Chapter One

Portals and Portlets: The Basics

Without a portal to plug into, portlets by themselves are quite useless. Put another way, just as you put letters together to form words and put words together to create sentences, you put portlets together to create one portal page. To understand what portlets are and where they come into play, we must therefore look at what a portal is. To do otherwise would be like trying to explain why letters are such a great thing without mentioning that you can combine them together to create words and sentences.

In this chapter, we briefly consider the different types of portals that exist and how portlets fit into the picture. Then, we take a closer look at the runtime environment of portlets: the portlet container. Just as the servlet container provides the infrastructure for running servlet components in the servlet world, so the portlet container provides the infrastructure for running portlets.

To get started, let's first look at what a portal is and explore the key benefit portals offer to the user — namely, the integration of several independent applications on one screen.

What Is a Portal?

Figure 1.1 depicts a typical portal page. This portal consists of several pages, including Welcome, My Workplace, and My Finances. The figure shows the My Finances page of the portal, on which five portlets interact with each other.

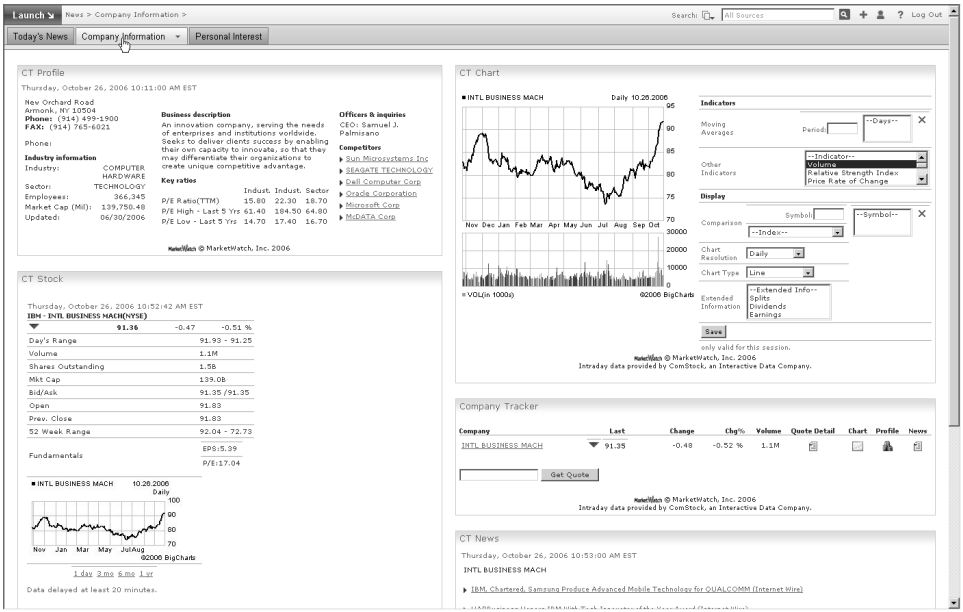

Figure 1.1: Sample portal page

At the top of the portal page, a navigation bar lets the user move between portal pages. On the upper-right side, administration links let the user log in or out of the portal, access the user profile, and get help. Immediately below this upper bar, two finance-related portlets appear below:

- The CT Profile portlet displays details about a specific company.
- The CT Chart portlet displays charts and graphs relevant to the company.

These portlets, and the others you see in the figure, are placed on the page and represent applications with which the user can interact. These different applications are now integrated onto one page with one consistent API and are managed centrally. This approach offers advantages for the user, who now can access different applications consistently and in one place, and for the portal administrator, who can manage access to back-end services for a specific user in one place. If we examine this example more closely, you can see what a portal really offers. Portals enable users, even when they're using the Web, to work more as they do on their desktop, with different applications on one screen that can be interconnected and exchange data.

Most readers of this book probably have an understanding of what a portal is, and specifically of what WebSphere Portal Server is and how it generally works, but for completeness we should go over some obligatory information. If you're unfamiliar with WebSphere Portal, the references at the end of this chapter will point you to more information.

Types of Portals

We encounter the concept of portals throughout our daily lives. Newspapers, magazines, company bulletin boards, and car dashboards are all examples of portals we take for granted every day. Newspapers and magazines are general portals into world events, sports scores, stock market prices, and local happenings. Company bulletin boards are tailored, providing a look at the world through the eyes of the company — for example, by displaying the cafeteria's current lunch menu, clippings from the latest CEO memo, and workplace safety information. Each posting on the bulletin board is a view into the company. Automobile dashboards offer us a very specific portal into the inner workings of the different systems of our cars. The speedometer, oil pressure gauge, temperature gauge, and odometer all provide crucial information about the current condition of our vehicle's systems.

Web pages, too, provide a view of some type of information. With the rise of the World Wide Web, sites such as Yahoo and Excite started turning up as some of the first examples of portals on the Internet. These portals give users access to the latest news, weather, and stock reports. Some current Internet portals let you make stock trades or book flights and hotel rooms. They act as a single point of entry into applications and information on the Internet or your intranet. Today's portal combines applications and information into a consistent, single user interface (UI). It also supports user customization, personalization, and single sign-on.

IBM's WebSphere Portal product, shown in Figure 1.2, is essentially a framework for building portals. WebSphere Portal is capable of creating portals that are more general (like magazines and newspapers), tailored (like a company bulletin board), or very specific (like a car dashboard). You can use WebSphere Portal to create portals you can access from a desktop computer, your cell phone display, a PDA, or even your voice telephone. WebSphere Portal enables this functionality by combining portlets on a single Web page to give the end user access to information.

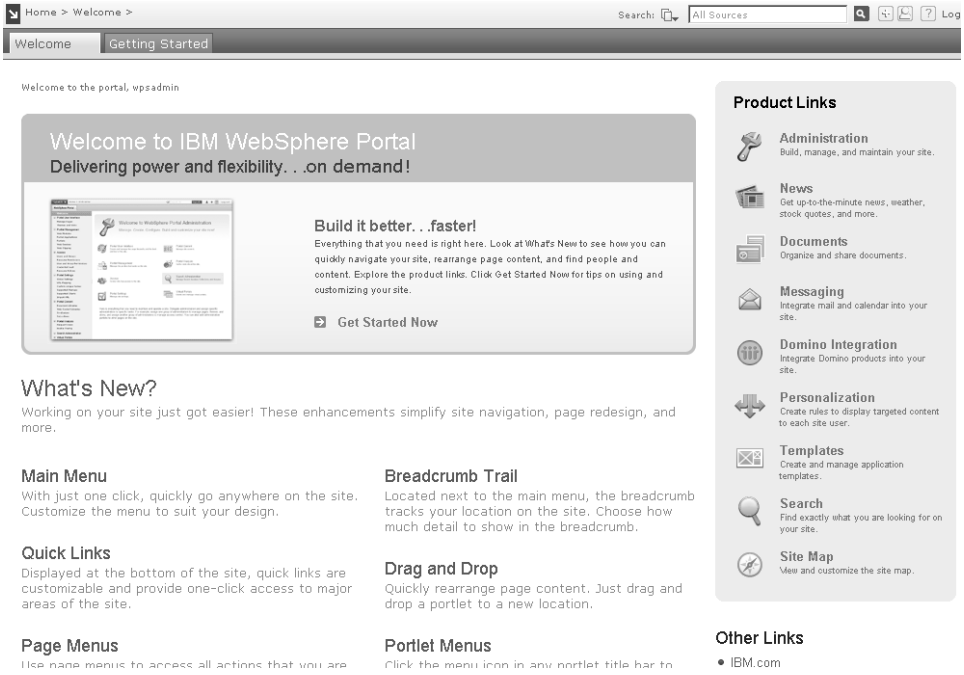

Figure 1.2: WebSphere Portal

Although we tend to imagine a browser interface when we think of WebSphere Portal, the product is really a set of complementary products that all combine to provide a dizzying array of function. These individual products that combine to create the portal include IBM DB2 Universal Database, IBM Tivoli Directory Server, and IBM WebSphere Application Server, as well as development tools, collaboration components, and, of course, IBM WebSphere Portal Server. To learn more about the various options of the product, visit *http://www.ibm.com/software/genservers/portal*.

What Does the Specification Say About Portals?

So how shall we define a portal? Many of the definitions available center on aggregation and integration. We'll take the following definition from the Java Portlet Specification V1.0 (Reference 1):

> "A portal is a web based application that — commonly — provides personalization, single sign on, content aggregation from different sources and hosts the presentation layer of Information Systems. Aggregation is the action of integrating content from different sources within a web page. A portal may have sophisticated personalization features to provide customized content to users. Portal pages may have different set of portlets creating content for different users."

In the next sections, we shed some more light on the different parts of the portal definition that all center on the main portal theme: application integration.

PORTAL APPLICATIONS AND PORTLETS

A *portal application* is a group of portlets that form a logically associated group. Portlets in an application are installed as a single package. When programmed appropriately, they're able to communicate with one another by sending and receiving messages.

Because WebSphere portal is the framework, you can think of portlets as the pieces of art we place into the frames created by WebSphere Portal. A portlet could be an article, as in a magazine. It could be column, as in a newspaper. A portlet could be the cafeteria menu or a speedometer. A portlet is one small piece of an overall portal. It is one element of many that could appear on the user's screen. In this book, we cover in detail how to create these portlets, and we look at various facilities WebSphere Portal offers for portlets. In addition, in the second half of the book, we discuss composite applications and some of the approaches for designing and developing them.

PORTAL ARCHITECTURE

Portlets are run by a component, called a *portlet container*, that provides the portlet with the required runtime environment. The portlet container manages the life cycle of all the portlets and provides persistent storage mechanisms for the portlet preferences, letting portlets produce user-dependent markup. The portlet container passes requests from the portal on to the hosted portlets. It doesn't

aggregate the content produced by the portlets; that's the portal's job. Figure 1.3 depicts the overall portal architecture.

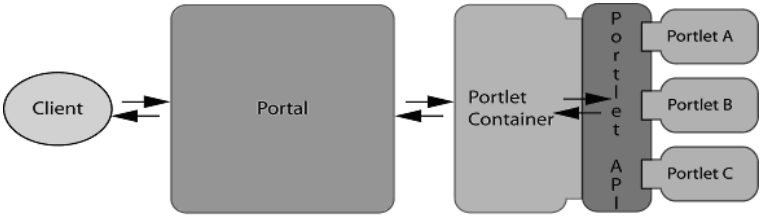

Figure 1.3: Portal architecture, with the portal aggregating the content and the portlet container running the portlets

Here's how it works:

1. A registered user (client) opens the portal, and the portal application receives the client request and retrieves the current user's page data from the portal database.
2. The portal application then issues calls to the portlet container for all portlets on the current page.
3. The portlet container, which holds the user's preferences, calls the portlets via the portlet API, requesting the markup fragment from each portlet and returning the fragment to the portal.
4. The portal aggregates all markup fragments together into one page, which the portal finally returns to the client/user, giving the user the integrated, useful interface he or she is used to on the desktop.

You'll learn more details about the portal architecture in later portions of this book as we explain the components and their subcomponents in more detail. For now, it should be enough to keep in mind that there are the two major components: the portal itself and the portlet container. Now that you know this much about portals, let's take a closer look at what portlets really are and what their role is in the big picture.

And What Is a Portlet?

Now that you know how a portal functions, it's time to take a closer look at portlets and their role in this environment. Let's start with a definition.

A *portlet* is a Java-based Web component that processes requests from a portlet container and generates dynamic content. The content generated by a portlet is called a *fragment*, which is a piece of markup (e.g., HTML, WML, XHTML) adhering to certain rules. A fragment can be aggregated with other fragments to form a complete document, called the *portal page*.

One could ask why portlets were invented and specified in the Java Portlet Specification at all. Why were existing J2EE concepts (namely the servlet) not enough? As you've already seen, that would lead to challenges in creating a consistent user experience. But what else is there that justifies creating a new component? Table 1.1 lists the reasons we think portlets are a separate component.

Table 1.1: Portlets vs. servlets as portal components

Servlets	Portlets
Web clients interact directly.	Web clients interact with portal. Portal acts as mediator, provides infrastructure.
Each servlet assumes it is the only responding component and produces a complete document.	Portlets assume other portlets are responding to the portal's request and produce markup fragments. Portal coordinates response to client, handles character set encoding, content type, and setting of HTTP headers.
Directly bound to a URL.	Addressed only via portal.
Less-refined request handling.	Request handling includes action processing and rendering options.
	Portlets have predefined modes and window states that indicate the function the portlet is performing and the amount of real estate in the portal page available to the portlet.
	Numerous portlets exist on a portal page and therefore require concepts such as the portlet window and portlet entity.

Table 1.1: Portlets vs. servlets as portal components (continued)

Servlets	Portlets
	Portlets need means for accessing and storing persistent configuration and customization data on a per-user basis. Because portlets need to be plugged into an existing portal, these storage functions must be provided by the portal infrastructure for the portlet and thus need to show up in the portlet API.
	Portlets need access to user profile information to generate user-specific output.
	Because portlets are plugged into portal systems, they need URL rewriting functions for creating hyperlinks within their content, letting URL links and actions in page fragments be created independently of the specific portal server implementation.

All these requirements made it a cleaner choice to introduce a new component, portlets, instead of bending the servlet definition to also fulfill these requirements. However, the Java Portlet Specification is closely aligned with J2EE concepts to permit the reuse of as much of the existing J2EE infrastructure as possible. The following points reflect this close alignment:

- Portlet applications are packaged as WAR files, with an additional portlet deployment descriptor (portlet.xml) for the portlet component, and can be deployed using the existing J2EE Web application infrastructure for WAR files.
- Portlet applications reuse the standard HttpSession; thus, portlets can share data via the session with other J2EE artifacts, such as servlets and JavaServer Pages (JSPs).
- Portlets can access the Web application context via the portlet API and share data with other J2EE artifacts on the context level.
- Portlets can access Web application initialization parameters defined in the web.xml file via the portlet context.

- Portlets can include servlets and JSPs via a request dispatcher.
- Portlet J2EE roles defined in the portlet.xml file can reference J2EE roles defined in web.xml, enabling a unified role mapping between portlets and servlets.

To leverage the existing J2EE infrastructure for portlets today, we can wrap them as servlets and deploy them in the Web container with the portlet container running on top of the Web container (see Figure 1.4). The Pluto Java Specification Request (JSR) 168 portlet reference implementation as well as the Jetspeed portal both take this approach.

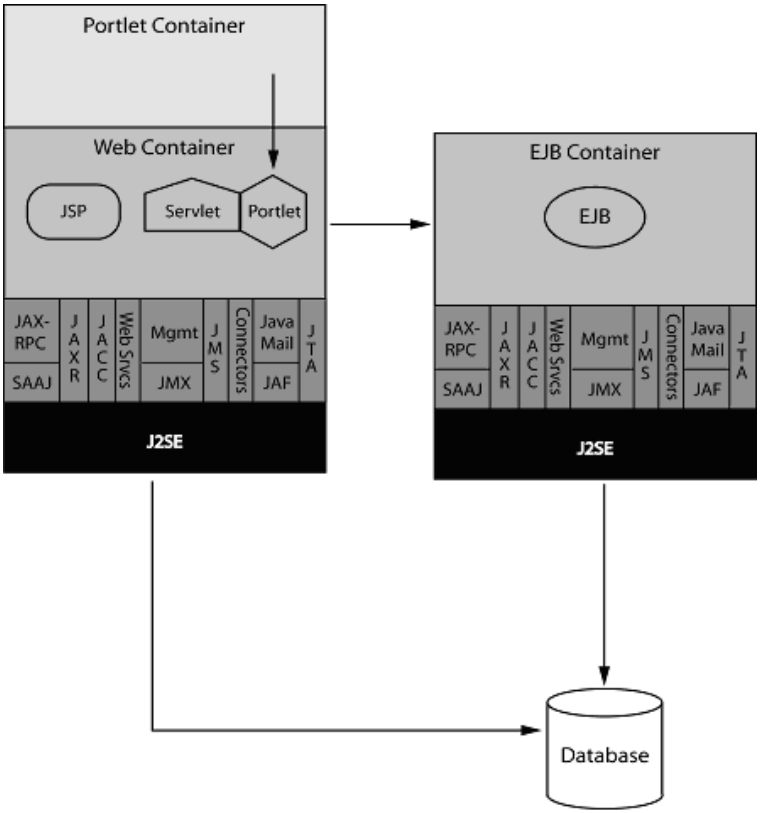

Figure 1.4: How portlets relate to J2EE

Today, while using as many J2EE concepts as possible, portlets aren't an integral part of J2EE. Thus the portlet container is running on top of the servlet container.

The plan is to keep future portlet specification aligned with the next J2EE versions. The goal is to integrate the Portlet Specification into J2EE in the future. This integration would permit treating portlets as first-class J2EE citizens, and the whole J2EE infrastructure, including application management, monitoring, deployment, and authorization, would support them. When this happens, a portlet container will be part of the application server and leverage the server's entire infrastructure, including administration and performance tuning. Portals will then be Web applications running on the application server and leveraging the different containers provided by the application server. Figure 1.5 shows the portlet container as an independent container beside the servlet and Enterprise JavaBeans (EJB) container. For more information about the J2EE specification, consult Reference 2.

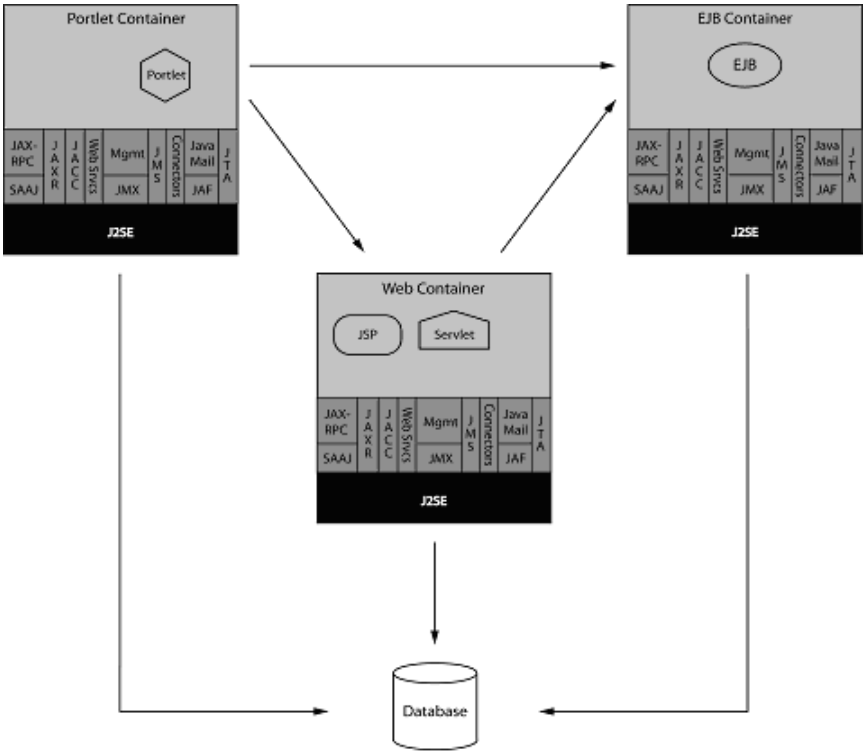

Figure 1.5: Future scenario in which portlets are part of J2EE and the portlet container is another J2EE container, like the servlet and EJB containers

The Java Portlet Specification is also aligned with another upcoming important J2EE technology, JavaServer Faces (JSF), which enables server-side user interfaces for Web components. For more information about JSF and portlets, see Chapter 6.

Portlets in Practice

What does our definition of portlets mean in practice? You've already seen a real-life portal page in Figure 1.1. Figure 1.6 depicts the basic structure of such a portal page.

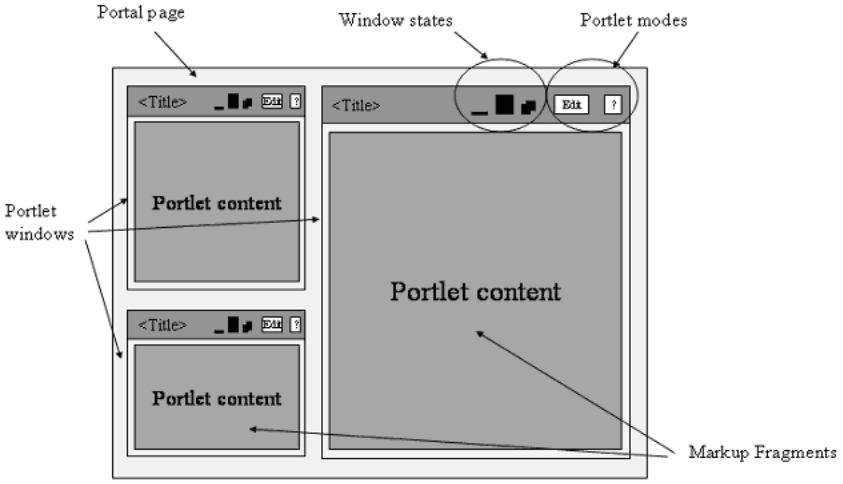

Figure 1.6: Portlet windows on a portal page

The markup fragments produced by portlets are embedded into a *portlet window* as *portlet content*. Portlet windows on a page have several basic elements. In addition to the portlet content, the portlet window has a *decoration area* that can include the portlet title and controls to influence the window state and the mode. The user can control the size of the portlet window via the portlet window controls, from minimized (only the title is displayed) to normal to maximized (only portlet on the page). The *portlet mode* influences the requested function of the portlet. A portlet may offer help in a help mode or allow customizing the behavior in an edit mode.

The portal may aggregate several portlet windows to produce a complete portal page. This means that each portlet produces only the portlet content and the

portal produces everything else visible on the portal page, such as the portlet window, the portlet window controls, and the layout of the page.

Until now, all our examples have been HTML-based, but portlets aren't restricted to HTML. Figure 1.7 shows an example of a portlet that can produce different markups for different devices. For desktop browsers, it produces HTML markup; for Wireless Markup Language (WML) devices (e.g., mobile phones), it produces WML markup. This multidevice support offered by portlets lets users access the same applications regardless of the device they use.

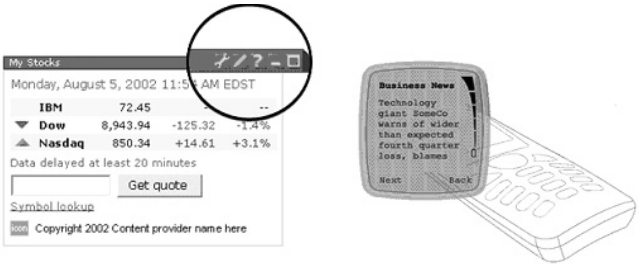

Figure 1.7: Alternative portlet markups (HTML for a desktop browser and WML on a mobile phone)

CREATING PORTLETS

Thus far, you've learned a lot about what portals and portlets are. In this section, we start looking at how to make portlets. There are three main scenarios:

- creating a new portlet-based application project
- migrating a portlet-based application from a proprietary portlet API to the standard portlet API (Java Portlet Specification)
- transforming an existing Web application into a portlet-based application

We'll cover all of these scenarios, beginning with creating new portlet applications from scratch.

Creating Portlet Applications from Scratch

Why would someone write portlet-based applications? That's a good question and one we'll answer in this section.

Previously, we defined a portlet as a Java-based Web component that processes requests and generates dynamic content. Thus our question should have been more precisely: Why would someone write portlet-based Web applications? In the days before portlets appeared, the Web programming model consisted of servlets and JavaServer Pages. We explain this programming model in more detail later, but for now we can concentrate on its major characteristics:

- *Adherence to a request/response paradigm:* Web applications communicate with the Web client by having the client send a request, and the Web application responds back to the client.
- *Self-contained application:* A Web application comes with all needed components and doesn't interact with other applications installed on the server running the Web application. This means you get one monolithic, consistent application that solves a specific problem, such as an Internet store.

While the first bullet also holds true for portlet applications, the second does not. In fact, this monolithic structure of Web applications was the reason portlets were invented. The monolithic structure of a complete application still has its use scenarios in a world of portlets — for example, in a self-contained online shop that doesn't need further customizations. However, more and more Web applications are becoming portal-like to permit users, as we've said, to work more in a desktop manner with different applications on one screen that can be interconnected and exchange data.

As the demand for more portal-like applications grows, the need to manage multiple applications in a single portal becomes critical. In all likelihood, more than one provider or group created the many applications typically included in a portal. Therefore, each Web application must be modular enough to fulfill several new requirements:

- Portlet applications must easily "plug in" to an existing portal to provide the portal with new functionality.
- Portlet applications must adhere to some rules to "play well" together with other portlet applications in the same portal.
- The portal must provide a unified user interface across multiple portlet applications.

The servlet API doesn't provide these kinds of rules and integration points; therefore, the new portlet API was created to support building modular Web applications that can be plugged into portals and produce content that is aggregated with content from other portlet applications into one page.

So the answer to the question at the beginning of this section is: You would write portlet applications if they will be used inside portals or if they need to be modular and integrate with other applications. But what if you've already created portlets using a proprietary portlet API? Let's consider this scenario.

JSR 168, the IBM Portlet API, and Other Tools

The current version of the Java Portlet Specification, JSR 168, was created jointly by several companies, including IBM, BEA Systems, Oracle, and Sun Microsystems, and released near the end of 2003. WebSphere Portal includes support for this standard to allow compatibility with other portals and portlet vendors.

IBM's strategy is to embrace open standards such as JSR 168. For this reason, we've decided to focus on that specification fully in this book. The JSR 168 standard is still evolving; the next version, called JSR 286, should be out toward the middle of 2007. (Appendix A describes the features expected in JSR 286.) Because of this, JSR 168 is still not as complete by itself as many portal teams require to achieve the functionality they need. With this point in mind, we focus the first part of this book on plain JSR 168 portlets, while the second part focuses on extensions and additional APIs for JSR 168 portlets that are WebSphere Portal–specific.

On another note, it's necessary to comment on portlet builder tools that have gained some popularity in the market lately. Over the past few years, several such products have become available. Some of these tools are better than others, and any of them may have a place in your organization or development effort. Having used some of these tools recently on various projects, we can say that it they indeed make it possible to quickly put together a simple portlet. Not every portlet or project is simple, however, and we want to caution you that total abstraction from the portlet API may not be possible or useful, and nearly every portal development effort requires some Java programming skills.

Moving from Proprietary APIs to Standard APIs

Here's another problem. Perhaps you were a portlet early adopter, but no standard existed when you started, or maybe you found the first version of the Java Portlet Specification too restrictive. For these reasons, you programmed your portlets against a proprietary portlet API. Now that the standard is in place, it's a good time to move from proprietary portlet APIs to the standard portlet API. Doing so will make your portlets vendor-independent, broadening their market or letting you move later on from your current portal vendor to a different one without throwing away all your portlet applications.

If you've written a portlet to a proprietary API, your situation looks like the one depicted in Figure 1.8. Here, the portlet is very tightly coupled to the available portal infrastructure because all the portlet APIs available are specific to the portal for which the portlet was developed.

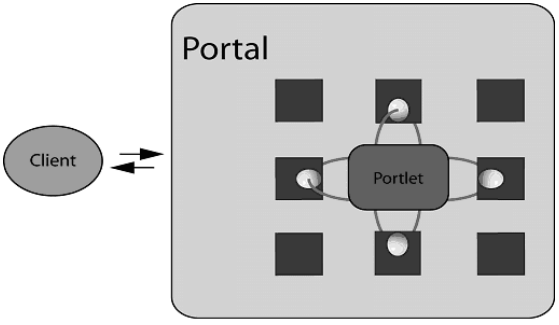

Figure 1.8: Portlets written against a proprietary portlet API

When you move to the standard API, there are two different cases to look at. In the first, the proprietary and standard APIs each support the same feature set; in the second, the proprietary API has a more robust feature set than the standard API.

Migrating Supported Features

The first case is the easy one. Here, the functionality that the portlet uses in the proprietary API is also available in the standard API; you can thus rewrite the portlet against the standard portlet API. This way, you end up with a portlet that's completely vendor-independent, guaranteed to run unchanged for years because

new versions of the portlet standard will be backward-compatible and easier to maintain. And even years from now, people will know how to program against V1.0 of the standard portlet API.

Migrating Non-Supported Features

Now for the more complex case. In this situation, your portlet uses some functionality currently unavailable in the standard portlet API, such as sending events between portlets. Even here, it may still be beneficial to move to the standard API and use vendor-specific extensions only for the functions not available in the standard API. You can then program your portlet to query the portal at runtime for supported extensions. If the portal lacks the extension (or extensions) you want, it can still run with degraded functionality.

Figure 1.9 depicts this scenario. As you see, the portlet plugs into two different APIs, the Java Portlet API and the Portal Extension API. Now, the portlet is decoupled from the portal infrastructure and can also run in a standard environment and just offer less functionality.

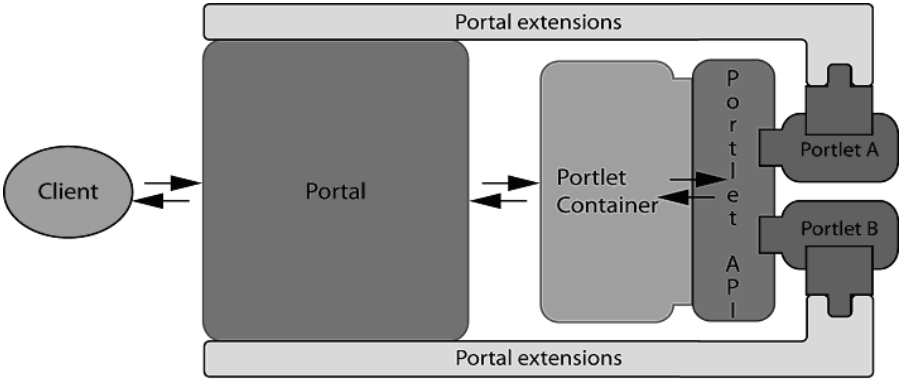

Figure 1.9: Portlet written against the standard portlet API but using vendor extensions when available

Now that we've covered creating portlet applications from scratch and converting portlets written against proprietary portlet APIs, let's take a look at transforming Web applications into portlet applications and thus handle the last use scenario for portlets.

Transforming Web Applications to Portlet Applications

This last scenario deals with another common case. You already have an existing, servlet-based Web application, and now you've read all this exciting stuff about portlets and want to leverage the advantages of portals and portlets. Of course, you'd like to avoid throwing away your existing code and starting again from scratch. It would be rather nice to put your existing Web application into a tool and have the tool transform it into a portal application. Unfortunately, reality isn't so easy, and only limited automated-tool support for transforming Web applications to portal applications is coming in the latest tool versions.

Let's look at the different ways your Web application might be implemented to see how easy or complicated it is to transform it into a portal application.

As with the last scenario, one implementation method is easier to transform than the other. If you were lucky enough to have written your Web application from scratch without using any Model-View-Controller (MVC) framework, such as Struts, just a few considerations apply. The answer largely depends on how modular your Web application is and whether you can easily factor out and remove the parts that don't fit in the portlet programming model we'll discuss in the next few chapters. These parts include

- using HTTP error codes or error JSPs. These parts must now be delegated to the portal.
- mixing of state-changing code and rendering code. Unless you separate these two, you'll have a hard time moving to the portlet programming model (or any MVC-based framework).
- parts of the code that deal with protocol handling and markup selection that aren't not cleanly separated. The portal handles these tasks in the portlet case.

If you stick to the Sun guidelines for J2EE Web applications (Reference 3), which recommend the MVC pattern, or if you've used an MVC-based framework, transforming your Web application into a portlet application should be a doable effort. You've still done only the first step with this, because your transformed portlet application most likely doesn't leverage the full power the portlet programming model provides. Using the complete data model will take some effort, but it will make your portlet application faster and easier to use.

MORE ABOUT TOOLS

Having mentioned tools above, let's take a quick look at the tools we'll be using for most of the portlets in this book. As Java programmers, we've mainly focused on building portlets in the traditional manner — that is, extending the PortletAdaptor or GenericPortlet class and filling in the code for the various methods (e.g., doView) that are provided. Among the different authors we take a few different approaches, but we developed most of the code, concepts, and images for this book using Rational Application Developer (RAD). Like any technology, these tools are rapidly evolving, and keeping up with the latest versions can be painful. For basic portlet programming, RAD is sufficient to create almost any portlet we discuss in this book. You may need some additional tools for some chapters, such as when we cover the Portlet Factory and WorkPlace Forms.

You can learn more about Rational Application Developer at the following location: *http://www.ibm.com/software/awdtools/developer/application/index.html*.

Required Skills

We've kept this first chapter brief so you can get started writing portlets right away! However, before we move on, some prerequisites will prove helpful in your portal development endeavor. Although we focus on using building portlets using available APIs and design and development best practices, you'll need a basic understanding of several topics to fully appreciate the information provided:

- a solid knowledge of Java and an understanding of server-side Java programming, such as servlets and JSPs
- familiarity with Web protocols, such HTTP
- a basic understanding of Extensible Markup Language (XML) to help understand Extensible Stylesheet Language Transformation (XSLT) and read deployment descriptor files
- basic familiarity with WebSphere Portal concepts, such as how to install portlets and set access control lists (ACLs)
- With this solid background and the information you'll learn from this book, you'll be writing portlets in no time.

SUMMARY

In this chapter, we covered the details of portals and portlets. First, we explained what a portal is and showed different applications of portals. The main point here is that portals are an integration point that integrates other applications into one consistent end-user application so users can work more in a desktop manner.

Next, we covered the role portlets play in the portal environment, noting that ssssthey are the central UI components that are rendered by the portal and that allow developers to extend the portal. We covered how portlets currently fit into existing J2EE architectures, why servlets aren't enough to provide portal components, and how portlets may be even more tightly integrated in future versions of J2EE. If portlets become part of J2EE, this will be a major achievement that will put portlets on par with servlets and enable them to leverage the complete J2EE infrastructure seamlessly.

Now that we've laid down some of the groundwork, let's get real in the next chapter by developing a real portlet in Chapter 2.

REFERENCES

1. Abdelnur, A., and S. Hepper. JSR 168: The Java Portlet Specification. *http://jcp.org/en/jsr/detail?id=168*.

2. Java 2 Platform Enterprise Edition 1.4 Specification. *http://www.jcp.org/en/jsr/detail?id=151*.

3. *Designing Enterprise Applications with the J2EE Platform, 2nd Ed.* Sun Developer Network, 2002. *http://java.sun.com/blueprints/guidelines/-designing_enterprise_applications_2e/index.html*.

4. Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.