Chapter 4

Managed Smart Clients

CHAPTER OVERVIEW

- Container-Managed Mobile Clients
- OSGi Containers
- OSGi Bundle Interactions
- IBM Service Management Framework
- A Simple Echo Service Example
- Smart Client with HTTP Front End
- The Pizza Order Example
- Mobile Gateways

In Chapter 3, we discussed the smart client paradigm. However, for complex mobile applications, monolithic clients with intertangled code segments are very hard to develop and maintain. We need a framework to develop modularized application and service components. In this chapter, we introduce the concept of managed smart clients—self-contained components inside software containers. The container provides common crosscutting services and a framework for components to communicate with each other. The industry standard for lightweight mobile containers is the OSGi (Open Services Gateway initiative) specification. Using several example OSGi applications running on IBM Service Management Framework, we explain how OSGi-managed smart clients work in real-world applications. In addition to supporting managed clients, OSGi applications can also deliver mobile gateway services. We will cover the gateway architecture near the end of this chapter.

4.1 Container-Managed Applications

In the field of software engineering, the term *container* refers to specialized software that runs other software. For example,

- The MIDP Application Management Software (AMS) is a container that installs, starts, pauses, stops, updates, and deletes MIDlet applications. In the CDC Personal Basis Profile, the Xlet programming model also features container-managed life-cycle methods.
- A Java servlet engine is a container that invokes servlets and provides access to the HTTP context.
- The Java Virtual Machine (JVM) itself is a container. It monitors Java applications for proper memory usage (garbage collector) and security.

In the next two sections, we will discuss the features and benefits of mobile containers.

4.1.1 Container Features

As mobile enterprise applications become mainstream, the complexity of smart clients grows. For example, fully commercial applications often require features such as user login, logging, transaction, and transparent data access. Without proper tools for code and service reuse, mobile developers have to duplicate those functionalities for every smart client. Wasting time reinventing the wheel is not only inefficient but also causes error-prone code.

Containers in J2EE
Containers are central to the serverside Java technologies (J2EE). For example, the core value proposition of the popular Enterprise JavaBean (EJB) technology lies in EJB containers that automatically take care of security, transaction, logging, synchronization, persistence, and many crosscutting application concerns. The EJB developers can focus on coding the value-added core business logic. The result is much better software and drastically improved developer productivity. Years of intensive research in J2EE have developed many advanced techniques to design and implement software containers.

Hence, it makes sense to make those common features available as services in software containers that run on mobile devices. An advanced container usually provides the following functionalities.

- *Self-contained applications*: Applications run inside the container are self-contained with portable code and necessary configuration files. The interdependence of applications and library components could be managed by the container. Examples are the WAR files for servlet containers and EAR files for EJB containers.
- *Life-cycle management*: By calling the life-cycle methods defined in the container framework and implemented by all applications, the container can install, start, stop, update, and delete any application programmatically or through an interactive console.
- *Application services*: The container provides services that are common to all applications. For example, an authentication module in the container could allow all applications to authenticate against a single password database.
- *Custom services:* The container should also allow its applications to offer services to each other. That encourages code reuse and prompts architectures for layered and modularized applications.

In this book, we use the term *container* rather loosely. Our containers do not impose arbitrary boundaries for API usages. Applications installed inside the container can transparently access any Java or native API available on the device. These containers are often known as *frameworks*. The container architecture on J2ME mobile devices is illustrated in Figure 4.1.

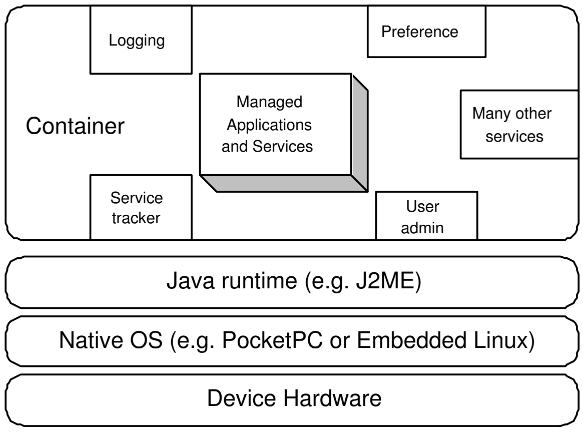


Figure 4.1. The container architecture for J2ME smart clients.

4.1.2 Benefits of Containers

The above container features translate to real benefits in mobile development projects:

- *Reduced code redundancy*: Since the common services are not repeatedly implemented, we can reduce overall footprint and potential number of errors while improving the developer productivity.
- *Managed update*: When we fix a bug or add a new feature in a service, all applications that use it automatically get the update. Some containers support service versioning for more refined controls.
- *Support for multitiered application models*: Services in a container offer natural separations between application tiers (e.g., the presentation and business layers).
- *Simplified application provisioning*: Self-contained applications can be easily deployed to any container. That enhances Java's value proposition of "write once, run anywhere."

Given these benefits, containers or frameworks are widely used in mobile Java application development. In the next section, we introduce a standard container specification for lightweight mobile devices: the OSGi specification.

Note

Every MIDP device comes with the AMS container for provisioning, security, and life-cycle management. However, the MIDP platform is too resource-constrained to run any more advanced containers. As a result, the containers we discuss in this chapter require at least J2ME/CDC or PersonalJava runtimes.

4.2 OSGi Containers

The OSGi Alliance is an industry consortium that creates open standard specifications for network-delivered services. Founded in March 1999, OSGi is a nonprofit organization with open membership. Its board of directors includes Acunia, BMW, Deutsche Telekom, Echelon, Gatespace, IBM, Motorola, Oracle, Philips, ProSyst, Samsung, Sun and Telcordia. The OSGi specification defines the mobile container framework and standard container services as Java APIs that span from J2ME to J2SE to J2EE.

The OSGi Service Platform Release 2 specification was released in October 2001. It has been widely adopted by vendors and has many implementations. The OSGi Service Platform Release 3 specification was made available in March 2003. The IBM Services Management Framework (SMF) v3.5 is targeted to be OSGi R3 compatible. It is available for free evaluation from the IBM Web site (see "Resources"). We discuss both Release 2 and 3 in this chapter. However, all examples are written for and tested on Release 2 containers.

Note

Despite the term *Gateway* in its name, the OSGi specification does not define any particular kind of gateway servers. It defines a framework for service components delivery and execution. The OSGi container provides the runtime environment for those services. Gateway server is only one of OSGi's application areas.

4.2.1 Bundles

OSGi applications are packaged as *bundles*, which are just standard JAR files. The OSGi bundle is completely self-contained with all the necessary metadata in its manifest file. The OSGi container completely manages the bundle life cycle:

- Install, update, and uninstall the bundle.
- Start and stop the bundle.
- Register, unregister, and track services in the bundle.

The bundle management interfaces are defined in the org.osgi.framework package. Since the bundles can be deployed to the container dynamically without restarting the container, the OSGi platform is an ideal choice for mobile application provisioning clients. It allows applications to be managed, tracked, and updated throughout its lifetime.

4.2.2 Standard Services

The OSGi container provides common crosscutting services such as device drivers, user preferences, and logging to all its bundles. Table 4.1 lists the OSGi services defined in OSGi Service Platform Release 2 specification. The new OSGi Service Platform Release 3 specification defines more services, some of which are of great importance to mobile applications. Those new services are listed in Table 4.2.

4.2.3 Bundle Interaction and Custom Services

The OSGi framework provides powerful ways for bundles to interact with each other. This encourages code reuse and makes it easier to architect complex multilayer applications. For example, the OSGi container on a stock trader's PDA might be provisioned with services bundles from major exchanges. Each bundle knows how to run real-time queries and execute trades in a specific exchange market, and it makes those functionalities available to other bundles. The trader can then deploy the actual trading client bundle, which provides a user interface, supports custom trade logic, and executes the query/trade through the individual service bundles. The possible interactions among bundles are as follows.

- *Static sharing*: The OSGi container runs on a single JVM instance but has a different classloader for each bundle. That means bundle namespaces are separate. We cannot directly access objects or classes in another bundle by default. However, a bundle can explicitly export some of its Java packages through the Export-Package attribute in its manifest file. It can also import Java packages exported by others using the Import-Package manifest attribute. The export and import features allow direct sharing of Java packages.

Table 4.1. OSGi Services in OSGi Service Platform Release 2 Specification (org.osgi.*)

Java package	Description
service.http	The HTTP service responds to HTTP requests. The service listens on ports specified in the container configuration. It dispatches each HTTP request to a handling servlet based on an URL-to-servlet mapping table registered by individual bundles. Finally, it returns the servlet's response to the HTTP requester. The service also handles HTML content without the help of a servlet.
service.device	The device service manages custom device adaptors. It allows bundle developers to plug in device drivers and develop algorithms to match devices to drivers. This service allows the OSGi bundles to respond to many different types of client devices.
service.prefs	The preference service manages a hierarchical collection of preference data resembling the JDK v1.4 preference API. It is much more advanced than simple Java property files.
service.useradmin	The user administration service provides role-based authorization service. It manages user credentials and user groups.
service.permissionadmin	The permission service allows operators to manage bundle permissions.
service.packageadmin	The package administration service manages Java packages exported by bundles (see Section 4.2.3 for exported packages).
service.metatype	The metatype service provides a mechanism for bundles to expose their configuration metadata.
service.cm	The configuration manager service administrates bundle configurations.
service.log	The logging service logs messages during the bundle execution. We can extend the basic logging service interface for custom logging needs.
util.tracker	The ServiceTracker class in this package provides easy ways to use and manage the container's service registry.

Table 4.2. New OSGi Services in OSGi Service Platform Release 3 Specification (org.osgi.*)

Java package	Description
service.startlevel	A policy service that allows the developer to specify the startup and shutdown sequence of bundles.
service.url	This service allows bundles to register URL schemes with content types and provide content handlers for the registered types.
util.xml	This is a utility service that allows bundles to use JAXP, SAX, and DOM XML parsers. Each parser interface can have multiple implementations.
service.wireadmin	It supports a convenient way to connect data producers and consumers. Two utility classes are commonly used with the wireadmin service to handle measurement-related (e.g., error calculation and unit conversion) and position-related (location, speed, orientation) data.
service.io	This service allows bundles to handle arbitrary network protocols using the J2ME Generic Connection Framework (GCF). Since the GCF is a layered and abstract framework, bundles only need to extend the abstract connection factory to return the correct connection class based on the URL string format.
service.upnp	This service makes OSGi bundles transparently available to universal plug-and-play networks.
service.jini	This service allows OSGi bundles to interact with Jini network services.

- *Dynamic services*: In addition to standard services provided by the container, any bundle can consume and provide services from/to other bundles at the same time:

    1. A bundle can dynamically register (or unregister) services with the container. The bundle needs to register the service interface with a concrete implementation class. Any change to the service (register, modify, unregister) will result in framework events that could be captured and processed.

2. Another bundle finds the service reference through a lookup API in the framework. It calls a framework method to obtain the service implementation object from the service provider bundle. The service object is now ready to use.

The interacting bundles allow us to deliver reusable services to any OSGi node, from the high function grid to pervasive devices.

4.2.4 OSGi Runtime Requirements

Currently, different OSGi vendors have different requirements for their products. The required execution environments range from PersonalJava v1.1 to J2SE. This has created considerable confusion in the developer community. In an effort to standardize the runtime requirements, the OSGi Service Platform Release 3 specification formally defines the following runtime environments:

- *The Java 2 Micro Edition*: All OSGi implementations should run under the CDC v1.0 plus Foundation Profile v1.0 runtime environment.
- *The OSGi minimum execution environment*: The specification also defines a subset of CDC/FP APIs, which allows devices not powerful enough for the CDC/FP (e.g., Palm PDAs) to run the OSGi framework. The OSGi minimum execution environment is defined to be a proper subset of CDC/FP and J2SE.

The standard execution environments make it easier for developers, especially resource-conscious mobile developers, to choose the right OSGi product.

4.3 A Simple Echo Service Example

In this section, we first introduce a J2ME-compatible OSGi implementation from IBM. Using a simple echo example, we demonstrate how to implement bundles and share services among them.

4.3.1 The IBM Service Management Framework

The IBM SMF is a readily available OSGi implementation. It has a memory footprint of 3 MB and runs on both execution environments defined in the OSGi Service Platform Release 3 specification. IBM supports the J2ME environments through WME (WebSphere Micro Environment JVM) and the minimum execution environment through WCE (WebSphere Custom Environment JVM) products. It can be tightly integrated into IBM's WebSphere Studio Device Developer IDE. The SMF product versions we cover in this book are v3.1 for OSGi R2 and v3.5 for OSGi R3.

The SMF installation process varies among devices. It generally involves the following steps.

1. Download and unpack the SMF toolkit from IBM.

2. Copy the following directories and files to the target device (or to a local execution directory, if you want to run SMF on a desktop computer). For my PocketPC device, I put all four items under the device root directory.

   - The jarbundles directory contains installed bundles.
   - The smf.jar file provides implementation classes for the OSGi specification.
   - The smfconsole.jar file provides a command-line management console for the container.
   - The smf.properties file specifies the runtime configuration.

3. Make sure that the com.ibm.osg.smf.bundledir property in the smf.properties file points to the correct bundle directory. For example,

   ```
   com.ibm.osg.smf.bundledir=jarbundles
   ```

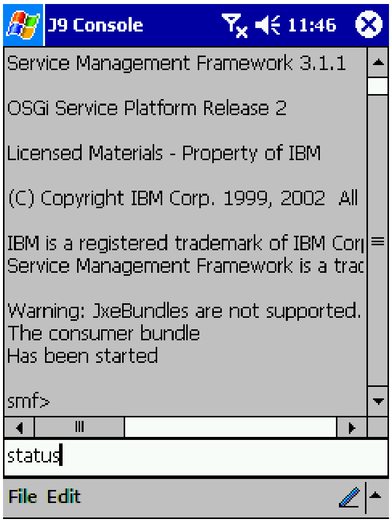
4. Now we can start the SMF console using the following command (in one line) or its equivalent on the device platform.

   ```
   java -classpath "smf.jar:smfconsole.jar"
   com.ibm.osg.smf.SMFLauncher -console "launch"
   ```

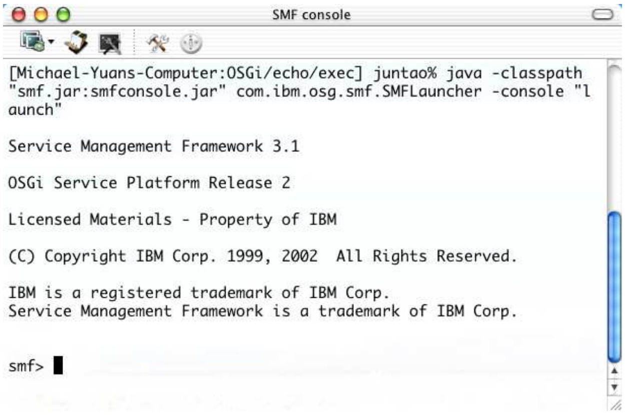
   For my PocketPC device with IBM WebSphere Micro Environment preinstalled, I use the following command (in one line). Please refer to the Appendix B for the steps to install the IBM J2ME runtimes on PDA devices.

   ```
   "\WSDD\j9.exe" -jcl:foun
   "-Xbootclasspath:\WSDD\lib\jclFoundation\classes.zip;
   \smf.jar;\smfconsole.jar" "com.ibm.osg.smf.SMFLauncher"
   -console "Launch"
   ```

After the SMF console is started, it loads all currently installed bundles into the container and presents the user a command-line interface for management tasks. For a complete list of management commands, please refer to the SMF manual. Figure 4.2 shows the command-line console on desktop and PocketPC devices. Table 4.3 lists some of the most frequently used commands.



```
Service Management Framework 3.1.1

OSGi Service Platform Release 2

Licensed Materials - Property of IBM

(C) Copyright IBM Corp. 1999, 2002  All

IBM is a registered trademark of IBM Cor|
Service Management Framework is a trac

Warning: JxeBundles are not supported.
The consumer bundle
Has been started

smf>
status
```



```
[Michael-Yuans-Computer:OSGi/echo/exec] juntao% java -classpath
"smf.jar:smfconsole.jar" com.ibm.osg.smf.SMFLauncher -console "l
aunch"

Service Management Framework 3.1

OSGi Service Platform Release 2

Licensed Materials - Property of IBM

(C) Copyright IBM Corp. 1999, 2002  All Rights Reserved.

IBM is a registered trademark of IBM Corp.
Service Management Framework is a trademark of IBM Corp.


smf>
```

Figure 4.2. The SMF console on desktop and PocketPC devices.

In the next two sections, we describe how to create and deploy two OSGi bundles: The EchoService bundle exposes an echo service; the EchoUIConsumer bundle presents a simple GUI client and uses the EchoService in the container to echo user input. Figures 4.3 and 4.4 show the two bundles in action in J2SE and J2ME OSGi containers. Figure 4.5 shows user interactive OSGi bundles.

4.3.2 The EchoService Bundle

The EchoService bundle demonstrates how to implement and register a service in an OSGi bundle. The service itself is extremely simple: It only defines one method that does nothing more than echo a string input. The steps to create the bundle are as follows.

1. Define the service interface as a Java interface (Interface EchoService, Listing 4.1).

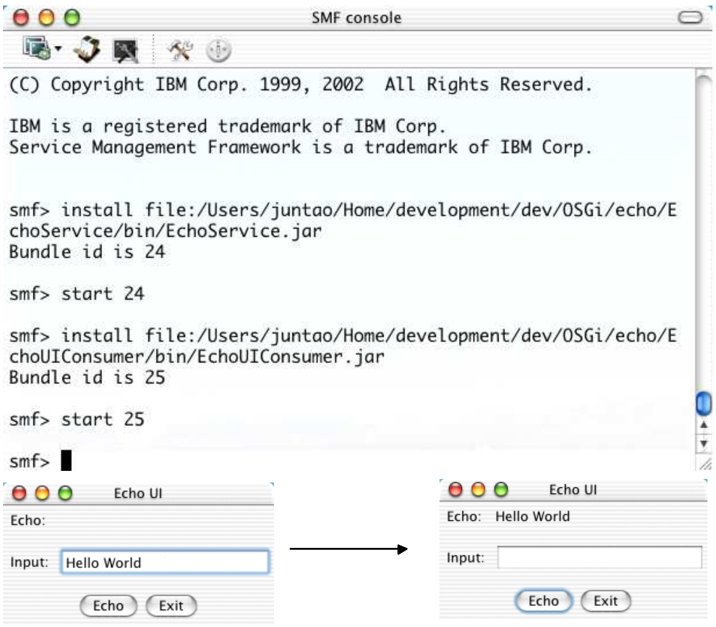
Table 4.3. Common Commands Available in the SMF Console

Command	Description
install url	Installs a bundle from a URL and returns a bundle ID. The URL could point to files on the local file system, such as file:/path/bundle.jar.
update id	Updates the package from the same URL as specified by the install command.
uninstall id	Uninstalls the bundle.
start id	Starts the bundle.
stop id	Stops the bundle.
bundle id	Displays information about an installed bundle.
status	Displays all installed bundles and registered services.
packages	Displays the imported and exported packages for each bundle.
close	Shuts down the container and exits the console.

2. Create an implementation class for the interface (Class EchoserviceImpl, Listing 4.2).
3. Create a BundleActivator class that implements the required OSGi life-cycle methods and registers the service with the container upon startup (Class EchoActivator, Listing 4.3).
4. Create a manifest file that specifies the BundleActivator class for this bundle and exports the service interface package (Listing 4.4). The OSGi container uses the manifest to find the entry point of the bundle and collect necessary configuration data.
5. Package the compiled classes and manifest file into a standard Jar file.

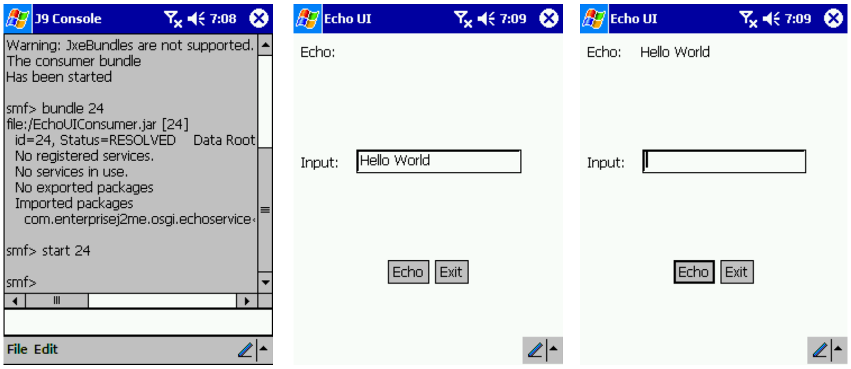
Listing 4.1. The EchoService interface

```
package com.enterprisej2me.osgi.echoservice;

public interface EchoService {
  public String echo (String s);
}
```



```
(C) Copyright IBM Corp. 1999, 2002  All Rights Reserved.

IBM is a registered trademark of IBM Corp.
Service Management Framework is a trademark of IBM Corp.


smf> install file:/Users/juntao/Home/development/dev/OSGi/echo/E
choService/bin/EchoService.jar
Bundle id is 24

smf> start 24

smf> install file:/Users/juntao/Home/development/dev/OSGi/echo/E
choUIConsumer/bin/EchoUIConsumer.jar
Bundle id is 25

smf> start 25

smf>
```

Figure 4.3. The echo bundles in action in J2SE.



```
Warning: JxeBundles are not supported.
The consumer bundle
Has been started

smf> bundle 24
file:/EchoUIConsumer.jar [24]
  id=24, Status=RESOLVED    Data Root
  No registered services.
  No services in use.
  No exported packages
  Imported packages
    com.enterprisej2me.osgi.echoservice

smf> start 24

smf>
```

Figure 4.4. The echo bundles in action in J2ME.

Listing 4.2. The `EchoServiceImpl` class

```
package com.enterprisej2me.osgi.echoserviceimpl;
```

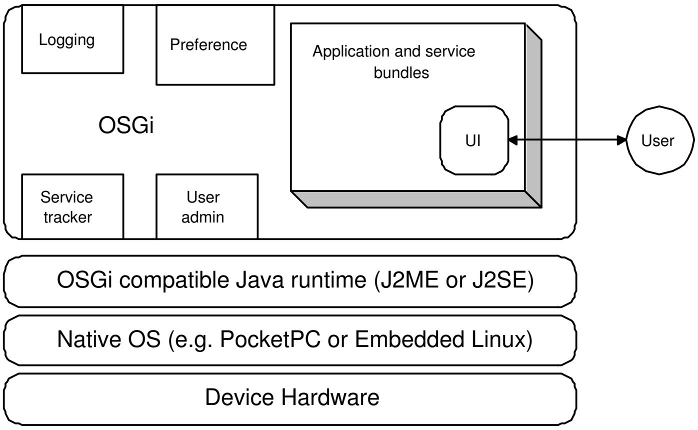

Logging
Preference
Application and service bundles
OSGi
UI
User
Service tracker
User admin
OSGi compatible Java runtime (J2ME or J2SE)
Native OS (e.g. PocketPC or Embedded Linux)
Device Hardware

Figure 4.5. OSGi bundles with UIs.

```
import com.enterprisej2me.osgi.echoservice.*;

public class EchoServiceImpl implements EchoService {

  EchoServiceImpl () { }

  public String echo (String s) {
    return s;
  }

}
```

Listing 4.3. The EchoActivator class

```
package com.enterprisej2me.osgi.echoserviceimpl;

import org.osgi.framework.*;
import com.enterprisej2me.osgi.echoservice.*;


public class EchoActivator implements BundleActivator {

  private ServiceRegistration reg;

  public EchoActivator () { }
  public void start (BundleContext context) throws Exception {
    EchoServiceImpl impl = new EchoServiceImpl ();
    reg = context.registerService (
        EchoService.class.getName(), impl, null);
  }

  public void stop (BundleContext context) throws Exception {
    reg.unregister ();
  }
}
```

Listing 4.4. The JAR manifest for the echo service bundle

```
Manifest-Version: 1.0
Bundle-Name: Echo service
Bundle-Description: Echo the input
Bundle-Activator: com.enterprisej2me.osgi.echoserviceimpl.EchoActivator
Import-Package: org.osgi.framework; specification-version=1.1
Export-Package: com.enterprisej2me.osgi.echoservice
Export-Service: com.enterprisej2me.osgi.echoservice.EchoSerivce
```

Now, we can install and start the package in our SMF console.

4.3.3 The EchoUIConsumer Bundle

The EchoUIConsumer bundle is created to demonstrate how to use the echo service through the framework:

1. Create a BundleActivator implementation as the entry point to the bundle (Class EchoUIConsumer, Listing 4.5).

2. In the EchoUIConsumer.start() method, create and open a ServiceTracker object to track the echo service we started.

3. Create the UI frame class EchoFrame (Listing 4.6) and pass the ServiceTracker object and the current bundle (i.e., the echo consumer bundle) to EchoFrame.

4. The EchoFrame object obtains the EchoService object from the ServiceTracker and uses the EchoService to echo any user input. When we hit the Exit button in the UI frame, the AWT event handler calls

the bundle's stop() method and triggers the container to invoke the EchoUIConsumer.stop() method. For more details, refer to method actionPerformed() in Listing 4.6.

5. In the EchoUIConsumer.stop() method, dispose the UI frame and close the ServiceTracker (Listing 4.5).
6. Create the manifest file (Listing 4.7). We import the package containing the EchoService interface here.
7. Package and deploy the JAR bundle.

Tracking the Services
In the service consumer bundles, we could manually look up the service objects from the framework. Then, we would have to register event listener and callback methods to handle situations when other bundles or the container itself changes those services (e.g., removes the service). This could be a tedious task. Instead, we take a shortcut and use a pair of ServiceTracker objects to automatically track those services.

The ServiceTracker object tracks a list of services meeting certain criteria passed to it in the constructor. It provides default event handlers for the services it tracks.

The ServiceTracker object can be instantiated with a ServiceTrackerCustomizer object. When a service in the tracker is added, modified, or deleted, the appropriate method in its associated ServiceTrackerCustomizer is called. For more usage examples of the ServiceTracker class, please refer to Section 4.4.

Listing 4.5. The EchoUIConsumer class

```
package com.enterprisej2me.osgi.echouiconsumer;

import org.osgi.framework.*;
import org.osgi.util.tracker.*;
import com.enterprisej2me.osgi.echoservice.*;

public class EchoUIConsumer implements BundleActivator {

  ServiceTracker echoTracker;
  EchoFrame frame;

  public EchoUIConsumer () { }
  public void start (BundleContext context) {
    echoTracker = new ServiceTracker (context,
               EchoService.class.getName(), null );
    echoTracker.open ();
    frame = new EchoFrame(250, 250, echoTracker, context.getBundle());
  }

  public void stop (BundleContext context) {
    frame.dispose ();
    echoTracker.close();
  }
}
```

Listing 4.6. The EchoFrame class

```
package com.enterprisej2me.osgi.echouiconsumer;

import java.awt.*;
import java.awt.event.*;
import org.osgi.framework.*;
import org.osgi.util.tracker.*;
import com.enterprisej2me.osgi.echoservice.*;

public class EchoFrame extends Frame
    implements WindowListener, ActionListener {

  private TextField entryText;
  private Label echoedText;
  private Button submit;
  private Button exit;
  private Panel content, top, bottom, middle;
  private ServiceTracker echoTracker;
  private Bundle echoUIConsumerBundle;

  public EchoFrame (int width, int height,
                    ServiceTracker t, Bundle b) {
    super ("Echo UI");
    setBounds(0, 0, width, height);
    echoTracker = t;
    echoUIConsumerBundle = b;

    entryText = new TextField (20);
    echoedText = new Label ("   ");
    submit = new Button ("Echo");
    exit = new Button ("Exit");
    submit.addActionListener (this);
    exit.addActionListener (this);

    top = new Panel ();
    middle = new Panel ();
    bottom = new Panel ();
    top.setLayout(new FlowLayout(FlowLayout.LEFT));
    top.add(new Label("Echo: "));
    top.add(echoedText);
    middle.setLayout(new FlowLayout(FlowLayout.LEFT));
    middle.add(new Label("Input: "));
    middle.add(entryText);
    bottom.setLayout(new FlowLayout(FlowLayout.CENTER));
    bottom.add(submit);
    bottom.add(exit);

    content = new Panel ();
    content.setLayout(new GridLayout(3, 1));
    content.add(top);
    content.add(middle);
    content.add(bottom);

    add (content);
    addWindowListener(this);
    pack ();
    setVisible (true);
  }

  public void actionPerformed (ActionEvent e) {
    if ( e.getSource() == submit ) {
      top.remove (echoedText);

      // Obtain the echo service object
      EchoService echoObj = (EchoService) echoTracker.getService();
      // Use the echo service to echo a string
      echoedText = new Label ( echoObj.echo(entryText.getText()) );

      top.add (echoedText);
      entryText.setText("");
      setVisible (true);
    } else if ( e.getSource() == exit ) {
      // see note
      // echoUIConsumerBundle.stop();
      dispose ();
    }
  }

  public void windowClosing(WindowEvent e) {}
  public void windowOpened(WindowEvent e) {}
  public void windowClosed(WindowEvent e) {}
  public void windowIconified(WindowEvent e) {}
  public void windowDeiconified(WindowEvent e) {}
  public void windowActivated(WindowEvent e) {}
  public void windowDeactivated(WindowEvent e) {}

}
```

Listing 4.7. The manifest file for the echo consumer bundle

```
Manifest-Version: 1.0
Bundle-Name: Echo UI consumer
Bundle-Description: Consume the echo service
Bundle-Activator: com.enterprisej2me.osgi.echouiconsumer.EchoUIConsumer
Import-Package: com.enterprisej2me.osgi.echoservice,
 org.osgi.framework; specification-version=1.1,
 org.osgi.util.tracker; specification-version=1.1
Import-Service: com.enterprisej2me.osgi.echoservice.EchoSerivce
```

Limitations of the Bundle State-Change APIs

Note that when we exit the EchoFrame UI, it merely disposes the UI window but does not stop the underlying bundle. That is because the synchronous state-change APIs in the current OSGi specification do not allow a bundle to change its own state safely. This limitation is being addressed by the OSGi expert group. When it is resolved in a future OSGi edition, the EchoFrame exit event handler can simply call the echoUIConsumerBundle.stop() method to dispose the window and stop the bundle.

The Espial DeviceTop
The Espial DeviceTop is an OSGi implementation running on PersonalJava platforms. In addition to standard OSGi services, it provides an *application* service that supports bundles with GUIs. The bundle's BundleActivator class can extend the espial.devicetop.refui.Application class, which automatically takes care of the interactions between the UI frame and the bundle itself. With Espial's proprietary Espresso UI library, we can create sophisticated mobile UI applications on the DeviceTop framework.

4.4 Smart Client with HTTP Front End

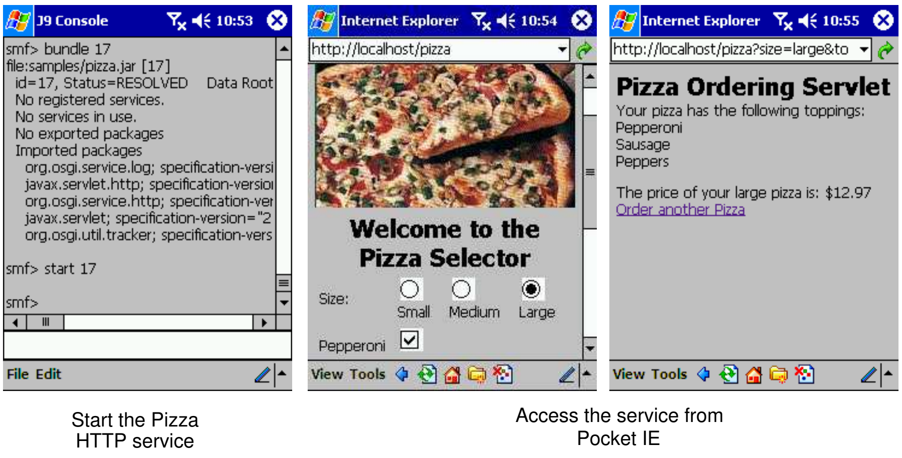
The managed GUI bundle uses only a fraction of the power provided by the OSGi container. Through its services, the OSGi container supports external applications and devices over the network. To use a separate program to render the UI, we can more effectively separate the business and presentation layers.

The "pizza order" example application distributed with the SMF illustrates the use of HTTP services in the OSGi framework. After starting the bundle from the SMF console, we can launch the device's built-in HTML browser (Internet Explorer for PocketPC or the Opera browser for Embedded Linux) and point the URL to http://localhost/pizza. An HTML page of a dummy pizza store appears. We can fill out the pizza order form, submit the form, and get response from a servlet running inside the bundle. The screen flow is shown in Figure 4.6.

This design allows us to build a simple UI very quickly, using HTML without messaging with complex event handlers in AWT code. It also allows the vast majority of serverside Java developers to transfer their skills and make use of their familiar patterns, such as the Model-Viewer-Controller pattern. The overall architecture of the smart client with HTTP front end is illustrated in Figure 4.7.

4.4.1 The Pizza Order Bundle

The PizzaBundle class (Listing 4.8) implements the BundleActivator interface. This bundle does not register or provide any new services. It customizes the container HTTP service to serve pizza order HTML content at a specified URL. It also uses the container logging services to record activities inside the bundle. If no logging service has been registered for this bundle, it logs to the standard output.



```
smf> bundle 17
file:samples/pizza.jar [17]
  id=17, Status=RESOLVED    Data Root
  No registered services.
  No services in use.
  No exported packages
  Imported packages
    org.osgi.service.log; specification-versi
    javax.servlet.http; specification-versio
    org.osgi.service.http; specification-ver
    javax.servlet; specification-version="2
    org.osgi.util.tracker; specification-vers

smf> start 17

smf>
```

http://localhost/pizza

Welcome to the Pizza Selector

Size: ○ Small ○ Medium ● Large

Pepperoni ☑

http://localhost/pizza?size=large&to

Pizza Ordering Servlet

Your pizza has the following toppings:
Pepperoni
Sausage
Peppers

The price of your large pizza is: $12.97
Order another Pizza

Figure 4.6. The pizza order application in action.

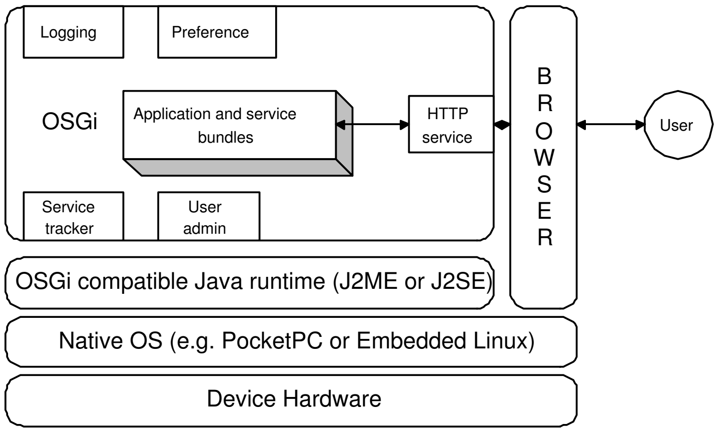

Logging
Preference
OSGi
Application and service bundles
HTTP service
Service tracker
User admin
OSGi compatible Java runtime (J2ME or J2SE)
BROWSER
User
Native OS (e.g. PocketPC or Embedded Linux)
Device Hardware

Figure 4.7. The smart client with HTTP front end.

1. The bundle start() method instantiates trackers for LogService and HttpService. The bundle then invokes their open() methods.

2. When the ServiceTracker for the HttpService is opened, it obtains all registered HttpService references from the container, adds them into the tracker, and invokes the corresponding ServiceTrackerCustomizer's addingService() method for each added reference.

3. The addingService() method obtains the HTTP service object from the container and customizes it with the pizza order servlet and other resources.
4. When the bundle stops, its stop() method calls the close() methods of the two ServiceTrackers. The HttpService tracker in turn calls the removedService() method, which unplugs the servlet from the HTTP service.

Listing 4.8. The PizzaBundle class

```
public class PizzaBundle implements
  BundleActivator, ServiceTrackerCustomizer {

  /** BundleContext for this bundle */
  protected BundleContext context;

  /** Log Service wrapper object */
  protected LogTracker log;

  /** Http Service tracker object */
  protected ServiceTracker tracker;

  /** HttpContext for HTTP registrations */
  protected HttpContext httpContext;

  // ... ...

  public PizzaBundle() { }

  // Methods in BundleActivator
  public void start(BundleContext context) throws Exception {
    this.context = context;
    httpContext = new HttpContext() { ... ... };
    log = new LogTracker(context, System.err);
    tracker = new ServiceTracker(context,
        HttpService.class.getName(), this);
    tracker.open();
  }

  public void stop(BundleContext context) throws Exception {
    tracker.close();
    log.close();
  }

  // Methods for ServiceTrackerCustomizer
  public Object addingService(ServiceReference reference) {
    HttpService http = (HttpService)context.getService(reference);
    if (http != null) {
      try {
        http.registerServlet(servletURI,
            new Pizza(), null, httpContext);
        http.registerResources(servletURI+imagesURI,
                          imagesURI, httpContext);
        log.log(log.LOG_INFO, "Pizza Servlet registered");
      } catch (Exception e) {
        // handle the exception
      }
    }
    return http;
  }

  public void modifiedService(ServiceReference
                  reference, Object service) {
  }

  public void removedService(ServiceReference
                  reference, Object service) {
    HttpService http = (HttpService) service;

    http.unregister(servletURI);
    http.unregister(servletURI+imagesURI);

    context.ungetService(reference);
    log.log(log.LOG_INFO, "Pizza Servlet unregistered");
  }
}
```

4.4.2 The Pizza Order Servlet

In the addingService() method, we use a servlet Pizza (Listing 4.9) to provide the custom HTTP service (the application logic). This is just a standard Java servlet that reads from HttpRequest and writes HTML data to HttpResponse objects. Those HTTP context objects are provided by the container.

Listing 4.9. The Pizza servlet

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class Pizza extends HttpServlet {
  protected void doGet(HttpServletRequest req,
                       HttpServletResponse res)
            throws ServletException, IOException {
    // Generate some output
    res.setContentType("text/html;" + "charset=iso-8859-1");
    PrintWriter out = res.getWriter();
    out.print(" ... ... ");

    // Get query parameters
    String queryString = req.getQueryString();

    // any pizza order logic
    // ... ...

    out.println("</body></html>");
  }
}
```

4.4.3 The Logging Service

A container can have multiple logging services. For example, one service implementation could log messages to a disk file while another could send the critical message as Instant Message alerts to administrators. The OSGi framework provides a common LogService interface for all logging services. Implementations of the LogService interface are provided and shared by individual bundles. In our pizza example, the LogTracker (Listing 4.10) object is a ServiceTracker object that tracks all available logging services from the container registry and makes sure each message is logged by all services.

1. The LogTracker.open() method is invoked in the bundle start() method to initiate the tracker.

2. LogTracker.open() calls its base class's open() method, which obtains all LogService references in the container.

3. When the bundle needs to log a message, it calls the LogTracker.log() method, which iterates through the current list of tracked LogService references. It obtains the service object for each reference and pushes the message to all available logging services.

4. If the container does not have any registered logging service (the reference list size is zero), the LogTracker object will call its noLogService() method to log to the standard output.

Note

The LogTracker class does not register any new LogService to the container.

Listing 4.10. The LogTracker class

```
public class LogTracker
     extends ServiceTracker implements LogService {

  protected final static String clazz =
        "org.osgi.service.log.LogService";
  protected PrintStream out;

  public LogTracker(BundleContext context, PrintStream out) {
    super(context, clazz, null);
    this.out = out;

    open();
  }

  // Implements various log() messages with
  // different signatures.
  // ... ...

  public synchronized void log(ServiceReference reference,
       int level, String message, Throwable exception) {
    ServiceReference[] references = getServiceReferences();

    if (references != null) {
      int size = references.length;
      for (int i = 0; i < size; i++) {
        LogService service = (LogService) getService(references[i]);
        if (service != null) {
```

```
        try {
          service.log(reference, level, message, exception);
        } catch (Exception e) { }
      }
    }
    return;
  }
  noLogService(level, message, exception, reference);
}

protected void noLogService(int level, String message,
   Throwable throwable, ServiceReference reference) {
  if (out != null) {
    synchronized (out) {
      switch (level) {
        case LOG_DEBUG: {
          out.print("Debug: ");
          break;
        }
        case LOG_INFO: {
          out.print("Info: ");
          break;
        }
        case LOG_WARNING: {
          out.print("Warning: ");
          break;
        }
        case LOG_ERROR: {
          out.print("Error: ");
          break;
        }
        default: {
          out.print("Unknown Log level[");
          out.print(level);
          out.print("]: ");
          break;
        }
      }
      out.println(message);
      if (reference != null) {
        out.println(reference);
      }
      if (throwable != null) {
        throwable.printStackTrace(out);
      }
      }
    }
  }
}
```

4.4.4 Rich UI Clients for the HTTP Service

Although the pizza order example supports clearly separated application layers, the drawback is that it does not really take advantage of the rich UI capability of smart clients. There are several ways to create rich UI clients for the HTTP service.

- *Rich browsers*: Instead of plain HTML content, the servlet can provide rich content such as Java Applet and Flash for capable browsers.
- *Standalone GUI*: We can also replace the browser completely with a standalone GUI application. The OSGi HTTP service can serve binary or XUL (XML User Interface) content and allow the standalone GUI front end to decide how to render it.

4.5 Mobile Gateways

In the previous sections, we discussed service and application bundles in clientside OSGi containers. Besides clientside containers, another major application area of the OSGi framework is to deploy and execute services on mobile gateway devices that do not have UI front ends. Small, pervasive devices delegate computationally expensive tasks to the more powerful gateway. In the gateway configuration, the OSGi powered hub provides services to a variety of devices:

- The Jini service (service.jini) allows us to incorporate an OSGi-based gateway into a Jini network. For example, the gateway can drive a Jini printer over the local WiFi network to print out a pizza order receipt.
- The UPnP service (service.upnp) allows an OSGi-based gateway to interact with UPnP network devices.
- The HTTP service (service.http) we discussed earlier is available to any HTTP-compatible devices. For example, MIDP-based or browser-based devices on the local WiFi network can order pizza through the OSGi HTTP service on the gateway device.

- The OSGi container also provides a generic device access service (service.device) that allows developers to plug in device drivers for arbitrary devices and network protocols.

The architecture is illustrated in Figure 4.8. But still, why do we need to run gateways in OSGi containers? Wouldn't a full-blown J2EE portal server be a much more powerful option? There are two important reasons.

- Since OSGi containers run on J2ME, we can place the OSGi-based gateway in the same mobile network as the pervasive devices it serves. For example, in an in-hand network, the PDA can be the gateway; in an in-home network, the TV-set top box is the gateway; in an in-car network, the entertainment console could be the gateway. Since the local wireless network is much faster, cheaper, and easier to maintain compared with national cellular networks, the local gateways are crucial to enable high-availability mobile applications.
- The OSGi specification supports dynamic service provisioning and deployment through bundles. This is a very important feature when you have thousands of mobile gateways around in your company.

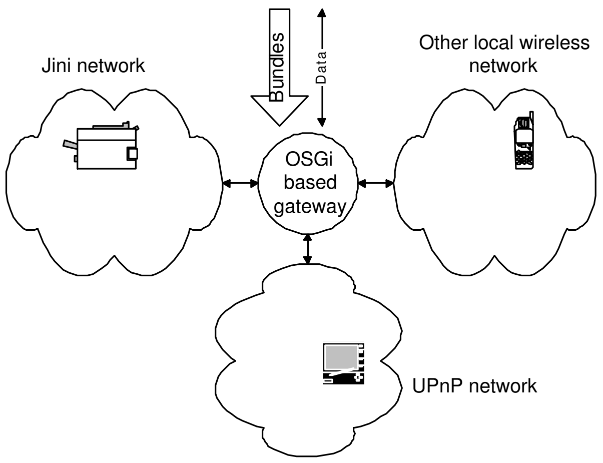


Figure 4.8. The OSGi-based local gateway architecture.

4.6 Summary

In this chapter, we discussed the benefits and architecture of managed smart clients. We introduced the OSGi specification and IBM's implementation: the SMF. Through a simple echo example, we demonstrated how to build the bundles, implement required life-cycle methods, import and export packages, expose and consume bundle services, and add UIs to a bundle application. The pizza order example shows how to reduce UI complexity and separate application layers using the available HTTP service. The pizza order application also demonstrates complex application service use and service tracking. In the last section, we briefly introduced the architectures and benefits of mobile gateways implemented over the OSGi platform.

Resources

[1] The Open Services Gateway initiative (OSGi). http://www.osgi.org/

[2] IBM WebSphere Studio Device Developer IDE (free evaluation). The page also contains a link to download the latest IBM Service Management Framework (SMF) software for free evaluation. http://www.ibm.com/embedded/

[3] The Espial DeviceTop is a clientside OSGi container with GUI support. http://www.espial.com/index.php?page=sol_devices_suite_over