Serializable`.

On the applet side, we'll use the method

`getNextAlbum()`, shown in Listing 4, to fetch an album object from the server each time the user clicks the applet's Next button. The applet can then interrogate this object and display data from it.

---

### Listing 4

```
private Album getNextAlbum() throws MalformedURLException, IOException {
    URL url = new URL(getCodeBase(), "/servlet/AlbumServlet");
    URLConnection con = url.openConnection();
    con.setUseCaches(false);
    InputStream in = con.getInputStream();
    ObjectInputStream objStream;
    objStream = new ObjectInputStream(in);
    album = (Album)objStream.readObject();
    return album;
}
```

---

As you can see, object streams give us a very convenient way to exchange complex collections of information between the applet and the server. Also note that rather than working with pure data, we're using a more object-oriented approach - even reusing our objects between the client and the server. This lets us build intelligence into the objects and avoid duplicating that logic in both client and server portions of our application.

### Raw Socket Connections

Another, less common strategy for applet-to-servlet communication is to create a raw socket connection from the applet back to the servlet. It's up to the developer in this case to design and implement an appropriate protocol to deal with the specifics of both the applet and the servlet sides of the socket connection.

◀ ▶ ↻ 🔍

One big benefit of using raw sockets is that the connection is persistent and bidirectional. In the HTTP-based connections we've been exploring are transient exchanges of information. If you need to continually update the applet with new information, as in our meter example earlier, you must continually make new HTTP connections back to the server. With a raw socket approach, you can establish a connection with the server and continually receive updates as they occur.

Of course, you'll probably want to implement a multithreaded system so that several applets can be in contact with the server simultaneously. And note that the use of raw sockets isn't usually appropriate, as most Internet firewalls aren't going to allow arbitrary communication across uncommonly used port numbers. However, in an intranet environment it might be a useful technique.

## APPLETS AND SERVLETS WORKING TOGETHER

In this article we've explored the useful strategy of building applications with applets on the front end and servlets on the back end. Applets have only one communication path available, but we have a choice of schemes to exchange information between an applet and a servlet. For simple data and messaging, we can just pass it through as plain text. To work with objects directly or encapsulate complex data structures, we can employ object serialization. For specialized internal applications requiring real-time, bidirectional communications, raw socket connections can also be used.

Sun's Application Programming Model recommends strategies like the ones we've discussed here. Separating business logic from the interface leads to flexible, scalable applications that are easy to design and maintain. By not requiring your applets to access and work directly with your enterprise data, you greatly reduce their complexity and increase the security of your systems. Applets and servlets can work together to help build better applications.

---

## FURTHER RESOURCES

- [Java Plug-in software](#)
- [JFC/Swing components](#)
- [Applet resources](#)
- [Java Servlet API](#)
- Duane K. Fields, [Java Servlets for JavaScripters: Expanding Your Server-Side Programming Repertoire](#), *View Source*
- Duane K. Fields, [Looking Into Enterprise JavaBeans](#), *View Source*
- JJ Kuslich, [Introduction to JavaServer Pages: Server-Side Scripting the Java Way](#), *View Source*

---

*View Source* wants your feedback!  
[Write to us](#) and let us know  
what you think of this article.

---

[Duane K. Fields](#) is a senior Internet systems engineer with Tivoli Systems, an IBM company, in Austin, Texas. A Sun-certified Java programmer, Duane has been developing network applications in one form or another since the early days of the Internet.

(8.99)

---

## Related Reading:

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

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

---

[iPlanet International](#) | [Year 2000](#) | [Site Map](#) | [Feedback](#)  
[Products](#) | [Solutions](#) | [Support](#) | [Services](#) | [Download](#) | [About Us](#) | [Developer](#)  
© 2000 Sun-Netscape Alliance. [All Rights Reserved](#) [Privacy Policy](#)