The Client Tier

FROM a developer’s point of view, a J2EE application can support many types of  
clients. J2EE clients can run on laptops, desktops, palmtops, and cell phones. They  
can connect from within an enterprise’s intranet or across the World Wide Web,  
through a wired network or a wireless network or a combination of both. They can  
range from something thin, browser-based and largely server-dependent to something rich, programmable, and largely self-sufficient.  
From a user’s point of view, the client *is* the application. It must be useful,  
usable, and responsive. Because the user places high expectations on the client,  
you must choose your client strategy carefully, making sure to consider both technical forces (such as the network) and non-technical forces (such as the nature of  
the application

**Client Considerations**Every application has requirements and expectations that its clients must meet, constrained by the environment in which the client needs to operate.  
Your users and their usage patterns largely determine what type of client or  
interface you need to provide. For example, desktop Web browser clients are popular for e-mail and e-shopping because they provide a familiar interface. For  
another example, wireless handheld clients are useful for sales force automation  
because they provide a convenient way to access enterprise resources from the  
field in real time. Once you have decided what type of interface you need, you  
should design your client configuration with network, security, and platform considerations in mind.

**Network Considerations**J2EE clients may connect to the enterprise over a wide array of networks. The  
quality of service on these networks can vary tremendously, from excellent on a  
company intranet, to modest over a dialup Internet connection, to poor on a wireless  
network. The connectivity can also vary; intranet clients are always connected,  
while mobile clients experience intermittent connectivity (and are usually online for  
short periods of time anyway).  
Regardless of the quality of service available, you should always keep in mind  
that the client *depends* on the network, and the network is imperfect. Although the  
client *appears* to be a stand-alone entity, it cannot be programmed as such because  
it is part of a distributed application. Three aspects of the network deserve particular mention:  
• Latency is non-zero.  
• Bandwidth is finite.  
• The network is not always reliable.  
A well-designed enterprise application must address these issues, starting  
with the client. The ideal client connects to the server only when it has to, transmits only as much data as it needs to, and works reasonably well when it cannot  
reach the server

**Security Considerations**Different networks have different security requirements, which constrain how  
clients connect to an enterprise. For example, when clients connect over the Internet,  
they usually communicate with servers through a firewall. The presence of a firewall  
that is not under your control limits the choices of protocols the client can use. Most  
firewalls are configured to allow Hypertext Transfer Protocol (HTTP) to pass across, but not Internet Inter-Orb Protocol (IIOP). This aspect of firewalls makes Webbased services, which use HTTP, particularly attractive compared to RMI- or  
CORBA-based services, which use IIOP.  
Security requirements also affect user authentication. When the client and  
server are in the same security domain, as might be the case on a company intranet, authenticating a user may be as simple as having the user log in only once to  
obtain access to the entire enterprise, a scheme known as *single sign on*. When the  
client and server are in different security domains, as would be the case over the  
Internet, a more elaborate scheme is required for single sign on, such as that proposed by the Liberty Alliance, an industry collaboration spearheaded by Sun  
Microsystems.  
The authentication process itself needs to be confidential and, usually, so does  
the client-server communication after a user has been authenticated

**Platform Considerations**Every client platform’s capabilities influence an application’s design. For example, a  
browser client cannot generate graphs depicting financial projections; it would need  
a server to render the graphs as images, which it could download from the server. A  
programmable client, on the other hand, could download financial data from a server  
and render graphs in its own interface.  
Another aspect of the platform to consider is form factor. Desktop computers  
offer a large screen, a keyboard, and a pointing device such as a mouse or trackball. With such clients, users are willing to view and manipulate large amounts of  
data. In contrast, cell phones have tiny screens and rely on button-based interactions (usually thumb-operated!). With such clients, users can’t (and don’t want to)  
view or manipulate large amounts of data.  
Applications serving multiple client platforms pose additional challenges.  
Developing a client for each platform requires not only more resources for implementation, testing, and maintenance but also specialized knowledge of each platform. It may be easier to develop one client for all platforms (using a browser- or  
a Java technology-based solution, for example), but designing a truly portable  
client requires developers to consider the lowest common denominator. Consequently, such a client implementation cannot take advantage of the various capabilities unique to each platform.

**General Design Issues and Guidelines**

**Presenting the user interface***—*Although a client presents the views to a user,  
the logic for the views may be programmed on the client or downloaded from  
a server.  
• **Validating user inputs***—*Although the EIS and EJB tier must enforce constraints on model data (since they contain the data), a client may also enforce  
data constraints by validating user inputs.  
• **Communicating with the server***—*When a user requests functionality that resides on a server, the user’s client must present that request to the server using  
a protocol they both understand.  
• **Managing conversational state***—*Applications need to track information as a  
user goes through a workflow or process (effectively conversing with the application). The client may track none, some, or all of this information, known  
as conversational state.  
How you handle these responsibilities on your client can significantly impact  
your development efforts, your application’s performance, and your users’ experience. Generally, the more responsibilities you place on the client, the more  
responsive it will be.

**Design Issues and Guidelines for Browser Clients***Browsers* are the thinnest of clients; they display data to their users and rely on  
servers for application functionality.  
From a deployment perspective, browser clients are attractive for a couple of  
reasons. First, they require minimal updating. When an application changes, erver-side code has to change, but browsers are almost always unaffected.  
Second, they are ubiquitous. Almost every computer has a Web browser and many  
mobile devices have a microbrowser.  
This section documents the issues behind designing and implementing  
browser clients

**Validating User Inputs**Consider an HTML form for completing an order, which includes fields for credit  
card information. A browser cannot single-handedly validate this information, but it  
can certainly apply some simple heuristics to determine whether the information is  
invalid. For example, it can check that the cardholder name is not null, or that the  
credit card number has the right number of digits. When the browser solves these  
obvious problems, it can pass the information to the server. The server can deal with  
more esoteric tasks, such as checking that the credit card number really belongs to  
the given cardholder or that the cardholder has enough credit.

<form name="account\_form" method="POST"  
action="default\_test.aspx"**onSubmit="return  
checkFamilyName();"**>  
<p>Family name: <input type="text" name="family\_name"></p>  
<!- ... -->  
<p><input type="submit" value="Send it!" /></p>  
</form>  
**Code Example 3.1** HTML Form Calling a JavaScriptValidation Function  
Code Example 3.2 shows how the JavaScript validation function might be  
implemented.  
<script language="JavaScript">  
<!--  
function checkFamilyName() {  
var familyName =  
window.document.account\_form.family\_name.value;  
if (familyName == "") {  
alert("You didn't enter a family name.");  
return false;  
}  
else {  
return true;  
}  
}

</script>

**Communicating with the Server**Browser clients connect to a J2EE application over the Web, and hence they use  
HTTP as the transport protocol.  
When using browser interfaces, users generally interact with an application by  
clicking hyperlinked text or images, and completing and submitting forms.  
Browser clients translate these gestures into HTTP requests for a Web server,  
since the server provides most, if not all, of an application’s functionality.  
User requests to retrieve data from the server normally map to HTTP GET  
requests. The URLs of the requests sometimes include parameters in a query  
string that qualify what data should be retrieved. For example, a URL for listing  
all dogs might be written as follows:  
http://javapetstore.sun.com/product.screen?category\_id=DOGS  
User requests to update data on the server normally map to HTTP POST  
requests. Each of these requests includes a MIME envelope of type  
application/x-www-form-urlencoded, containing parameters for the update. For  
example, a POST request to complete an order might use the URL

**Managing Conversational State**Because HTTP is a request-response protocol, individual requests are treated  
independently. Consequently, Web-based enterprise applications need a mechanism for identifying a particular client and the state of any conversation it is  
having with that client.  
The *HTTP State Management Mechanism* specification introduces the notion  
of a *session* and *session state*. A session is a short-lived sequence of service  
requests by a single user using a single client to access a server. Session state is  
the information maintained in the session across requests. For example, a shopping cart uses session state to track selections as a user chooses items from a catalog. Browsers have two mechanisms for caching session state: cookies and URL  
rewriting.  
• A *cookie* is a small chunk of data the server sends for storage on the client.  
Each time the client sends information to a server, it includes in its request the  
headers for all the cookies it has received from that server. Unfortunately,  
cookie support is inconsistent enough to be annoying: some users disable  
cookies, some firewalls and gateways filter them, and some browsers do not  
support them. Furthermore, you can store only small amounts of data in a  
cookie; to be portable across all browsers, you should use four kilobytes at  
most.  
• *URL rewriting* involves encoding session state within a URL, so that when the  
user makes a request on the URL, the session state is sent back to the server.  
This technique works almost everywhere, and can be a useful fallback when  
you cannot use cookies. Unfortunately, pages containing rewritten URLs consume much bandwidth. For each request the server receives, it must rewrite