Application hosting for pervasive computing

by S. G. Hild
C. Binding
D. Bourges-Waldegg
C. Steenkeste

This paper reviews the impact that the emerging pervasive "sea of devices" may have on the task of service provisioning and introduces Whale, an architecture that enables an application not only to vary the format of the content generated for each particular device, but also allows the author to define a device-specific view on the application's data and features, thus providing optimal application interaction for each device. Whale achieves this through the strict separation of content presentation from content generation, using JavaServer Pages™ and JavaBeans™ technologies, and by creating WhaleInvoker as an enhancement of WebSphere™, which dynamically selects and executes the appropriate combination of JavaServer Pages and JavaBeans to satisfy a data request from an end-user device. The paper also describes the first commercial deployment of the Whale architecture—Swissair's Easy Check-In service.

Pervasive computing enables a broad range of end-user devices to access data and applications on servers, much like today's PC (personal computer) access to HTML (HyperText Markup Language) sources across the Internet. However, these new form factors differ from the conventional HTML-centric PC, not only with regard to their input and output capabilities, but also, and more significantly, in the way they interact with the user. This includes alternative modalities (such as voice) as well as alternative usage patterns (such as mobility). The common feature among those devices is their browser-centric architecture: content is transmitted from the server to the client in the form of a marked-up document, and all interaction between the users at the client device is orchestrated through a user-interface that, although controlled by the server through the marked-up document, is generated and maintained by a browser program that is independent of the server and its application and resides permanently on the device. The user interface itself is not necessarily display-based, but may very well be a voice interface. Such browser devices are also frequently referred to as "thin" clients, not because the browser infrastructure on the device is necessarily thin or the device is compact in size or has limited storage or memory, but because the application is maintained exclusively on the server. Apart from simple caching, no aspect of the application is resident on the client beyond individual request-reply boundaries.

In this paper, a review is given of the impact that this pervasive "sea of devices" may have on the task of service provisioning, and the Whale prototype is introduced, an architecture that enables an application not only to vary the content format generated for each particular device, but also allows the author to define a device-specific view on the application's data and features, thus providing optimal application interaction for each device.

Whale acts as a bridge between the user's receiving device on the one side and an application server's back-end data sources on the other. The back-end data sources may be transaction engines, in which case Whale provides a suitable access "window" into those transactions, allowing the user to select among

the offered transactions, enter the relevant parameters, trigger those transactions, and retrieve the resulting data sets. Alternatively, the back ends can merely be plain data sources (such as databases, data warehouses, news feeds, etc.), in which case Whale serves in addition as the hosting platform for the actual logic of the application. The JavaServer Pages** (JSP**) mechanism has been selected to drive the overall Whale application architecture. JSP pages potentially have a number of advantages over the common stylesheet approach. The most relevant of these is their ability to poll the application logic from within the output template; if suitably used (not necessarily in the way the JSP specifications advertise their usage), then one can guarantee that only the application logic for a specific device is being executed. Whale also enables the separation of the JSP-based content presentation and the bean-based content generation by introducing the WhaleInvoker, which dynamically selects and executes the appropriate combination of JavaServer Pages and JavaBeans** to satisfy a data request from an end-user device.

The paper concludes with a summary of the history of, and the experience gained from, the first commercial deployment of the Whale server infrastructure—Swissair's Easy Check-In service.

Sailing the "sea of devices"

Today the World Wide Web revolves around browsers running on bulky, mostly stationary, desktop personal computers, rendering a content format known as HTML.[1] This dominance is about to end: a large number of diverse, browser-based user devices are emerging that are using an almost equally large number of diverse content formats, communication protocols, and user interfaces. HTML, a derivate of SGML[2] (Standard Generalized Markup Language), has, in its brief history, spawned a number of derivatives for alternative form-factor devices as well as alternative uses. None of these derivatives, however, managed to gather sufficient momentum to become a threat to HTML. More recently, focus has shifted away from the markup languages to alternative form-factored devices. It is these devices that are now driving the requirements for markup languages and are demanding adapted versions of HTML. Such form factors differ from the conventional HTML-centric devices, such as PCs, not only with regard to their input and output capabilities, but also, and more significantly, in the way they interact with the user. This includes alternative modalities (such as voice) as well as alternative usage patterns (such as mobility). Some of those devices are depicted in Figure 1. The common feature among these devices is their browser-centric architecture: content is transmitted from the server to the client in the form of a marked-up document, all interaction between the users at the client device is orchestrated through a user-interface that, although controlled by the server through the marked-up document, is generated and maintained by a browser program that is independent of the server and its application and resides permanently on the device. The user interface itself is then not necessarily display-based, but may very well be a voice interface, controlled for example by VoiceML[3] (Voice Markup Language) documents. Such browser devices are also frequently referred to as "thin" clients, not because the browser infrastructure on the device is necessarily thin or the device features only compact hardware, or limited storage or memory, but because the application is maintained exclusively on the server—apart from simple caching, no aspect of the application is resident on the client beyond individual request-reply boundaries.

Such applications have several advantages: no client-resident part needs to be maintained or managed, which in turn allows the application to be upgraded easily by upgrading only the server; the client is robust because the application's state and data can easily be regenerated from the server side; and the browser, through the markup language, provides to the application programmer a high level of abstraction from the hardware and software details of the end-user device. On the down side, any application interaction must be exercised between the client and the server, which, depending on the latency of the network involved, may prove slow. This can be remedied, to some extent, through a local scripting capability, which is supported by most browsers and enables at least some level of dynamic content generation on the client.

Taking such scripting capability to the other extreme, "thick" clients are based merely on such a capability. Instead of the browser, a more or less generic client execution environment provides application support on the device. Application developers can program for such an execution environment and, within that environment, are free to organize the user interaction as well as the interaction with the server-resident part of the application (if any). It should be obvious that such programming models give the programmer maximum freedom to generate the "look and feel" of the application, albeit at the price of having to develop the software for the particular pro-

Figure 1 Sailing the “sea of devices.” Shown here (clockwise from top left) are smartcards as an example of a pervasive authentication token, IBM's Watchpad (two images), an IBM wearable computer prototype, the Palm Pilot™ as an example of a connected personal digital assistant, two IBM microdrives, and two i-mode smart phones.

gramming model provided by a specific device infrastructure. General-purpose development platforms, such as the Java** virtual machine and its environment of libraries and utility functions, attempt to provide execution environments that have applicability across device boundaries. What seems to prove difficult is to provide a generic API (application programming interface) to the device-interaction aspects of the device. A compromise must be found between granting applications tight control over the display and input features of the device, while at the same time shielding them from too many device-specifics. Both Java’s Swing and Abstract Windowing Toolkit (AWT) attempt to provide such an interface and, whereas both have succeeded in some aspects, they fail in others: Swing is big and slow, AWT is low level. Both approaches are also in danger of enforcing a particular look and feel, hence reducing the freedom bought by the application programmer by not using browser infrastructure on the device. In that respect, these user interface libraries are very similar in their device-abstraction levels to browsers and do not, in effect, provide a significantly higher level of control over the user interface characteristics or interaction with the applications. (For more details on Java technology, see Reference 4).

In summary, the overwhelming success of the browser-centric World Wide Web is a strong endorsement of the thin-client approach. However, with scripting capabilities being added to many browsers and with browsers growing in terms of capabilities and performance, the boundaries are beginning to blur. Local device storage, once a premium, is now cheap enough to enable much larger quantities of data to be held locally than could reasonably be downloaded over some ubiquitous (e.g., mobile) communication networks. Application models make use of such lo-

cal storage by embedding microservers into the device itself, which may be regarded as proxies for the server on the browser device.

Browser performance may also play a role in deciding between the two models. As of now many browsers are much slower for some applications (such as games) than native GUI (graphical user interface) programming and provide a much higher level of abstraction between the application and the display. Applications that require close control over the display and/or high interaction speeds with the user/server stretch the capabilities of some of today's browsers.

Enabling server-based applications to support a "sea of devices" is a complex endeavor.

Concepts in service provisioning

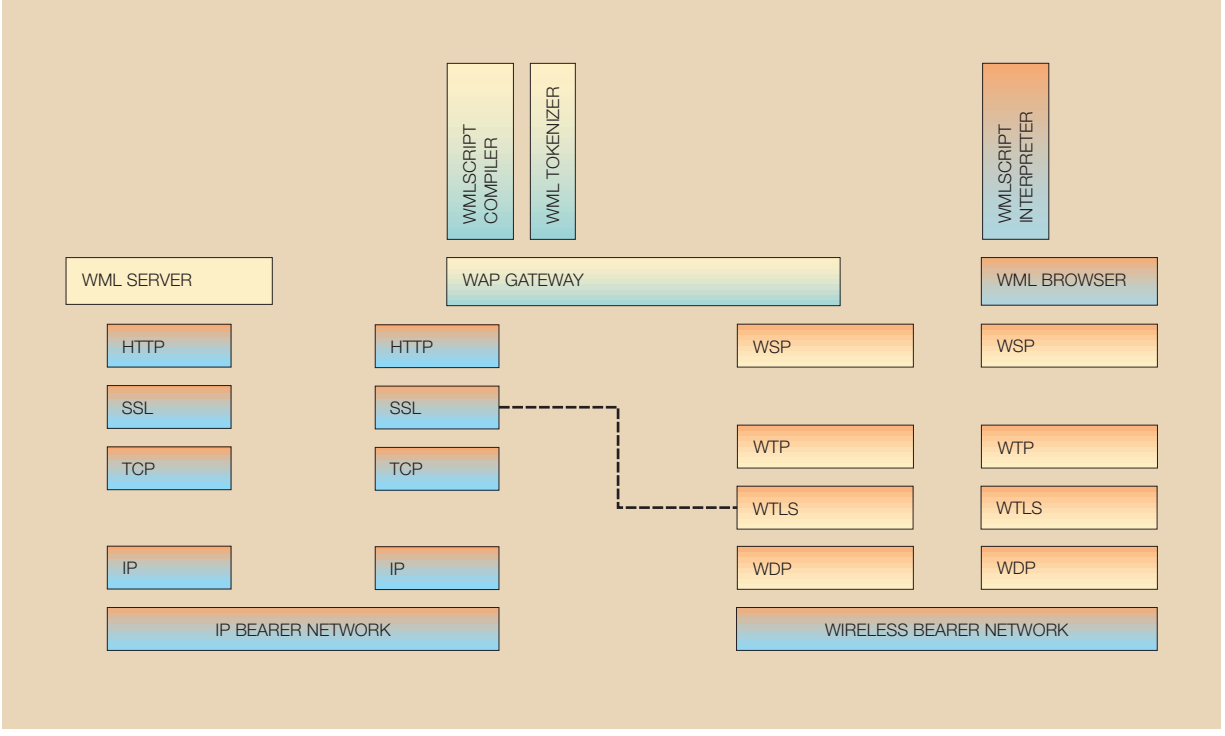
Enabling server-based applications to support a "sea of devices" is a complex endeavor. The variety of devices and browser infrastructures require dedicated gateways and application servers on all three levels of the application/communication stack: the "protocols" layer that provides the rudimentary communications primitives, the "application" layer that makes use of such communications, and the "session" layer that is the glue between the two.

As far as the protocols layer is concerned, communication is accomplished primarily using the TCP/IP (Transmission Control Protocol/Internet Protocol) suite of protocols,[5] conveyed by standard bearer networks such as Ethernet or token ring in the wired world, and CDMA (Code Division Multiple Access), and in the future GPRS (General Packet Radio Service), in the mobile/wireless world. However, islands of alternatives do exist, in particular in the mobile area. Notable here is the WAP (Wireless Application Protocol) suite of protocols,[6] the WAP Datagram Protocol (WDP) and the WAP Transaction Protocol (WTP), where the first is essentially User Datagram Protocol (UDP), the latter a reincarnation of the "transactional TCP."[7] Most existing wireless cellular communication systems also implement proprietary communication stacks; i-mode,[8] the Japanese competitor of WAP, for example, uses the datagram service built into the Japanese personal digital communication (PDC) cellular system, which is itself based on an earlier cordless standard. Bridging between such proprietary systems or WAP to standard TCP/IP is possible only when protocol conversion is involved. However, with some standards such conversion usually impacts the security layers that are employed in such systems. WAP, for example, specifies its own security layer WTLS (Wireless Transport Layer Security), which is placed between WDP on one side and WTP on the other. Conversion above the WTP layer then requires that the secure session be terminated at the bridge and thus interrupts the end-to-end secure session between the client and the server. Hence, the bridge must be a part of the server infrastructure and cannot be a function that is relegated to a telecom or Internet service provider (ISP). Figure 2 shows a typical WAP setup, including the bridging function into the wired TCP/IP world.

On the application level, the browser and the client application execution environment differ both in syntax and semantics: from a syntax point of view, the Internet is currently centered around HTML. HTML started as a simplification of SGML and has since grown organically into something that is probably more complicated and certainly less structured—so much so that a subset of HTML has emerged, i.e., Compact HTML[9] or cHTML, which is used for i-mode among other things. The Wireless Markup Language (WML),[10] on the other hand, was specified for WAP and is based on an XML[11] (Extensible Markup Language) document type definition (DTD), which at least ensures some level of commonality to other XML-based content representations. Semantically, cHTML is a true subset of HTML, whereas WML was developed specifically for a particular use (the smart phone) and thus has a range of features that are useful in this domain, such as built-in support for soft keys, the notion of "cards" to segment content into small renderable units, and a range of tools to navigate between such cards. This variety of content formats is supplemented by an equally broad range of scripting and programming environments that are featured in such devices (e.g., HTMLScript for HTML-based devices, WMLScript for WML-based devices, and so on).

Finally, on the session level, much browser interaction is now accomplished with HTTP[12] (HyperText Transfer Protocol) 1.1-compatible service primitives,

Figure 2 WAP protocol stack and browser (on the right) together with the WAP gateway (middle) bridging to a conventional TCP/IP-based wired network connection and an HTTP server (on the left). All protocol layers are shown, including the security layers (SSL on the wired side, WTLS on the WAP side). Note the "mismatch" of layer functions on the WAP side!



e.g., GET, PUT, etc. The HTTP traffic itself is often transmitted in plain ASCII (American National Standard Code for Information Interchange), such as in the conventional "wired" Internet; WAP, on the other hand, specifies an otherwise compatible binary encoding mechanism called Wireless Session Protocol (WSP).[13] Among other standards, WSP also features an extension to HTTP 1.1 to enable push communication from the server to the client.

Switching from the client side to the back-end side of a WAP or Web application, the variance is equally broad. Very few of the data that are currently accessible through the Internet in HTML form are actually stored in that form. Typically, back-end database systems and data warehouses are accessed through a sometimes complex chain of data processing systems. The conversion of the resulting data set into HTML, the "front-ending," is logically the last step before the data are transmitted to the client device. However, many of the existing services have grown over time; nowadays, one is frequently faced with back-end systems in which the logic of the application is no longer architecturally separated from the front-ending into the required client format. One simple example of such a setup is the popular CGI (common gateway interface) scripting mechanism. Here, HTML content is generated dynamically as a side effect to processing an incoming HTTP request for data. This is accomplished essentially by calling an executable component that is expected to generate HTML as its output. From an architectural point of view, such a setup is less than desirable because there are no enforceable specifications with regard to the exact inner working of such scripts. Many such scripts mix freely the generation of the output HTML with the application logic itself. It is no surprise that such a setup complicates the addition of support for alternative device types, where the front-ending step needs to be modified, because such modifications must occur between the application logic and the client and may not even be isolated in one particular

script or component, but must occur throughout the entire data chain.

Although there exists a growing but still limited number of standards for the overall back-end architecture, capabilities, and setup of the client devices and client infrastructure, the IT (information technology) shops of application service providers tend to differ. Standard products and infrastructure elements are still being employed, such as TCP/IP and MQSeries,[14] CORBA**[15] (Common Object Request Broker Architecture**), or any of the RMI/RPC[16] (Remote Method Invocation/remote procedure call) solutions for interprocess communications; Java or servlet[17] execution environments such as JServ[18] or IBM's WebSphere* AS[19] (application server) for hosting logic elements; and data connectors using SQL[20] (Structured Query Language) for database access or DOM[21] (Document Object Model) for data exchange. However, their interconnections and the distribution of functionality of the entire application among the various infrastructure elements remain proprietary and specific to each installation.

Given the huge investments that each of those infrastructures represents, it should be little surprise that application service providers are tempted to prolong the lifetime of that infrastructure as long as possible. In particular, adding support for new devices should ideally be an additive process with as little impact on the existing application as possible. In some cases, complex back-end systems cannot be reengineered, either because detailed knowledge of the workings of the system is lost when its designers are no longer available, or because lengthy modifications to the back end would inevitably lead to system downtimes that could impact the day-to-day running of the business (an earlier treatment of this topic can be found in Reference 22).

The desire to extend services to new pervasive devices with the fewest possible modifications to the already existing Web infrastructure is, therefore, at the top of most IT managers' wish lists. Transcoding, the process of transforming data from one representation into another, appears to be an attractive solution.

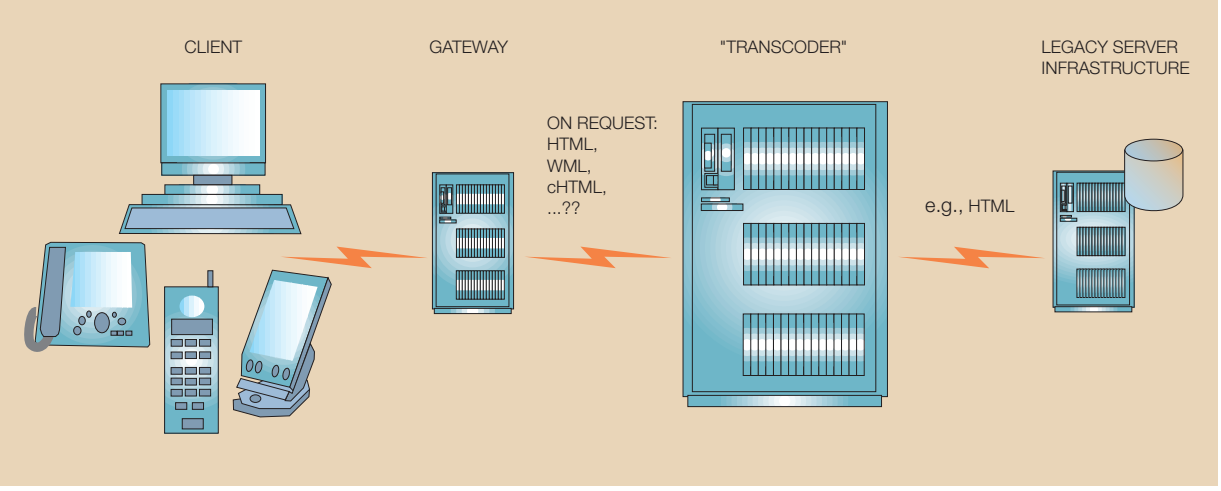
Transcoding. Transcoding solutions are not new and have frequently been employed in the past at the lower levels of the application/communications stack. Essentially, transcoding is necessary whenever two computer systems that are not completely compatible need to communicate (such as a client computer running a browser and a server providing marked-up content). In particular, transcoding is employed frequently within communication networks to resolve incompatible packet formats between two different networks (e.g., SNA [Systems Network Architecture] and TCP/IP networks) or to resolve symbolic addresses when a packet is transmitted across the network or is being routed.

Pushing transcoding up to the higher levels within the application stack, to the application itself, requires transcoders to operate on the application data, with much broadened semantic and syntactic content (see Figure 3 for an example of such a setup). Leaving aside the security implications for the moment (which can essentially only be alleviated by assuming that the transcoder resides, if not together with the application server, certainly within the same security domain), transformations at the application layer are not easy in the general case. For example, transcoding an original HTML page that is designed to be viewed on a desktop PC into a WML version, which is to be viewed on a mobile telephone handset that may allow 4 lines of 10 characters each, can hardly be an automatic process. On the other hand, removing the color information from an image because the receiving device features only a black-and-white display is feasible.

The bottom line is that transcoders can be made to work, but must be "trained," e.g., by use of examples, or programmed to execute the desired modifications. Any change in the back-end system is likely to require additional programming and training of the transcoder. The process of programming or training such a transcoder can be elaborate. (The exact workings of such transcoder systems are presented in more detail elsewhere in this issue).

Even with advanced transcoder engines, two problems still remain. First, transcoders typically operate on an interaction-by-interaction basis and they are not usually designed to diverge from this mode. However, in order to adapt to the characteristics of the device, it is necessary to modify the data flow across interactions. For example, consider the very popular Travelocity[23] Internet application, which allows users to get quotes for and purchase airplane tickets, as well as to make hotel and rental car reservations. The airline reservation application requires about seven mouse clicks from entering the Travelocity site to making the final purchase (assuming the user has already signed up for the Travelocity service, which itself takes many more interactions

Figure 3 "Mission impossible"? A transcoder translates existing content formats into whatever format is required. In practice, transcoders require programming and training to accomplish this feat and limitations still remain.



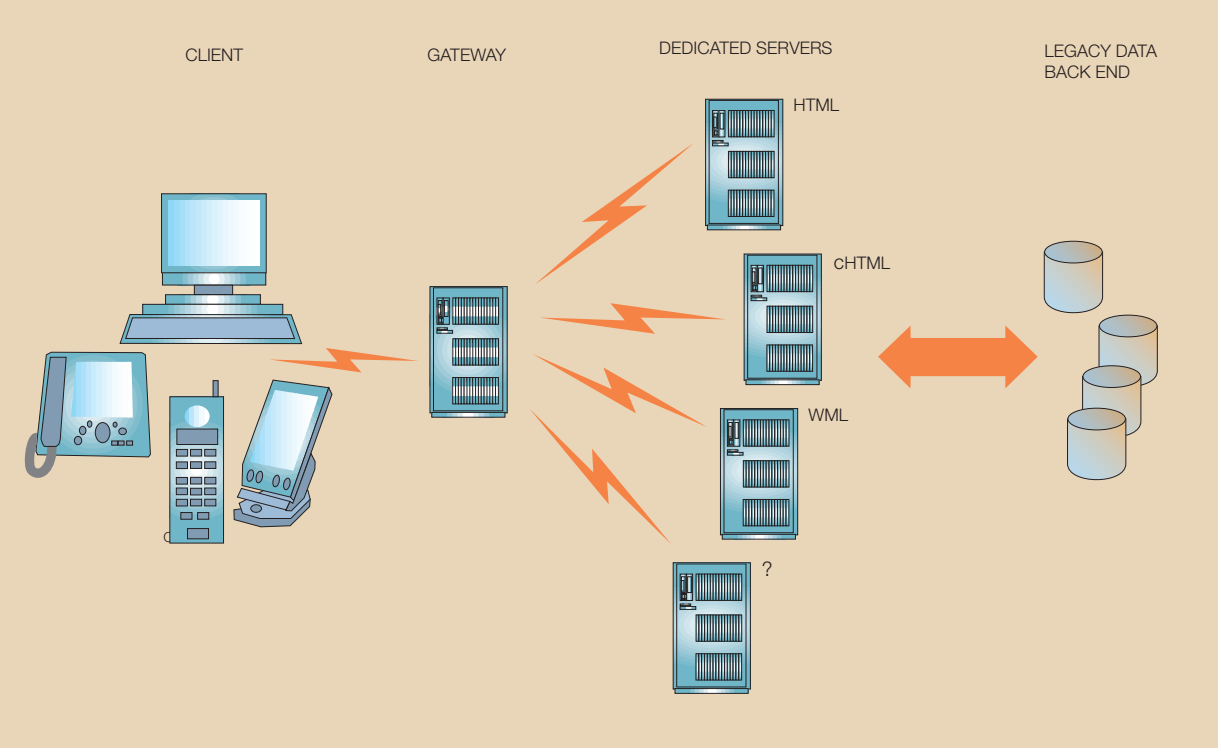
but needs to be done only once). A transcoder supporting this service on a mobile personal digital assistant (PDA) device could, with some programming or training, translate the relevant HTML pages into a suitable format for the mobile PDA, but it would still deliver seven HTML pages that need to be navigated. Ideally, one would expect the flight-booking application for the mobile device to require fewer interactions, perhaps by sacrificing some of the less frequently used options, by using a more elaborate customer profile, or by employing location information to "guess" certain parameter values and thereby streamline the data flow. If a user wishes to purchase an airplane ticket using a mobile device, one might reasonably assume that the request is urgent (otherwise why not use a more convenient Web interface?) and infer that the departure city is the user's current location.

Second, transcoders usually support existing services. If the new devices require or permit new services or forms of interaction, transcoders have difficulty supporting them. One good example is the "push" capability that many pervasive devices support, a feature that is by and large rare in the existing Internet/Web world. Although this feature is sometimes highly valued (for the above-mentioned travel service, push is the ideal way to notify a passenger of flight delays), if the back-end systems do not already support push features, it might be difficult to imbed these in a transcoder.

In view of these disadvantages it is not surprising that, over time, many solutions that originally started as "plain" transcoders have evolved into full-fledged application platforms, with the transcoder no longer merely performing transformations on data, but also executing application-specific logic to those data to enhance the interaction pattern of the application for the target device, or adding new features such as push functions to a legacy application. One might view this, of course, as a natural development, but in effect it is adding logic to the back-end system and thereby adding to the fragmentation of logic across the various components. Furthermore, it is questionable why such applications should process data formatted for other front ends rather than access the data at their source. Once this has been achieved, a transcoder has fully mutated into an additional application platform (next to the original server), not unlike a dedicated server.

Dedicated device-class infrastructure. A dedicated device-class infrastructure consists of a replica of any existing Web back end, with modified front-end generators for alternative content format, and potentially augmented application logic and behavior to accommodate device characteristics such as input, output, browser capabilities, and usage patterns. Dedicated servers may be put in place for alternative device classes, such as a dedicated WML server for WAP phones—hence the term *dedicated device-class infrastructure* (see Figure 4). Providing such in-

Figure 4 Service provisioning through dedicated servers for each device class

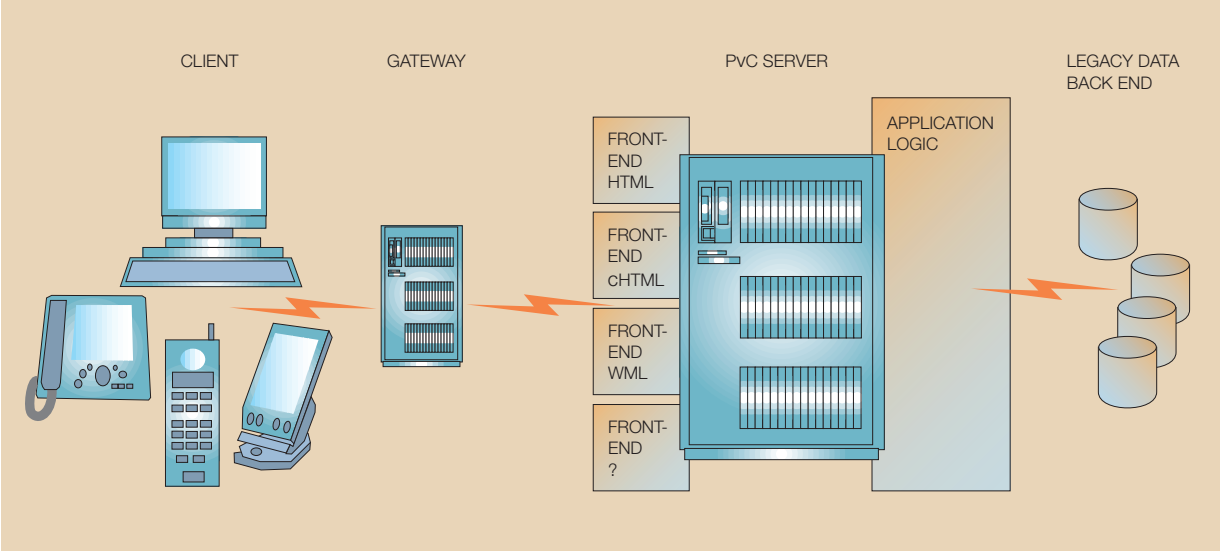


frastructure for each device class is frequently the fastest way to accommodate a new device class; as outlined above, it is often much quicker to provide a completely new front-end generator with specialized back-end functions, than to accommodate new front-end requirements within the existing front-end infrastructure.

Many service providers have resorted to such an architecture, most of them fully aware that it can, at best, be a temporary solution before a complete "integration" can be achieved. In the meantime, such a setup helps establish market share and gain that all-important "first-mover" advantage. It also enables services to be nicely tuned to the particular characteristics of the individual device class, allowing device-specific service provisioning. Unfortunately, those advantages are often more than offset by the cost of installing and maintaining such back-end infrastructure. Any new device class requires comparatively high startup costs and continuous maintenance thereafter: any new feature or update that gets rolled into such an application must be added to each dedicated server separately for each device class. As device manufacturers show no signs of slowing the pace of developing new devices and device classes, the number of such "dedicated server infrastructures" should necessarily grow rapidly and quickly become unmanageable. Furthermore, even within each device class, different device models show sufficient differences in capabilities and user interface to necessitate slightly augmented service offerings for each version, further adding to the burden by requiring additional specialized servers.

Rehosting the application. The preceding discussion raises the question of how application service back ends can be re-engineered to allow their services to be accessed from a variety of different browser-based end-user devices. The requirements for this application "rehosting" seem daunting. The application server has to identify the type and version of the receiving device; the output that is being generated in the "front end" has to be adapted to the class of device—i.e., a different markup language (HTML, WML, cHTML) must be chosen depending on the device

Figure 5 Content provisioning through a server for pervasive computing (the PvC Server) that differentiates between available content logic and device-dependent front ends, thus providing adapted and usable services to all device classes from within a single-server architecture



browser. The exact content of each page must be augmented depending on the device model because screen size and display capabilities vary among different models of the same device class (see Figure 5). The application flow must be modified in accordance with the overall device characteristics and usage models. Finally, individual features of the application may be exposed or hidden, depending on the browser's ability to handle them (for example, push functions).

It should be obvious that fulfilling all these objectives requires significant modifications to the server and back-end infrastructure. In essence, it requires a solution to the above-mentioned problem, the lack of a clear architectural distinction between the application logic, the data, and the presentation of those data, i.e., the front-end user interface. In many cases HTML is used to convey all these three elements, amalgamated into a single text stream. Indeed, it must be considered one of the major failings of HTML that data content and data representation are not separated. As mentioned above, due to the organic growth of many server back-end infrastructures, such separation was rarely realized. Within the IETF[24] and W3C,[25] the two standard bodies largely responsible for the development of Web technology, several efforts have recently gone a long way toward providing the technological foundation to achieve such a separation. Some of the most relevant of those technologies are being developed under the auspices of the XML working groups.

The XML effort comprises a number of specifications, in particular the XML[11] metalanguage for specifying markup languages, Extensible Stylesheet Language[26] (XSL) and Extensible Stylesheet Language Transformations[27] (XSLT) to manipulate XML data sets and potentially create front-end generators.

The syntax of a language to be specified using XML is contained in the so-called document type definition (DTD). Various such concrete languages have already been specified using XML, among those VoiceXML[3] and WML[10] of WAP.

Yet another popular mechanism for dynamically generating content in some markup language without mixing the particular syntax of that language with the generation of the dynamic content is JavaServer Pages[28] (JSP).

JSP pages/beans. JSP pages control the content of the HTML pages sent to the client. JSP pages contain an escape syntax used to include Java code fragments and, in particular, JavaBeans property access meth-

Figure 6 A sample JSP, accessing a bean representing a set of train connections and rendering these connections in WML syntax

```
<%@ content_type="text/vnd.wap.wml" %>
<bean name="Connections" type="com.ibm.zrl.Connections"
      introspection="yes" create="yes"></bean>

<?xml version=1.0?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
    "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
    <template>
        <do type="prev">
            <prev/>
        </do>
    </template>

    <card id="Train connections">
        <p>
            Results:
            <repeat index=i>
                From: <%= Connections.getFrom(i) %><br/>
                To: <%= Connections.getTo(i) %><br/>
                [...]
            </repeat>
        </p>
    </card>
</wml>
```

ods. For example, the JSP compiler contained in IBM's WebSphere AS uses "<%@" and "%>" to encapsulate compiler directives, "<%=" and "%>" to indicate accessor methods that must be evaluated, and "<%" and "%>" for arbitrary Java code fragments.

The JSP page compiler translates these templates into Java servlets whose generated output consists of the markup data into which the string-type values of the referenced JavaBeans properties are inserted; this step may either be carried out during preprocessing (at install time of a JSP on the server), or the first time a JSP is being requested from the user device. Any subsequent call to the same JSP will simply invoke the generated servlet code; hence, the performance impact of having to execute the JSP compilation is of consequence only the first time a JSP is requested.

JavaBeans[29] are Java components that provide a codified set of access methods to set and get the values of properties. The bean's properties have names, and the naming convention for access methods is to concatenate the operation with the property name. For example, a bean property "prop" would be accessed through access methods getProp( ), which by common definition would return a Java "String" object. Setting the value of "prop" is through method setProp(String value). Properties can also represent an array of values; this is termed "indexed properties," for which the access methods take an additional indexing parameter. Indices that are out of bounds are indicated by raising the Java exception ArrayIndexOutOfBoundException.

One inherent feature of the JSP/bean model is introspection. At call time, every argument that is passed to the URL (uniform resource locator) in the HTTP-typical name-value fashion is interpreted as a set-Property call to the underlying beans behind the requested JSP. Hence, bean properties can be explicitly set by passing appropriate argument strings to the URL call itself. A simple example of a JSP and a corresponding bean is provided for illustration purposes in Figures 6 and 7.

From the application developer's point of view, JSP combined with such JavaBeans divide the applica-

Figure 7 A skeleton implementation of Connections.java. Properties may be set by using the introspection feature of JavaBeans; for example, if the JSP resides on the local host as train.jsp, connections from "New York" to "Boston" may be retrieved by sending: http://localhost/train.jsp?FROM=NewYork&TO=Boston.

```
class Connections {
    // properties for desired itinerary
    String origin, destination;
    // indexed properties for possible train times
    String[ ] from, to;
    // itinerary current?
    boolean current = false;

    // accessor methods for desired itinerary;
    void setFROM(String f) { origin = f; };
    void setTO(String d) { destination =d; };

    // accessor methods for actual itinerary; db request as side-effect
    String getFROM(int i) throws java.lang.ArrayIndexOutOfBoundsException {
        if (!current) {
            getConnections();  // go to train db, get connection information, fill arrays from[ ] and to[ ]
            current = true;

        }
        return from[i];

    }
    [...]

}
```

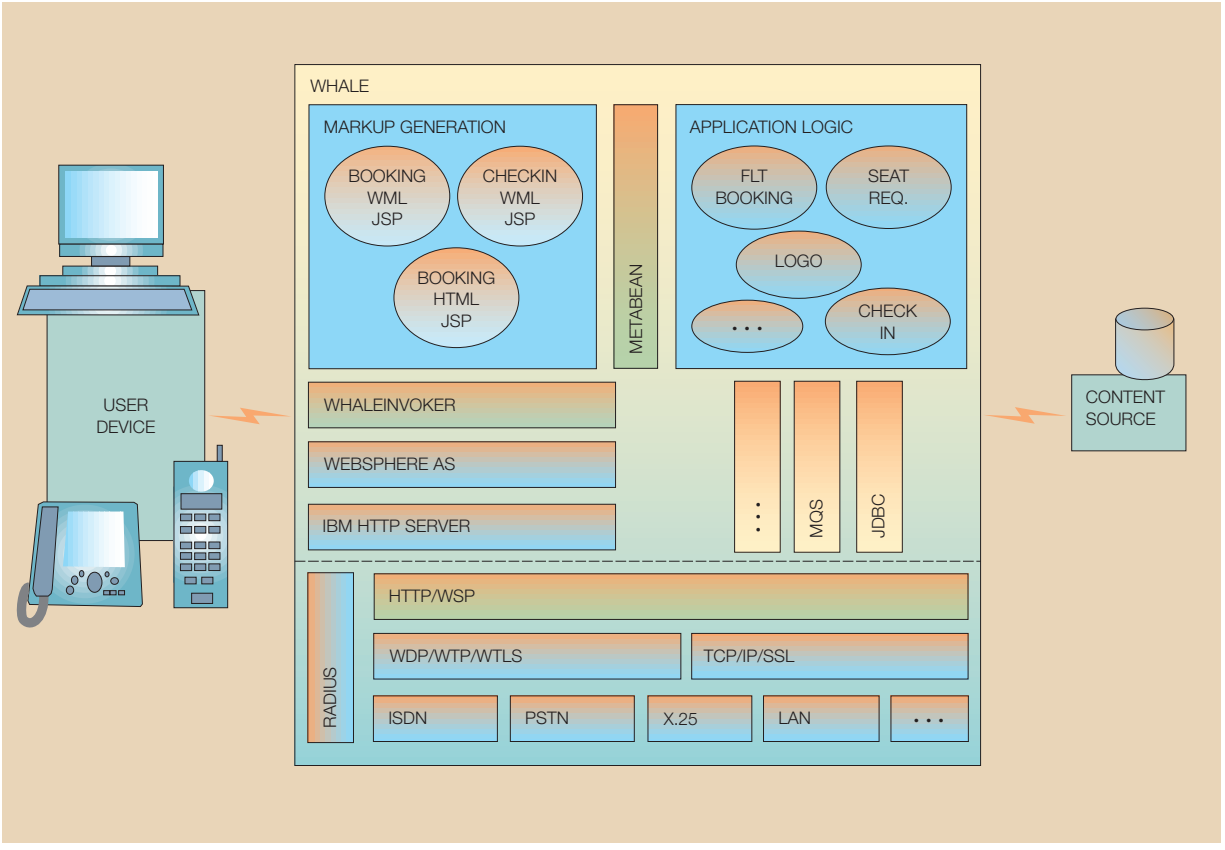
tion into the user interface aspects embodied in the JSP pages' markup and the data access encapsulated in the JavaBeans' components. How a given bean obtains the actual values for its properties is entirely open to the bean implementor; the entire Java language and its supporting APIs can be used to embed application logic as well as access logic. A more modular design of the application thus becomes possible with the well-known advantages of modularization, such as reuse of components, ease of maintenance, etc. Authoring tools have been developed that allow such JSP templates to be generated in an essentially WYSIWYG (what-you-see-is-what-you-get) manner.

Whale

"Whale" is the name of a research project at IBM's Zurich Research Laboratory, and is a loose fit to the more descriptive name "Wireless Application Hosting Environment," although the scope of the project soon expanded beyond wireless receiving devices. Whale is an attempt to provide an execution environment for applications that can correctly interact syntactically and semantically, with a large set of browser-based receiving devices. Whale started in early 1999; the approach taken has proved its validity in a well-publicized project with Switzerland's national airline Swissair, which now offers a WAP-based check-in service to selected customers using a Whale application server in the back end. In this and the next section of this paper, the Whale architecture and the Swissair application are introduced.

Whale architecture. Whale can be viewed as a bridge between the user's receiving device on the one side and an application server's back-end data sources on the other (see Figure 8). The back-end data sources may be transaction engines, in which case Whale may simply provide a suitable access "window" into those transactions, allowing the user to select among the offered transactions, enter the relevant parameters, trigger those transactions and retrieve the resulting data sets. Alternatively, the back ends can merely be plain data sources (such as databases, data warehouses, news feeds, etc.), in which case Whale serves in addition as the hosting platform for the application logic. Common to both configurations is the need to provide a single communications platform to the set of receiving devices.

Figure 8 The Whale server attempts to enforce a clean separation between device-independent application logic and device-dependent front ends by adding the WhaleInvoker component and the MetaBean component.



This communications platform is provided using the HTTP protocol layer as a common abstraction. As pointed out above, HTTP is already widely used in many Web-based client/server systems. When HTTP derivatives are used (as in WAP, which uses WSP, a binary version of HTTP), there is a clear mapping to HTTP primitives, except for features, such as "push," that have no clear counterpart in HTTP. Here, the common solution is to overload HTTP directives and add header field information.

Whale is, therefore, based on the conventional set of IBM middleware products, in particular the eNetwork Wireless[30] product suite, which already supports a large number of network-proprietary communication standards and bridges those into TCP/IP, together with the required remote access functions, such as RADIUS[31] or other third-party provided authentication services.

eNetwork Wireless has recently been completed with the necessary bridging functions to operate as a gateway for WAP, thus also providing a bridge to HTTP. Security is provided here in the low levels of the protocol stack, either in the form of WTLS for WAP, or SSL[32] (secure sockets layer) or IPSec[33] for TCP/IP. Sessions are identified in a manner that is specific to the underlying bearer service, and may be based on the source address of the incoming requests (e.g., in the case of IP connections, or on equivalent packet-based mobile links, such as GPRS- or CDMA-connected mobile clients), caller-line identification (e.g., with circuit-switched dial-up links), or application-level cookies if supported. It is specific to the Whale setup that a session can also be identified based on header information carried with incoming requests, i.e., packaged as "application level" data. This feature has been added to counter either the lack of support for similar services from the network oper-

ators, or the great variety in which such support is granted by different network operators. In addition, many of the existing client authentication schemes do not operate under all conditions; caller-line-identification information, for example, does not travel beyond the boundaries of the home network. Mobile users that have roamed into an alien network can therefore not be authenticated using such a feature.

Once established, the session information is continuously updated and made available to the applications sitting on top of the middleware either through a session database or additional header fields that are carried across the HTTP API. In the future, user and session management functions will be developed further and APIs provided for third-party subscriber management systems or Tivoli's "TSM."[34]

Once an incoming call has been identified and accepted, the incoming application level data are carried through the HTTP API to a standard set of Web-hosting products, such as IBM's HTTP server and WebSphere AS (application server). It is at this application level that Whale diverges from most conventional server configurations. Here, Whale aims to provide a framework for application developers that forces the developer to clearly and cleanly separate the two fundamental elements of a server application—the application logic on the one side, and the generation of the user interface and the data representation on the other. It is worth noting at this stage that Whale allows applications to be triggered using the same URL for any device; application service providers want to be able to advertise a single URL rather than one for each device type.

In deciding on the fundamental technology to provide that separation, the decision has been taken not to use "style sheets." Style sheets are essentially applied to previously generated data sets to generate a device-dependent representation of such set. The problem with this approach is that such a data set must be generated up front; this is usually accomplished through some programmed application element, typically a servlet. If that servlet is device-neutral, it must implicitly assume the richest possible client device, and completely "fill" the data set. In practice, this may mean that a travel application, for example, generates a rich data set about some travel destination, including pictorial and video data about a journey's destination. If browsed from a powerful desktop device, the entire data set may be available once a suitable representation is generated by an HTML style sheet. However, if the same request was made from a much simpler WAP-capable cellular handset, a different style sheet is applied that prunes much of the data, maybe 99 percent of the generated data volume. In such a setup the load on the server is not correlated to the data requirements of the receiving device—on average, the server generates more data than are required.

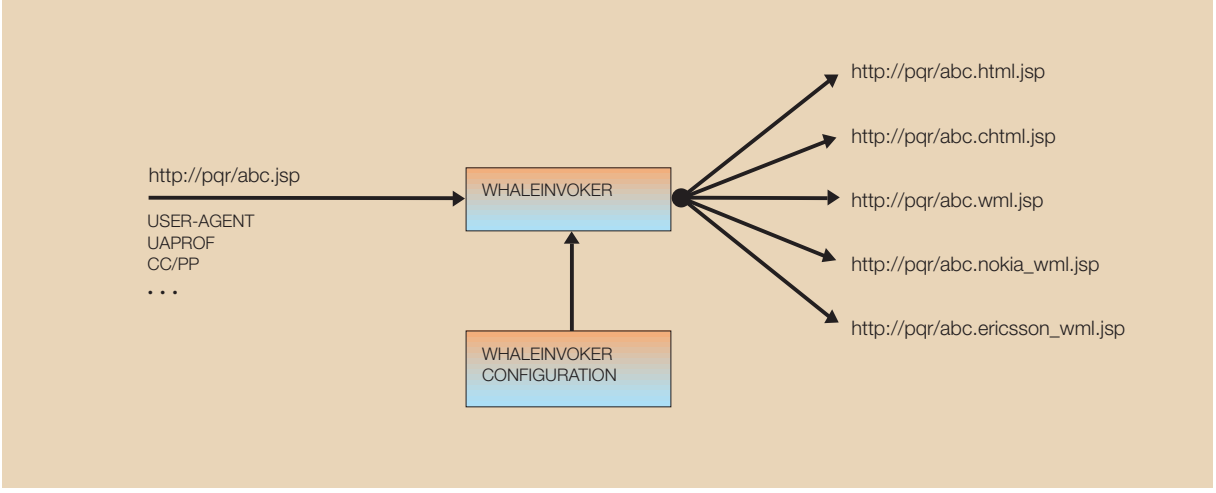
Alternatively, the servlet may, of course, implement logic to differentiate between receiving devices and then augment its data-generation behavior as appropriate. In that case, however, the servlet is no longer device-neutral and the implementor is again forced to mix device-dependency with the application logic inside that servlet. Supporting additional device types, then, requires fundamental changes to the entire application stream.

Instead, the JSP mechanism has been selected to drive the overall Whale application architecture. JSP pages potentially have a number of advantages over the style-sheet approach. The most relevant of those is their ability to poll the application logic from within the output template; if appropriately used (not necessarily in the way the JSP specifications advertise their usage), then one can guarantee that only application logic that is relevant to a specific device-dependent markup is being executed. Second, JSP pages can be authored easily using existing authoring tools and provide an intuitive framework to encode the application's user interface by virtue of specifying the menu structure through which the application's logic is made accessible.

Consequently, Whale uses JavaBeans or Enterprise JavaBeans** for the logic aspects of the application. As pointed out above, those beans either improve application logic directly, or act as wrappers and accessor methods to databases or transaction systems in the back end of the application service provider.

However, the standard JSP mechanisms do not fit the bill entirely for two reasons: first, they are tied to specific URLs—different JSP pages for two different device types will also have two different URL addresses. Second, the JSP syntax requires a comparatively tight coupling to the application logic components, the beans. Although not a concern per se, this requires that JSP templates need to be "touched" whenever the configuration of the application logic changes, and vice versa. Whale, on the other hand, aims to provide enough separation to update and modify the application logic without requiring ad-

Figure 9 The WhaleInvoker analyzes the incoming requests and attempts a classification of the device. The requested URL is then rewritten according to the device classification.



ditional changes to the JSP pages and, equally, to update the JSP repository (for example to add support for a new device), without having to modify the application logic. Consequently, in addition to providing effective tool support for the generation of JavaBeans (see the subsection on bean tools later), two components have been added to the WebSphere environment, which is at the core of Whale, to alleviate these concerns. The two components are the WhaleInvoker and the MetaBean.

WhaleInvoker. The WhaleInvoker replaces the standard Invoker in the WebSphere environment. It is the component in WebSphere AS that is presented with incoming URL requests and has the responsibility to fan those requests out to either the servlet runner, the JSP compiler, the style sheet interpreter, or any other execution environment.

The WhaleInvoker has the additional task of gathering the device type information from the request and redirecting the requested URL to the correct URL for that device type. The type of device can be identified using one of three mechanisms. All browsers today (including pervasive devices' browsers such as Palm's cHTML browser or the WAP-typical WML browsers) send user-agent information as one of the standard header fields with every request. It is unfortunate that the exact format of that header field is not specified; by common convention, browsers encode their type and version as strings. The WhaleInvoker can match the user-agent field against a known set of user-agent fields (fuzzy matches are also possible) and thus infer the correct device type. In the future, browsers may also send structured user-agent information. Two closely related standards exist for structuring the said information; in the Web space, the World Wide Web consortium is promoting the composite capability preference profile[35] (CC/PP) for that purpose. Within the WAP Forum, the same functionality is achieved using the closely related User Agent Profile[36] (UAPROF). Both standards use the Resource Description Framework[37] (RDF) as their encoding base, which in turn is an XML-based language. One interesting feature of both CC/PP and UAPROF is that they are hierarchical. Parts of the complete user agent definition are static and can reside on hosts in the Internet; those parts can then be referenced by a URL from within the actual CC/PP or UAPROF component, which may also include the more dynamic parts of the complete profile.

The WhaleInvoker is set up by a configuration file, which holds a list of all supported device classes and identifies rules for each one. Once the device class is known, the URL redirection is executed by simply rewriting the requested URL. Essentially, a request to URL "http://pqr/abc.jsp" is augmented by inserting device-type information, for example by rewriting said request into "http://pqr/abc.wml.jsp" if the request is coming from WAP devices, or into "http://pqr/abc.html.jsp" if the request has its origin on a conventional HTML-based browser (see Figure 9 for an illustration of this process and Figure 10 for

Figure 10 A sample Whale configuration file, specifying two device classes. Additional device classes are easily added using a simple command language.

```
# what kind of device classes we can distinguish:
DeviceClasses=wml,html

# actual user agent fields for each device class:
wml=Nokia7110,EricssonR320
html=Mozilla,Netscape,Internet Explorer

# accept headers that may indicate a device class,
# used in case user agent is unavailable
wml_content=text/vnd.wap.wml, text/vnd.wap.wml-wbxml, text/vnd.wap.wmlscript,
      application/vnd.wap.wmlscriptc, application/vnd.wap.wmlc, image/vnd.wap.wbmp
html_content=text/html, text/plain

# error files
error404=errorPages/404.jsp.whale
error500=errorPages/500.jsp.whale
errorNoUserAgent=errorPages/noUserAgent.html.jsp
```

an example of a WhaleInvoker configuration file). The device class is inserted in front of the final type-extension (".jsp" in this case) in order to allow standard mechanisms within WebSphere AS to handle the correct fan-out to the execution environment, in essence by calling the standard WebSphere AS Invoker from within the WhaleInvoker.

A backtrack mechanism is built into the WhaleInvoker, which is triggered if the rewritten URL cannot be located, i.e., if it returns error code 404 in the status line of the resulting stream. The backtrack mechanism itself is easily extended; however, in the current setup the final attempt is being made to the originally requested URL (in the example above, "http://pqr/abc.jsp").

One interesting option of the WhaleInvoker and its backtrack mechanism is the configuration of hierarchical classification schemes. The device identification is capable of distinguishing among different vendor and version types of the same base class, e.g., between a "Nokia 7110" and an "Ericsson R320" WAP device. In fact, under certain conditions it is necessary to distinguish and provide alternative JSP responses for those two devices. Hierarchical device classification can be provided, identifying the device as a "wml" device on the top level, a "nokia_wml" device, or a "nokia_7110_wml" device. Correspondingly, the rewriting and backtracking mechanism rewrites the URLs to: "http://pqr/abc.nokia_7110_wml.jsp," "http://pqr/abc.nokia_wml.jsp," and "http://pqr/abc.wml.jsp," in that order, before resorting to the original "http://pqr/abc.jsp." The first URL to hit a real JSP target is then executed and the result returned to the device. Unfortunately a small performance hit is experienced by using the backtracking mechanism, which, however, is easily reduced by ensuring that the JSP tree is indeed complete, maybe simply by providing symbolic links from the originally unused leaf nodes in the JSP tree to the appropriate parent nodes.

In summary, the WhaleInvoker provides an application-independent framework for selecting the correct JSP to process a user request. It allows users to utilize a single URL entry point across a range of devices, and it allows the application providers to differentiate to an arbitrary level their responses to those requests, depending on the device class and type. Mixed differentiation is also possible: for some pages strong differentiation may be required, whereas others are generic enough to be used across a higher-order class of devices, i.e., "all WML devices."

MetaBean. The MetaBean is a Whale-unique component that provides the desired separation of the JSP pages from JavaBeans. It essentially operates as

a dispatcher of property set and get calls from the JSP pages to the actual bean instances. The standard JSP mechanism provides the "use bean" directive, which couples a JSP to one or more JavaBeans, which can then be queried by embedding the appropriate set and get calls to the bean's properties. This coupling is comparatively tight because it implies that any change to the beans' structures also requires changes to the JSP templates that use those beans. Whale inserts a "meta-bean" layer between the JSP pages and the beans in order to alleviate this tight coupling and allows application providers to maintain and upgrade separately either JSP pages or JavaBeans.

Conceptually the MetaBean is a singular bean that exhibits the union of all properties exposed by the set of beans that exist in the logic part of the application. MetaBean getProperty and setProperty calls originating from the JSP pages are redirected to the appropriate concrete application logic bean. This decouples the JSP templates from the beans and improves the level of separation between the two application components. Whale has a built-in tool that automatically generates MetaBeans based on an abstract definition of the properties set and a mapping from those properties to the concrete beans exhibiting those properties. There exists an implementation of the tool that derives both the set of properties and the mapping from the beans themselves by dynamically introspecting each bean class. Every time a get-Property call is executed against the generated MetaBean, the appropriate concrete bean that exhibits that property is located in the mapping table; then the get-Property call is executed against the bean, and the result is returned to the calling JSP. This implies that for every property name there is exactly one concrete bean that is being called to satisfy the "get" requests; it is not uncommon for several beans within such a setup to exhibit the same property. In such situations the tool allows the user to define the "master" for each property name, i.e., to define one particular bean that is responsible for a specific property. If a null value is returned from the master bean, the other beans exhibiting that property name are invoked in an implementation-specific order until the list is either exhausted (in which case a null value is passed to the JSP), or a non-null result value is being received from a concrete bean. Conversely, by convention, the "set-Property" call of a JSP page is passed on to every concrete bean that exhibits that specific property.

The MetaBean approach is particularly effective if the overall application logic is implemented in a large set of individual JavaBeans, each of which implements an essentially "atomic" transaction against the application's back-end servers. This allows each call to start a variable number of such atomic beans and ensures that the server load is finely tuned to the actual data requirements of the incoming request. Larger, more-than-atomic beans are in danger of generating more data than are actually required for a particular data request. The exact split of transaction functionality between the individual beans is, of course, a matter that is particular to every data back-end system. For example, in relational database systems it is cheaper to retrieve an entire row and use a caching mechanism within the bean to return individual columns within that row than to access those columns individually.

Bean tools. Many aspects of a server application and back-end integration are actually recurring problems. As an example, applications frequently need to access data that are stored in relational data systems. Such tasks can be solved by implementing beans that follow a common implementation model and, as a result, can to a large extent be generated automatically. Whale incorporates three simple tools that help application developers in such situations.

BeanBox. The BeanBox is an abstract bean implementation that holds all data structures associated with the common JavaBean mode, such as the names of the individual bean properties and hash tables to store their concrete values. It is also "self-cloning," i.e., it can generate copies of itself on demand, for example for indexed properties. A master instance of a set of cloned BeanBoxes (with special index '0') holds and maintains all references to its clones, and orchestrates all access to those clones. Abstract BeanBoxes can be generated automatically using the BeanTemplateGenerator tool, which generates the boilerplate Java code and placeholders where the implementor manually adds the actual data generation part. BeanBoxes are also the foundation classes used for the SQLBeanGenerator and the XMLBeanGenerator.

In other words, the BeanBox is a generic data structure that exhibits the conventional JavaBean API. Once instantiated with the list of property names, any property exists an "indefinite" number of times and can be accessed by accessing indexed methods on the master instance; the master instance clones itself if previously nonexistent property instances are

being "set." A new instance inherits the settings of the older "clones."

SQLBeanGenerator. Many enterprise data still reside in relational databases. Making such databases accessible from new application domains has been the subject of many efforts in the academic and industrial community, a good overview of which is given in Reference 38.

The SQLBeanGenerator can automatically generate an accessor bean into an SQL-accessible database back end by expanding on the BeanBox type. On invocation of the generator, the database's schema are explored and a bean attribute is generated for each row. For each property, a "get" and "set" method is generated with the following semantics:

On a set-Property call, an SQL selector is narrowed to include the selected property/value pair. On the first get-Property call, an SQL select statement is issued to retrieve the data sets that satisfy the conditions issued in the previous "set" calls. Those set calls are combined to generate the SQL selector statements, which are then executed against the database. As an example, consider a database dbName and a set of rows (or properties) property_1, property_2, . . . property_N and a corresponding set of values (set using setProperty calls) of value_1, value_2, . . ., value_N, then the corresponding SQL statement is:

$$\text{SELECT * FROM dbName}$$
$$\text{WHERE property}_1 = \text{value}_1 \text{ AND property}_2 = \text{value}_2$$
$$\text{AND . . . AND property}_N = \text{value}_N$$

For every column returned from the database, a new clone of the present BeanBox instance is generated, each returned row setting the value of the corresponding property. The requested property is then returned as the result of the get-Property call. Any subsequent get-Property request is answered immediately by exploring the set of columns returned as the result of the previous get-Property call. This result is invalidated when another set-Property call is being encountered. It is worth noting here that the handling of multiple returned columns and rows is an optimization that takes into account that applications rarely request a singular row/column value from a database. There is, hence, a high chance that subsequent get-Property calls from the JSP template will access other data values from that data set. A nonoptimized implementation may execute a slightly different SQL query—namely, if the requested property is $\text{property}_?$:

$$\text{SELECT property}_? \text{ FROM dbName}$$
$$\text{WHERE property}_1 = \text{value}_1 \text{ AND property}_2 = \text{value}_2$$
$$\text{AND . . . AND property}_N = \text{value}_N$$

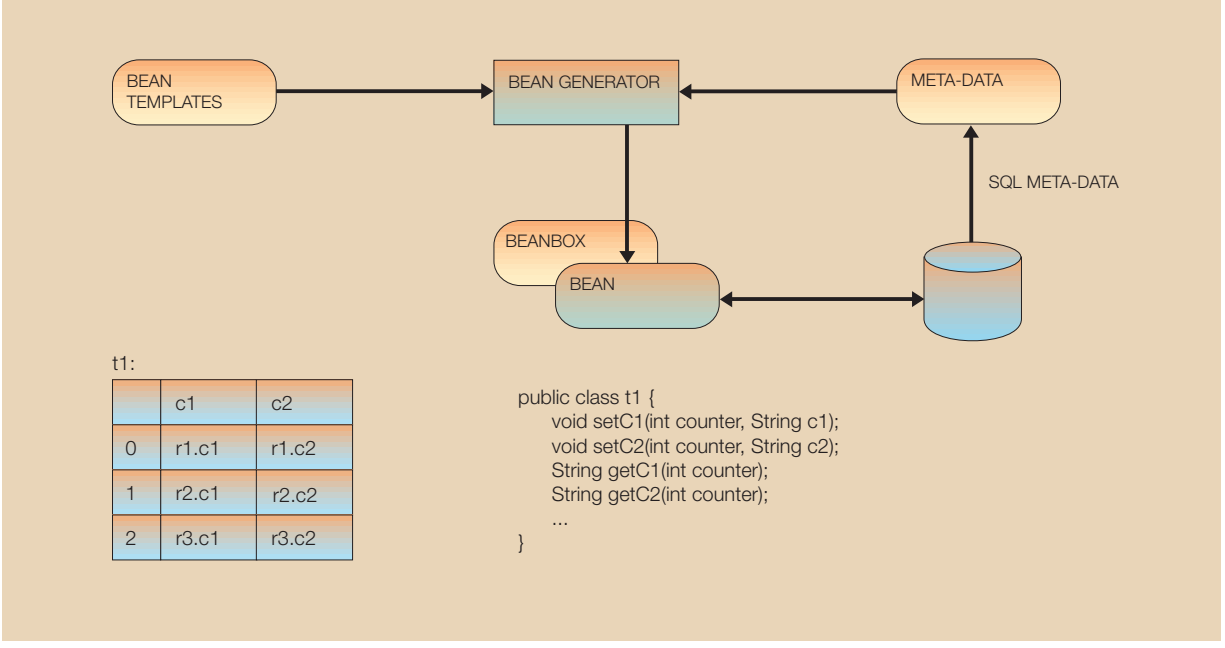
Figures 11 and 12 illustrate the operations of the SQLBeanGenerator and show some sample code segments as generated by the respective generator. The output of the SQLBeanGenerator is an immediately usable Java bean, where simple database queries are sufficient. Currently manual modifications or programming are necessary if the required SQL query has a different format, for example if the logical connectors between the property/value statements are other than simple ANDs.

XMLBeanBox. Another source of data with growing popularity are sites that exhibit XML-type content. Again, the main interest here is in data-centric XML rather than representational XML instances. Unlike relational data, which due to their table-like form are easily converted into a matrix of indexed bean properties, XML data are hierarchical in nature; the hierarchy expresses a containment relationship. On the other hand, for use in JSP pages the beans better limit their API to return string-typed properties. This limitation removes the need to "walk" complex data structures from within JSP pages, which itself would require considerable "logic" to be embedded in those JSP pages.

The approach of flattening the hierarchical data has been driven by the desire to keep the structure of the JSP pages that use the generated beans simpler. Therefore the decision was taken to not reflect the data's hierarchy also in the beans hierarchy, where nested XML elements would correspond to JavaBeans typed properties. Consequently, the hierarchically structured data are mapped onto a flat, tuple-like data model. In addition, this flattening should be performed automatically, i.e., it should be driven by the structural definition of the XML data contained in the DTD associated with the actual data. Especially in this case the generation of such XML accessing beans by hand represents a cumbersome, error-prone and, hence, inefficient method; automatic beans generation is thus highly desirable.

The method to perform this task is based on the following observations:

Figure 11 SQL database access can be accommodated through automatically generated accessor beans. These convert a relational database table into a bean with the following structure: each column is turned into a bean property, each row into a separate instance of the same bean which subsequently becomes accessible as an indexed property.



- The XML data model can be split into "structure" and "information." The structure is given by the nesting of the elements, whereas the information content is contained in attributes and character data.
- It is assumed that a nested element inherits all information content from its containing element. This is particularly true for XML DTDs, which implicitly represent a hierarchical data structure. This observation is used to flatten the XML structure in establishing a "joint" in the relational calculus sense between the nesting element and its nested elements.
- The technique is restricted to XML structures that do not contain mixed content, i.e., an element either contains only character data or only one or several other elements, but not both PCDATA (parsible character data) and other elements.
- A further restriction consists in disallowing recursion in the element structure. That is, in a given XML element E there may not exist another element from which E can be derived: a flattening on such recursive structures would be unbounded.

Formally, let $a_1, a_2, \ldots, a_n$ be the set of attributes of an element E, which also contains other elements $E_1, E_2, \ldots, E_m$, each of which contains only character data (PCDATA). No beans are generated for $E_1, E_2, \ldots, E_m$—instead properties with the element name to represent those elements are created. For E's attributes one property per attribute is created. Thus, the bean corresponding to element E will have properties $a_1, a_2, \ldots, a_n, E_1, E_2, \ldots, E_m$, all of which are of Java-type "String." This algorithm can be applied recursively down the XML hierarchy.

In the case of there being a child node C of E that contains only character data, the element names are concatenated and thus generate a bean called E_C, which has all the properties of E as well as the properties derived from XML attributes and PCDATA subelements of C, using the algorithm described above. Figure 13 illustrates the progression of the flattening algorithm through an XML tree.

This process is essentially equivalent to performing a "join" operation in a relational database system.

Figure 12 Sample code generated by the SQLBeanGenerator, accessing an SQL-based user database

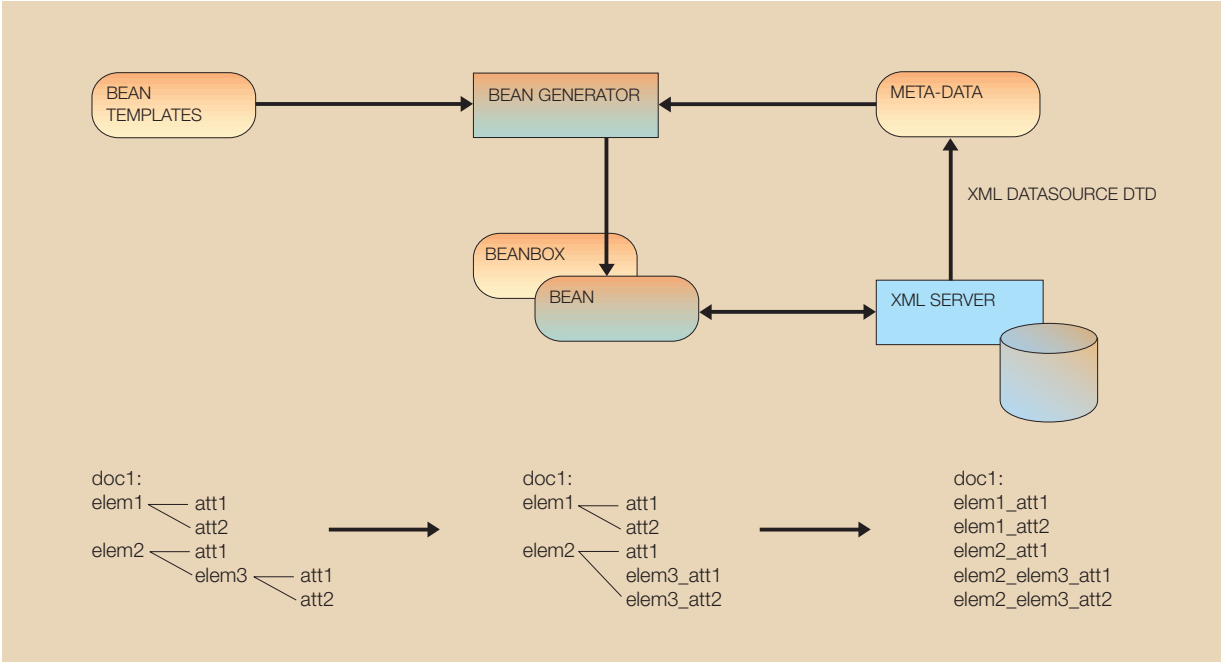
```
// This java file was automatically generated by the "SQLBeanGenerator".
// Do not edit manually.
package ...;
import ...;
public class USER extends SQLBeanBox implements java.io.Serializable {
   public static String[ ] atts = { "FIRST_NAME", "LAST_NAME", "TITLE", "MOBILE_PHONE", "STREET", "CITY",
                                                    "POSTAL_CODE", "COUNTRY", "PWD", "WAPID", "WEBID" };
   public USER() { addAttribute(this, "USER", atts); }
       // for each attribute:
   public void setFIRST_NAME(String v) { setAttribute{"FIRST_NAME", v}; }
   public void setFIRST_NAME(int i, String v) { setAttribute(i, "FIRST_NAME", v); }
   public String getFIRST_NAME() throws Exception { return getAttribute("FIRST_NAME"); }
   public String getFIRST_NAME(int i) throws Exception { return getAttribute(i, "FIRST_NAME"); }
       [...]
       // 'getData()' is called by BeanBox as appropriate, passing the gathered selector as an argument
   public void getData(String constr) throws IOException, InstantiationException, IllegalAccessException,
                                                                    ClassNotFoundException, SQLException {
         // fills in entire population
         ResultSet rs = null;

         // build constrainer
         String constrainer = buildConstrainer();

         // connect to database using driver and access information passed to the SQLBeanGenerator tools at run-time
            Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").newInstance();
            Connection dbConn = DriverManager.getConnection("jdbc:db2:WAPUDB", "username", "password");

         if ( dbConn == null)
            throw new IOException("Failure to connect to DB");

         Statement stmt = dbConn.createStatement();
         rs = stmt.executeQuery("select * from USER" + constrainer);

         ResultSetMetaData md = rs.getMetaData();
         for (int instanceNr = 0; rs.next(); instanceNr++) {
            for (int columnNr = 1; columnNr <= md.getColumnCount(); columnNr++) {
                  String c = md.getColumnName(columnNr);
                  String v = rs.getString(c);
                  setAttribute(instanceNr, c, v};
            }
         }
         dbConn.close();
   }
}
```

All descendants of a given XML element inherit the data properties of the element's attributes and subelements, i.e., the Cartesian product of the properties of E and the properties of C is implicitly being built.

As usual, the properties of the created beans are indexed in order to support repeated occurrences of identically labeled paths through the XML data hierarchy.

The efficiency of this approach is limited by the necessity to create such "join" operations over the XML hierarchy, and thus replicate the data unnecessarily. This stands against the main advantage of automatically deriving a JavaBeans code that is intuitively accessed from within the JSP templates without requiring additional access logic inside the JSP pages, as would be necessary if the hierarchical XML structure is represented directly in the form of a hierarchical beans structure. Figure 14 illustrates this ad-

Figure 13 Outline of the XMLBeanGenerator process while flattening the XML input tree



vantage by showing a JSP fragment accessing a JavaBeans component using the flattening approach against a potential JSP solution using a hierarchical bean structure.

A more complete treatment of the XML flattening algorithm can be found in Reference 39; other descriptions and alternative approaches can be found in References 40 and 41.

Swissair's Easy Check-In

A first commercial deployment of the Whale architecture has recently been put through its paces for Switzerland's national airline. There, Whale is used to host pervasive computing services to Swissair passengers (see Figure 15). The pervasive device of particular user interest in this context is of course the WAP phone, which at the time of the first discussions between IBM Zurich Research and Swissair was just a promising technology, but its popularity has since grown wildly. That is not surprising since cellular telephones are seeing a tremendous market penetration among the airline's most valued (business) passengers. For the airlines it is, therefore, an ideal communications channel to their mobile clientele. The ability to automatically use this communications channel to provide update information from their back-end system and allow passengers direct interactions with those systems is of extremely high interest. On the other hand, the airline industry is in a fortunate position because the vast majority of their business activities and all of the interactions with their customers is conducted and recorded on line; thus, the airlines have gathered tremendous amounts of customer-related data that are available for data mining, and subsequently for customizing and personalizing the pervasive service offerings for each passenger individually.

Yet the selection of the appropriate services for such devices is not straightforward. Simply migrating existing services to WAP is difficult because of the obvious input and output limitations of the receiving device. Whale's strength of providing customized, device-dependent interfaces to existing back-end systems is showcased in this area. Use cases require that many application scenarios be modified and updated. For example, providing ordinary ticket booking capability via WAP has little justification, because there exist other communication channels that are much more adapted to and better suited for this transac-

Figure 14 Sample JSP fragment accessing an XML hierarchy through a simple flattened Java bean (top) and a potential solution using nonflattened bean structures (bottom). Here, it is assumed that a flight-database exists where a property "ALT" is available to indicate the number of available flight segments. Clearly seen is the need to include programming elements and detailed knowledge of the underlying bean data structure.

```
<repeat index=i>
        [...]
        Alternative: <%= i %>
        <% FLIGHT.setALT(i); %>
        <repeat index=j>
                From: <%= FLIGHT_SEGMENT.getFROM(j) %>
                To:     <%= FLIGHT_SEGMENT.getTO(j) %>
        </repeat>
        [...]
</repeat>

<repeat index=i>
        [...]
        <% SEGMENT seg = FLIGHT.getSEGMENT(i); %>
        <repeat index=j>
                From:
                <insert bean=seg property=FROM(j)>
                </insert>
                To:
                <insert bean=seg property=TO(j)>
                </insert>
        </repeat>
        [...]
</repeat>
```

tion—in particular the airline's call centers, travel agents, or the Web channel. The application service provider must therefore define an appropriate service that makes the best use of the device characteristics as well as the usage paradigms of the intended end-user device, and define the correct integration scenario for that service into the existing offerings that utilize other communication channels. Here, Whale's ability to integrate different channels against a variety of back-end systems is of interest.

The application selected by the customer is a check-in service. The check-in transaction, usually executed by the traveler on arrival at an airport, serves two primary purposes: it registers the traveler with a particular outgoing flight and assigns a seat number. A boarding pass is issued to the traveler. It also flags the passenger's entry in the airline's reservation system and feeds that information further to the airline's revenue and availability applications. Enabling passengers to execute this transaction through the WAP phone has a positive impact on the passenger as well as the airline: once the passenger has checked in, the airline can use the WAP channel to continuously update the flight status. If the flight is delayed, the user is informed immediately. The passenger is also informed if his or her check-in status changes (for example if a standby passenger is granted a seat). Having such up-to-date information and needing no manual check-in at the airport (which could involve queuing at the counter and therefore an unpredictable holdup) enables the passenger to better schedule a trip to the airport and limits waiting time once in the terminal. For the airline, having passengers checking in early allows better capacity and availability planning for the outgoing aircraft and, as a secondary step, reduces the burden on the airline staff in the terminal. In addition, check-in is a transaction that can be executed with minimal input and is thus a good fit for the device capabilities.

The WAP-based check-in solution interacts with a related project, called FastTrack, involving Swissair, IBM, and other partners. FastTrack is based on so-called radio-frequency-tagged cards (RF cards) that

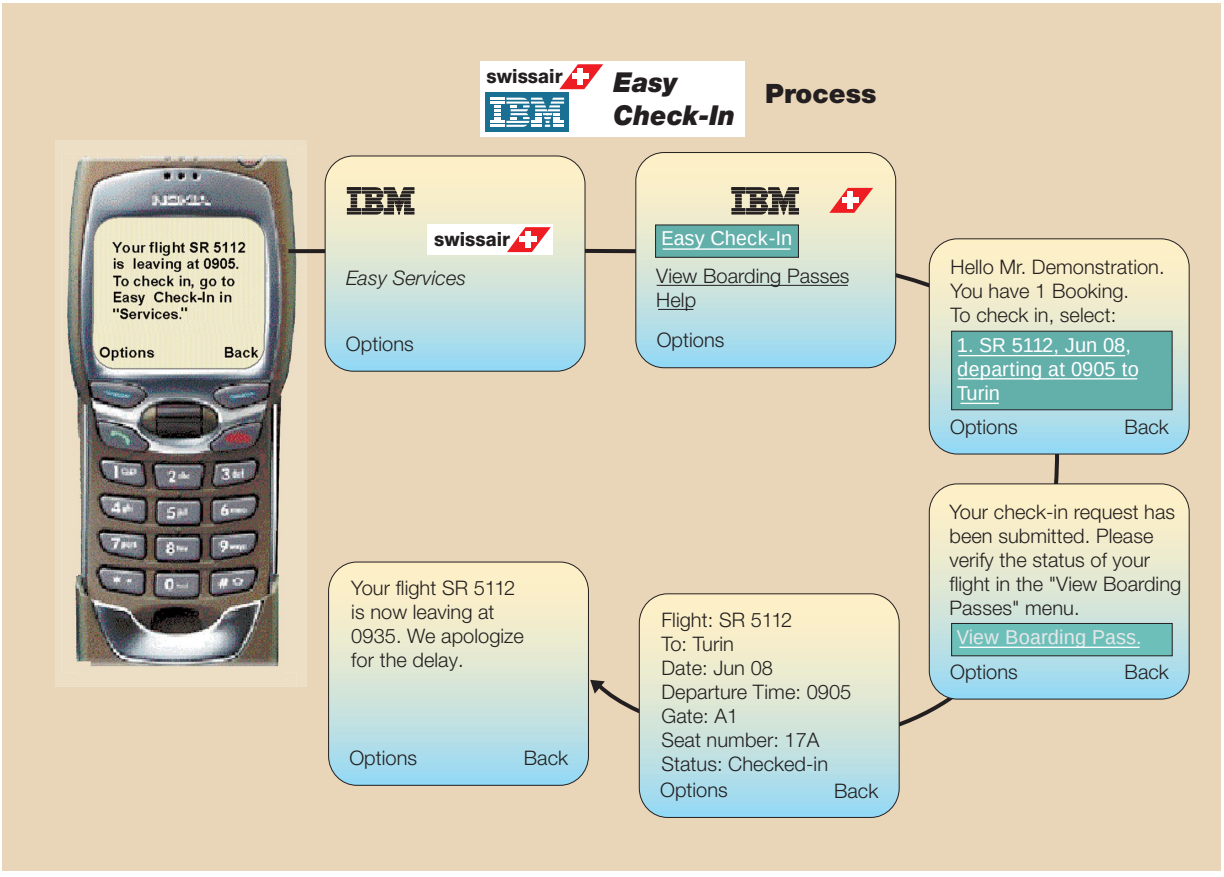
Figure 15 The IBM/Swissair WAP services home page, allowing passengers to check in to Swissair flights, accessed from a Nokia 7110 WAP device

are used in Swissair's frequent-flyer customer loyalty program. In addition to the conventional magnetic stripe, all cards issued by Swissair to its passengers are also equipped with a passive radio transmitter that, once carried through a magnetic field of specified strength and frequency, emits a radio signal that can be encoded with the owner's frequent flyer number. If the passenger agrees, the number is encoded on the card and the passenger's movements within the terminal are traced through corresponding receiver gates that are located at strategic locations in the building. The FastTrack infrastructure itself is interesting and provides the basis for a number of services for the airline and the passenger. For brevity this paper is only concerned with its role as a complement to the WAP-based and Whale-hosted check-in service.

The usage model for the WAP-based check-in is simple (see Figure 16): Using the mobile phone, the user selects the WAP service, which immediately establishes a connection to Whale and the Swissair back end. The user is identified and greeted; a list of available flights is presented. This list will of course contain the flights for which the passenger has a booking, but may also contain alternative flights to which the passenger is free to check in. High-fare passengers are usually allowed to check in to alternative flights at will, and they may do so because they are early for their booked flight and have a valid earlier flight at their disposal. Upon selection of the desired flight, the check-in is executed and an "electronic boarding pass" is generated and rendered to the user device. All further steps such as passport control and boarding are controlled using the FastTrack system. If any of the boarding pass information changes, an appropriate update message is transmitted and displayed to the user. The entire check-in process itself can be triggered either by the user, or on arrival of the passenger at the airport if the network operator has the technical capability to provide such information and suitable agreements exist among the user, the network operator, and the application service provider (i.e., Swissair in this case). In addition, the currently operational setup also generates a reminder message for the passenger two hours prior to departure of a booked flight.

Implementing this seemingly simple application reveals some lesser-known areas of issues and concerns with such pervasive computing applications. From a business process engineering point of view, implementing such applications is nothing short of a "palace revolution." The business processes in place at the airport have remained essentially unchanged since the early 1970s. This has had an effect on the IT back-end systems that supported those processes inasmuch as they have, over the years, been optimized and continuously streamlined to support exactly those processes. In the case of a check-in, the existing process involves a check-in staff member who enters the passenger's booking data from the paper ticket, verifies these data against the data stored in the airline's reservation system, and finally executes the check-in transaction on the airline's departure control system (DCS). The DCS is preloaded nightly with the details of all departing passengers and flights for the next 24 hours through what is essentially a replication process of the reservation system. Both the DCS and the reservation system are, however, indexed exclusively by flight numbers, not passenger names. Hence, it is not possible to generate the list of booked flights for a given passenger name or passenger frequent flyer number. Easy Check-In must rely on yet another intermediate database, which is continuously replicated out of the DCS or the res-

Figure 16 The Swissair Easy Check-In process

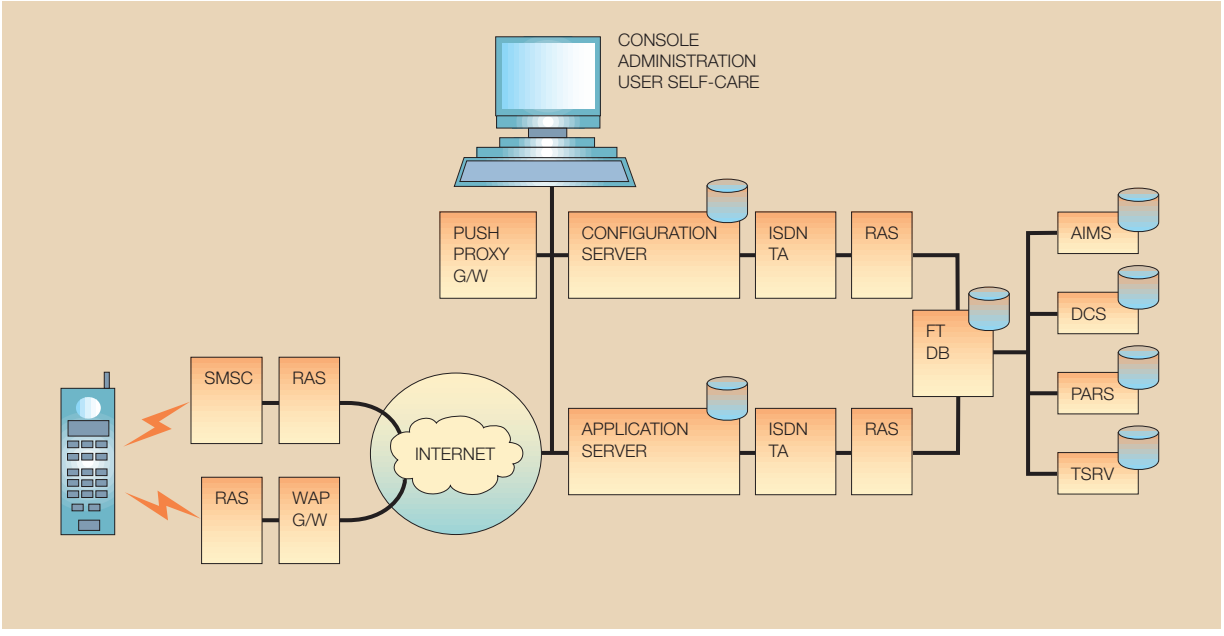


ervation system and generates an additional index on the booking information using the frequent flyer identifier (ID). Integrating alternative flight offerings requires the integration of additional back-end systems. Pervasive computing initiatives in other industries will probably face similar challenges, namely the need to change business processes that have left their footprint throughout the IT infrastructure.

In addition to the actual application and the delivery path associated with it, a further element that had to be put in place as part of this service offering is the capability to subscribe users, manage their personal data, and configure their user devices accordingly. These functions have been implemented and are hosted as additional applications on Whale. They are accessible through an HTML interface and are thus easily integrated into Swissair's existing call center IT infrastructure. Using the interfaces provided, the Swissair call center staff can access a simple Web page and enter the personal details of a new subscriber. On the press of a button, the back-end infrastructure is updated and a personalized configuration set is generated and transmitted to the user device. Within less than 30 seconds, the configuration set is delivered by SMS (short messages system) and installed automatically on the user's device, and the user is ready to access the service. The complete architecture of the solution put in place for Swissair can be seen in Figure 17.

Swissair's Easy Check-In went on line on December 16, 1999. Swissair issued an initial batch of 150 WAP-capable telephones to selected frequent flyers. The user group remained constant at this level for roughly the first three months of operation. During this time, valuable usage statistics (see Figure 18)

Figure 17 IBM/Swissair Easy Check-In service architecture, showing the complex Swissair database system on the right: the Airport Information Management System (AIMS) stores gate information, the Departure Control System (DCS) processes the actual check-in, the Reservation System (PARS) gathers booking information, and the Tracking Server (TSRV) tracks passengers within the airport. Also shown are the data delivery channels for the application itself (via Application Server, WAP gateway, and RAS), as well as the SMS-based push channel (via Application Server, RAS, and SMSC), and the provisioning circuits for subscriber and device management (via Configuration Server and Push Proxy gateway). Both the application server and the configuration server are based on the Whale architecture.



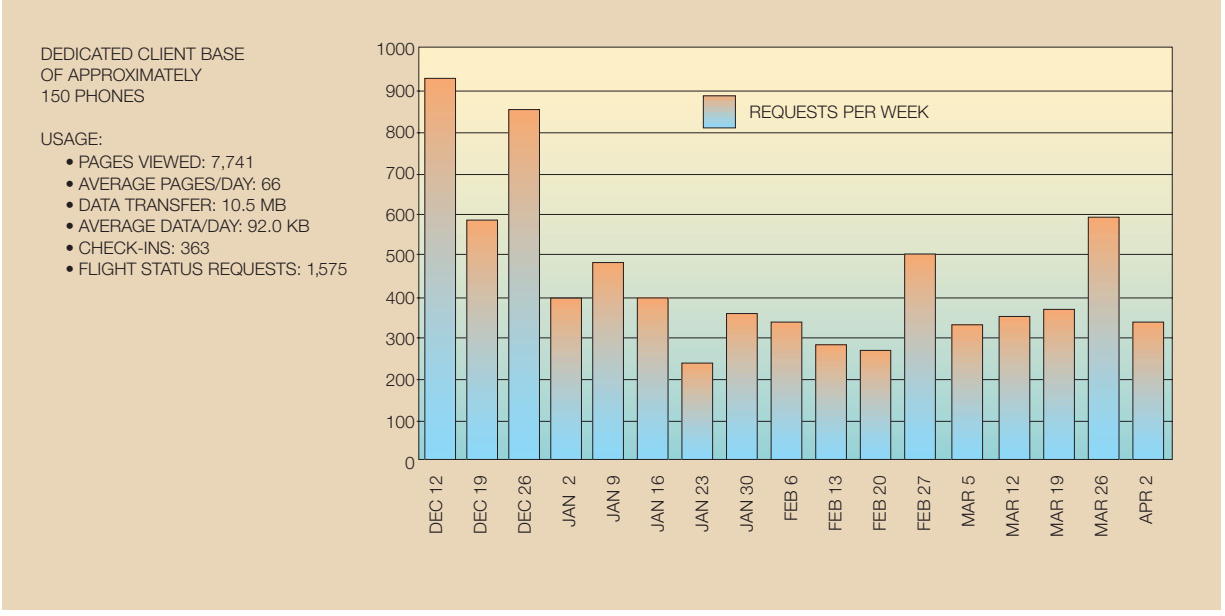
have been gathered, which will help predict the resources required for larger roll-outs.

Conclusions

Whale, a Web application server architecture developed at the IBM Zurich Research Laboratory and deployed at present in a high-profile customer engagement, provides a framework for application service programmers and providers that enforces a clean separation between the application logic and the markup generation. This is necessary because of the growing variety of end-user devices that require device-specific content due to different markup languages, a diverging set of device features that need to be integrated into the application, and a different usage and interaction model. By separating the logic components of an application from the front end in a thorough manner, additional devices can be added easily by supplying an additional set of markup generation components, without requiring modification to other parts of the application. Conversely, migrating an existing application to a modified back-end system with the same functional characteristics only affects the logic components and does not require changes to the markup generation. This requires that the transactions and services provided by the back-end system be completely and cleanly exhibited in the form of beans within the application logic components of the entire application, and that this set of beans be as fine in granularity as possible.

There is very little need to use existing Web back ends, in particular HTML, for data sources. Typically, service provisioning happens by or in collaboration with the owner of the content, and more suitable access mechanisms to the content source are usually available. Whale has tool support to generate beans automatically for a number of standard application scenarios, and tools to automatically generate the bridge between these application components and

Figure 18 Preliminary usage statistics for the Swissair Easy Check-In after 100 days of continuous service. Taking into account the nature of the application and the limited number of end-user devices, the number of executed check-in transactions is unexpectedly high and reflects the great interest in this service. This is further evident in the comparatively stable usage exhibited in these statistics.



the set of JSP templates that represent the user interface to those application logic components.

Support for additional devices requires an additional JSP template, which can be authored using one of the growing set of WYSIWYG JSP authoring tools. Many application service providers believe that the above requirement is not prohibitive: Given the extremely limited user interface "real estate" of many pervasive computing devices, it is instrumental for the application service providers to exactly and tightly control the look-and-feel of every single application element. In particular, a semiautomatic translation even of image data (e.g., the application service provider's company logo) had to be dropped in favor of custom-designed versions for various receiving devices. Controlling the corporate identity and its representation on the customer devices here seems to rank above ease and speed of implementation.

Acknowledgments

Thanks are due to our colleagues at IBM Zurich Research, especially Francois Dolivo and Philippe Janson, to the anonymous reviewers, to the editorial staff of the *IBM Systems Journal* for their help in improving the final version of this paper, and to Swissair for giving us the opportunity to apply our technology in a real-world setting.

Cited references

1. D. Raggett, A. Le Hors, and I. Jacobs, "HTML 4.01 Specification," W3C (December 1999), http://www.w3.org/TR/REC-html40/.
2. *ISO 8879:1986. Information Processing—Text and Office Systems—Standard Generalized Markup Language (SGML)*, International Organization for Standardization, Geneva/New York (1986).
3. W3C, "Voice Browser," http://www.w3.org/Voice.
4. M. Campione and K. Walrath, *The Java Tutorial—Programming for the Internet*, *Second Edition*, Addison-Wesley Publishing Co., Reading, MA (1999).
5. J. Postel, *Transmission Control Protocol*, RFC 793, IETF Network Working Group (September 1981), http://www.ietf.org/rfc/rfc0793.txt.
6. WAP Forum, "WAP Forum Specifications," http://www.wapforum.org/what/technical.htm.

7. R. Braden, "T/TCP—TCP Extensions for Transactions," RFC 1644, IETF Network Working Group (July 1994), http://www.ietf.org/rfc/rfc1644.txt.
8. NTT DoCoMo, "What Is i-mode?" http://www.nttdocomo.com/i/index.html.
9. T. Kamada, "Compact HTML for Small Information Appliances," W3C (February 1998), http://www.w3.org/TR/1998/NOTE-compactHTML-19980209.
10. WAP Forum, "Wireless Application Protocol: Wireless Markup Language Specification" (August 1999), http://www.wapforum.org.
11. T. Bray, J. Paoli, and C. M. Sperberg-McQueen, "Extensible Markup Language (XML) 1.0," W3C (February 1998), http://www.w3.org/TR/1998/REC-xml-19980210.
12. T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext Transfer Protocol—HTTP/1.0," RFC 1945, IETF Network Working Group (May 1996), http://www.ietf.org/rfc/rfc1945.txt.
13. WAP Forum, "Wireless Application Protocol: Wireless Session Protocol Specification" (August 1999), http://www.wapforum.org.
14. L. Gilman and R. Schreiber, *Distributed Computing with IBM MQSeries*, John Wiley & Sons, Inc. (October 1996).
15. M. Debusmann, R. Kruger, and C. Weyer, "Towards an Automated Management of Distributed Applications," *Proceedings of the IFIP International Working Conference on Distributed Applications and Interoperability Systems* (DAIS'97), Cottbus, Germany (October 1997).
16. Sun Microsystems, "RMI—Remote Method Invocation," http://java.sun.com/products/jdk/1.1/docs/guide.rmi.
17. J. D. Davidson and S. Ahmed, "Java Servlet API Specification," Sun Microsystems (November 1998) http://www.javasoft.com/products/servlet/index.html.
18. S. Mazzochi, "The Java Apache Project Information System" (December 1998), http://java.apache.org.
19. IBM, "WebSphere Application Server," http://www-4.ibm.com/software/webservers/appserv.
20. C. J. Date and H. Darwen, *A Guide to the SQL Standard*, Third Edition, Addison-Wesley Publishing Co., Reading, MA (1993).
21. V. Apparao et al., "Document Object Model (DOM), Level 1 Specification," W3C (October 1998), http://www.w3c.org/TR/1998/REC-DOM-Level-1-19981001.
22. S. Hild, "Service Provisioning for the Wireless Application Protocol," Technical Report RZ3130, IBM Zurich Research Laboratory (March 1999).
23. Travelocity Travel Service home page, http://www.travelocity.com.
24. The Internet Engineering Task Force, http://www.ietf.org.
25. The World Wide Web Consortium (W3C), http://www.w3.org.
26. World Wide Web Consortium (W3C), "Extensible Stylesheet Language (XSL)," http://www.w3.org/Style/XSL.
27. J. Clark, "XSL Transformations (XSLT)," W3C (November 1999), http://www.w3.org/TR/xslt.
28. E. Pelegri-Llopart, L. Cable, and S. Ahmed, "JavaServer Pages Specification," Sun Microsystems (November 1999), http://www.javasoft.com/products/jsp/download.html.
29. Sun Microsystems, "JavaBeans" (July 1997), http://www.javasoft.com/beans/doc/spec.html.
30. D. Pozefsky, R. Turner, A. K. Edwards, S. Sarkar, J. Mathew, G. Bollella, K. Tracey, D. Poirier, J. Fetvedt, W. S. Hobgood, W. A. Doeringer, and D. Dykeman, "Multiprotocol Transport Networking: Eliminating Application Dependencies on Communications Protocols," *IBM Systems Journal* **34**, No. 3, 472–500 (1995).
31. C. Rigney, A. Rubens, W. Simpson, and S. Willens, "Remote Authentication Dial-In User Services (RADIUS)," RFC 2138, IETF Network Working Group (April 1997), http://www.ietf.org/rfc/rfc2138.txt.
32. Kocher et al., "The SSL Protocol Version 3.0," Internet Draft, Internet Engineering Task Force (November 1996).
33. P.-C. Cheng, J. A. Garay, A. Herzberg, and H. Krawczyk, "A Security Architecture for the Internet Protocol," *IBM Systems Journal* **37**, No. 1, 42–60 (1998).
34. R. Leins et al., "Tivoli Storage Manager Version 3.7: Technical Guide," IBM Redbooks, SG24-5477-00, IBM Corporation (December 1999).
35. R. Franklin, J. Hjelm, S. Dawkins, and S. Singhal, "Composite Capability/Preference Profile (CC/PP): A User Side Framework for Content Negotiation," W3C (July 1999), http://www.w3.org/TR/NOTE-CCPP.
36. WAP Forum, "WAP User Agent Profiling Specification" (August 1999), http://www.wapforum.org.
37. D. Brickley and R. V. Guha, "Resource Description Framework (RDF) Schema Specification 1.0," W3C (March 2000), http://www.w3.org/TR/2000/CR-rdf-schema-20000327.
38. J. M. Cheng, N. M. Mattos, D. D. Chamberlin, and L. G. DeMichiel, "Extending Relational Database Technology for New Applications," *IBM Systems Journal* **33**, No. 2, 264–279 (1994).
39. C. Binding, D. Bourges-Waldegg, and S. Hild, "Generation of XML Data Accessing Java Beans," *Proceedings of the First International Conference on Web Information System Engineering*, Hong Kong (June 2000).
40. D. Bourges-Waldegg, C. Binding, and S. Hild, "Bringing Legacy Data to Pervasive Devices," IBM Pervasive Computing Conference, Singapore (November 1999).
41. M. Reinhold, "An XML Data-Binding Facility for the Java Platform," Sun Microsystems (July 1999), http://java.sun.com/xml/white-papers.html.

Accepted for publication September 10, 2000.

Stefan G. Hild *IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland (electronic mail: sgh@zurich.ibm.com).* Dr. Hild is a research staff member in the Mobile Internet research group of the IBM Zurich Research Laboratory. He holds doctorate and undergraduate degrees in computer science from the University of Cambridge, England, and the University of London, respectively. Before he joined the IBM Research Division in 1997 he held positions at the IBM Scientific Center, Heidelberg, Germany, and at the IBM Hursley Development Laboratory, England.

Carl Binding *IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland (electronic mail: cbd@zurich.ibm.com).* Dr. Binding is a research staff member in the Mobile Internet research group of the IBM Zurich Research Laboratory. He holds a doctorate in computer science from the University of Washington, Seattle, and a diploma of electrical engineering from the Eidgenössische Technische Hochschule in Zurich, Switzerland. He previously held positions with Olivetti Research Center and Union Bank of Switzerland.

Daniela Bourges-Waldegg *IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland (electronic mail: dbw@zurich.ibm.com).* Dr. Bourges-Waldegg has been a research staff member in the Mobile Internet group of the IBM Zurich Research Laboratory since January 1999. She holds Ph.D. and M.S. degrees in computer science from Université de

Rennes 1, France, and a B.S. degree in electrical engineering from the Metropolitan Autonomous University of Mexico City.

Céline Steenkeste *IBM Global Services, EMEA, WEST, BIS, Telecommunication Sector Immeuble Zeus, Bercy 40, Avenue des Terroirs de France, 75611 Paris 12, France (electronic mail: csteenkeste@fr.ibm.com).* Ms. Steenkeste is an IT professional in the Business Innovation Services branch of IBM Global Services in France. She holds an engineering diploma from Telecom INT (Institut National des Telecommunications, Evry, France) and a certification in communication systems and multimedia communications from the Eurecom Institute (Sophia Antipolis, France). She completed her professional thesis requirements at the IBM Zurich Research Laboratory during the spring semester, 2000.