SERENA

USING CHANGEMAN® ZMF
WEB SERVICES

A Practical Business Use Case with Sample Code

January 2010

SERENA™

TABLE OF CONTENTS

Executive Summary

USING THE CHANGEMAN ZMF WEB SERVICES

Serena® ChangeMan® ZMF is the leading software change management tool for z/OS mainframe systems. ChangeMan ZMF ensures reliable control of software assets and automates the migration of software changes from a development environment to multiple test and production environments.

A wide variety of interfaces are provided in ChangeMan ZMF to assist customers with using the product, integrating it with other tools in the customer development environment, and customizing its function to suit customer needs. Among these interfaces is the ChangeMan ZMF Web Services Application Programmer Interface (API), which implements the latest Web 2.0 and Service Oriented Architecture (SOA) technologies. The ZMF Web Services API was introduced in ChangeMan ZMF 6.1.

This white paper illustrates the use and implementation of the ChangeMan ZMF Web Services in a practical business application: a change package approval checklist that can be browsed from the World Wide Web. The checklist is generated on demand and shows all change packages awaiting approval by management prior to deployment into production. Any desired mix of packages may be approved at once.

Sample code is provided to illustrate the technology and to assist customers with rapid implementation of this ZMF Web Services application.

Where the ChangeMan ZMF Web Services Fit

FLEXIBLE USER ACCESS, INTEGRATION, & CUSTOMIZATION

Serena® ChangeMan® ZMF automates the migration of software changes from a development environment to multiple test and production environments. Along the way, it protects software assets by packaging changes in discrete units, versioning the modules affected by change, synchronizing the disparate objects required to implement a build, auditing the changes for possible errors, soliciting change approvals from management, deploying changes to production in a controlled way, and backing them out smoothly in the event something goes wrong. All these functions are accessible to developers from a variety of interfaces, including programmatic interfaces for customization and cross-product integration.

ISPF

User Exits

Windows GUI

Eclipse IDE

XML Services

Web Services

The user and developer interfaces to ChangeMan ZMF include:

- An interactive ISPF interface for 3270 mainframe devices or emulators
- Predefined exits in the ZMF software which may be customized with user-written code
- A Windows GUI interface to ZMF change management functions and files
- An Eclipse integrated developer environment (IDE) on Windows and Unix platforms
- An XML Services application programmer interface (API) for programmatic access to all ChangeMan ZMF functions from a local mainframe program or a TCP/IP client
- A Web Services API for programmatic access to ChangeMan ZMF functions from a client on an intranet, extranet, or the World Wide Web

CHOOSING A WEB SERVICES IMPLEMENTATION

The Web Services API was introduced in ChangeMan ZMF 6.1. Choose a ChangeMan ZMF Web Services implementation when:

- Remote access is needed to the full functionality of ChangeMan ZMF
- Connectivity is supported through an intranet, extranet, or the World Wide Web
- Custom ZMF processing or custom integration with third-party software is required
- Online transaction processing (that is, a limited number of request/response cycles serviced on demand) is suitable for performing the desired task

Web Services Technology in ChangeMan ZMF

WEB SERVICES MESSAGING

The ChangeMan ZMF Web Services API is a Web-enabled messaging layer built on top of the native XML Services API. To a service requestor on the Web, a ZMF Web Services message looks like a ZMF XML Services message wrapped in open-standard messaging syntax called SOAP (Simple Object Access Protocol).

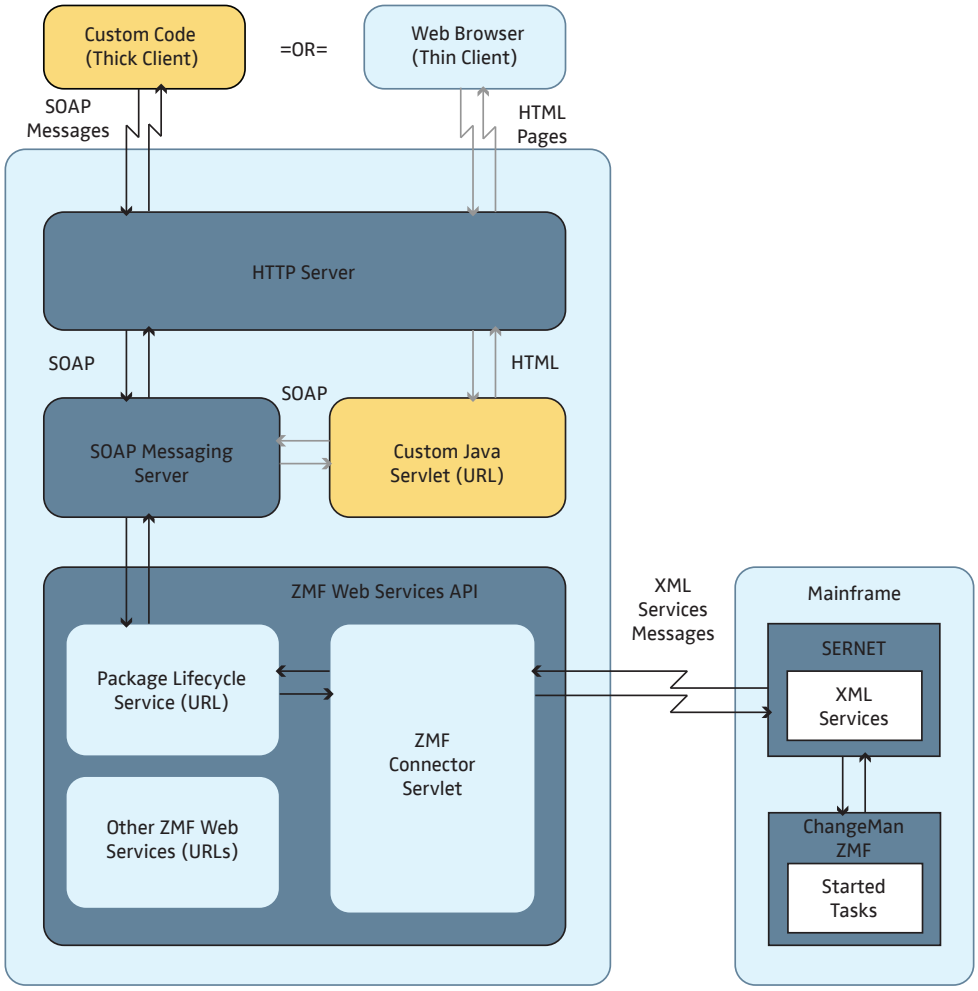


Figure 1.
ZMF Web Services
Message Flow

Web Services messages are exchanged in a simple request/response pattern between a requestor on the Web and a ChangeMan ZMF started task on the mainframe. Mediating between the requestor and responder is a Web Services application server, such as Apache Tomcat on a distributed server system or IBM WebSphere on the mainframe. The application server provides a SOAP messaging service that maps unique Web Service names to specific URLs and routes messages accordingly. The ChangeMan ZMF Web Services reside on the Web Services application server (Figure 1).

THICK CLIENT OR THIN CLIENT

Although all roads to the ZMF Web Services pass through SOAP, customer-written applications have a choice of approaches for enabling end-user access to those services. Using the “thick client” approach, a special-purpose Web client application can be written for installation on the remote PC. The thick client issues SOAP requests to the ZMF Web Services over the Web using the HTTP transport protocol. No Web browser is involved.

Alternatively, using the “thin client” approach, no special software is installed on the remote PC. Instead, the end-user browses to a designated URL on the Web Services application server using a standard Web browser. The landing page for the URL contains

a server-side dynamic element that invokes a customer-written servlet. This servlet, which resides on the application server, sends HTML pages to the remote end-user and responds to user actions by issuing SOAP requests to the ZMF Web Services API — either locally or on another application server. The ZMF Web Services API is agnostic regarding the choice of thick or thin clients; it handles either.

When a ZMF Web Service receives a SOAP message, it validates the message, then passes it to a back-end ZMF connector servlet. The connector servlet manages the TCP/IP link between the application server and SERNET, a connectivity component of ChangeMan ZMF on the mainframe. On an inbound request message, the servlet transforms the SOAP message into XML Services syntax and forwards the request to SERNET. SERNET parses the XML request and schedules the desired service for execution by the desired ZMF started task.

ChangeMan ZMF returns the outcome of request execution to SERNET. SERNET formats an appropriate XML Services response message, then routes it to the back-end ZMF connector servlet on the Web Services application server. On an outbound message, the servlet wraps the XML Services response in SOAP message syntax and returns it to the appropriate ZMF Web Service. That service, in turn, forwards the SOAP response to the appropriate requestor.

ACCESSING WEB SERVICE FUNCTIONS

Each Web Service is identified by a unique service name in the SOAP message header. This name is bound to a port number on the application server when the Web Service is installed. (All Web Services on the same application server share the same IP address.) Mapping between service name and port ID is handled by the SOAP messaging server at run time.

Each Web Service supports multiple functions. For example, the package lifecycle management service supports separate functions to create a package, freeze a package, and approve a package. The particular function desired is identified by a named method in the SOAP message header. Method names are mapped to executable entry points by the Web Service.

Seventeen Web Services are currently provided in the ChangeMan ZMF Web Services API. These services cover all the developer functionality of ChangeMan ZMF, as well as the basic connectivity and dataset management functions supported by SERNET. Some ChangeMan ZMF administrator functions are supported as well in read-only mode.

Web Services Use Case: A Software Change Approval Checklist

The following example shows how you might build a custom application that invokes the ChangeMan ZMF Web Services using the “thin client” approach.

BUSINESS AS USUAL: SOFTWARE CHANGE APPROVALS

At Fearless Financial Group, no change to mainframe production software takes place without the prior approval of the IT Operations Manager. ChangeMan ZMF enforces this requirement automatically. When approval is needed for deployment of a change package into production, ZMF issues an approval request notice for that package by email. Deployment of the change is placed on hold until all required approvals are received. At that point deployment is allowed to go forward on a date designated in the ZMF installation calendar.

The standard notification and approval process is fine for most purposes. However, it has limitations. For example, when approval request notifications are received one package at a time, the big picture of overall deployment activity is unclear at the moment the approval decision is made. In addition, deployment calendar rules do not recognize variations in package complexity or risk, and they take no account of the people on hand to oversee deployments.

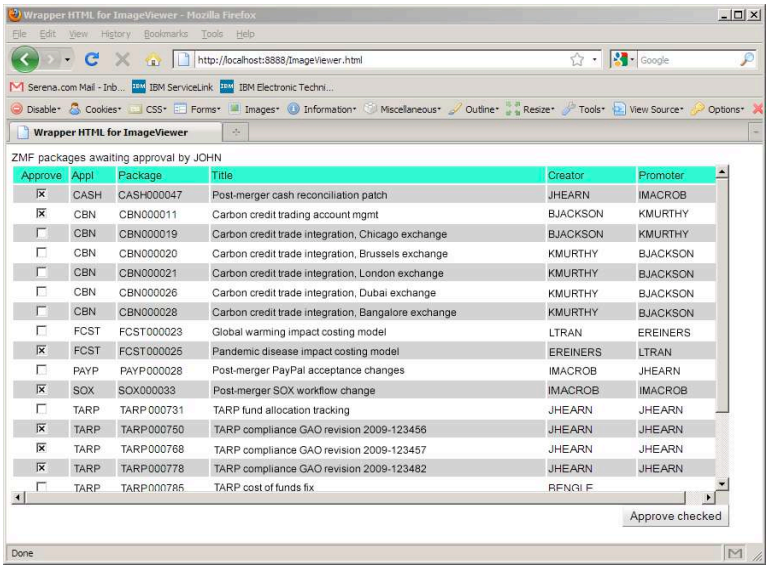
WEB SERVICES SOLUTION: AN ON-DEMAND SOFTWARE APPROVAL CHECKLIST

The Fearless IT Operations Manager would like to have a software change approval tool tailored for use by senior managers. This tool would show him the big picture of pending deployments before he approves a change package. It would also allow him to approve a judicious mix of packages all at once. With such a tool, the Operations Manager could use his approvals to:

- Smooth out peaks and valleys in scheduled software change activities
- Match change package content to the availability of expert personnel at install time
- Minimize competing deployment activity during rollouts of complex or high-risk changes

A custom application using ZMF Web Services is written to meet these requirements (Figure 2).

Figure 2.
ZMF Web Services
Package Approval Checklist



Approve	Appl	Package	Title	Creator	Promoter
☒	CASH	CASH000047	Post-merger cash reconciliation patch	JHEARN	IMACROB
☒	CBN	CBN000011	Carbon credit trading account mgmt	BJACKSON	KMURTHY
☐	CBN	CBN000019	Carbon credit trade integration, Chicago exchange	BJACKSON	KMURTHY
☐	CBN	CBN000020	Carbon credit trade integration, Brussels exchange	KMURTHY	BJACKSON
☐	CBN	CBN000021	Carbon credit trade integration, London exchange	KMURTHY	BJACKSON
☐	CBN	CBN000026	Carbon credit trade integration, Dubai exchange	KMURTHY	BJACKSON
☐	CBN	CBN000028	Carbon credit trade integration, Bangalore exchange	KMURTHY	BJACKSON
☐	FCST	FCST000023	Global warming impact costing model	LTRAN	EREINERS
☒	FCST	FCST000025	Pandemic disease impact costing model	EREINERS	LTRAN
☐	PAYP	PAYP000028	Post-merger PayPal acceptance changes	IMACROB	JHEARN
☒	SOX	SOX000033	Post-merger SOX workflow change	IMACROB	IMACROB
☐	TARP	TARP 000731	TARP fund allocation tracking	JHEARN	JHEARN
☒	TARP	TARP000750	TARP compliance GAO revision 2009-123456	JHEARN	JHEARN
☒	TARP	TARP000768	TARP compliance GAO revision 2009-123457	JHEARN	JHEARN
☒	TARP	TARP000778	TARP compliance GAO revision 2009-123482	JHEARN	JHEARN
☐	TARP	TARP000785	TARP cost of funds fix	BENGLE	

The custom application adopts a “thin client” approach, so there is no special software for the Operations Manager to install or use. Whenever the Operations Manager wishes to see a list of software change packages awaiting his approval, he browses to the URL of a normal HTML Web page that functions as a package checklist viewer.

The viewer displays a package approval checklist table and invokes a Java servlet to populate it. The Java servlet uses the ZMF Web Services to request a list of all software change packages in ChangeMan ZMF that are currently waiting for the manager’s approval. A list of pending packages is returned by ZMF to the servlet, which passes them to the HTML page for display. The Operations Manager checks the packages he wishes to approve on the Web page, then clicks the Approve Checked button. On this click, Javascript on the HTML viewer page passes a list of the approved packages to the Java servlet, which generates package approval request messages for them and sends those messages in a batch to ChangeMan ZMF. When ZMF records the approvals, the packages are ready for deployment into production.

Implementing the Package Approval Checklist

Sample code for the Package Approval Checklist in Figure 2 is available for download from Serena at http://www.serena.com/docs/repository/products/zmf/ZMF_Web_Svcs_Approval_Chklst_Sample_Code.zip. Portions of that code are shown here for illustration.

PACKAGE APPROVAL CHECKLIST VIEWER

Code to build and display the checklist table and handle the **Approve Checked** button can be found in the download file ZMFSample.java, which resides in the zmf > sample > client directory of the downloadable code sample. This code is written using the Google Web Toolkit (GWT), which is available at http://code.google.com/webtoolkit/. The package checklist viewer does not itself call the ZMF Web Services, so it is not shown here.

PACKAGE APPROVAL CHECKLIST PROCESSING SERVLET

The following ZMF Web Services are needed to implement the package approval checklist and perform the listed functions:

- **Connect** — mainframe logon and logoff
- **Package Search and Summary** — search for packages awaiting approval by manager
- **Package Lifecycle** — approve packages
- **Developer Environment Administration** — define valid approval codes

The first three services — Connect, Package Search and Summary, and Package Lifecycle — are invoked directly by Java servlet PackageManagementImpl.Java. The checklist viewer invokes this servlet to populate the checklist table with the results of a package search, and to submit a list of package approval messages to ZMF. Sample code for this servlet is shown in Figure 3. Code for this member can also be found in the `zmf > sample > server` directory of the downloadable code sample.

The checklist processing servlet could be supplemented in several ways by calling additional ZMF Web Services. For example, it could display the scheduled install dates for the packages awaiting manager approval in the checklist. More elaborately, the application could supplement the package approval checklist view with a deployment calendar view that shows all approved packages scheduled for installation over the next week or month. The Operations Manager could then toggle between the package approval checklist view and the deployment calendar view from within the HTML viewer page. For simplicity, however, such options are not included in this example.

Figure 3.
Package Approval Checklist
Processing Servlet

```
package com.serena.zmf.sample.server;

import java.util.LinkedList;

import com.serena.zmf.sample.client.PackageManagement;
import com.serena.zmf.sample.client.common.ApprovalInfo;
import com.serena.zmf.sample.client.common.PackageInfo;
import com.serena.zmf.webservices.client.internal.PackageLifeCycle.ApprovePackageRequest;
import com.serena.zmf.webservices.client.internal.PackageLifeCycle.ApproverAction;
import com.serena.zmf.webservices.client.internal.PackageSearchSummary.TypeYesNo;
import com.serena.zmf.webservices.client.internal.Connect.LogonRequest;
import com.serena.zmf.webservices.client.internal.PackageSearchSummary.QueryPackagesRequest;
import com.serena.zmf.webservices.client.internal.PackageSearchSummary.QueryPackagesResults;
import com.serena.zmf.webservices.client.internal.PackageSearchSummary.QueryPackagesResultsResult;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;

@SuppressWarnings(“serial”)
public class PackageManagementImpl extends RemoteServiceServlet implements
      PackageManagement {

   /*************************************************************************************
    * Logon with fixed userid and password to fixed ZMF started task. In real life, a
    * more flexible method would be used.
    *************************************************************************************/
   private void logon(ZMFServer zServer) {

      LogonRequest request = new LogonRequest();

      request.setUser(“someUser”);
      request.setPassword(“somePassword”);
      request.setNewpassword(null);
      request.setSubsystemid(“someZMFSubsysID”);
      request.setVersion(“610”);
      request.setHost(“someHost”);
      request.setPortid(“6011”);

      zServer.logon(request);
      /*
       * Note we have no error checking here. We should make sure we can actually log in.
       */
   }

   public QueryPackagesResults getPackagesForApprover(ZMFServer zServer, String entity) {

      QueryPackagesRequest request = new QueryPackagesRequest();
      request.setApprovalEntity(entity);
      request.setSearchForApprovalPending(TypeYesNo.Y);
      request.set_package(“*”);
      QueryPackagesResults results = zServer.getPackages(request);
      return results;
   }
```

```
   /*****************************************************************************************
    * Get packages awaiting approval by entity “someUser”.

    *
* The code retrieves the matching packages, creates a linked list of transport
    * objects and converts those that to an array (a linked list would have
    * worked just as well as a linked list is GWT serializable.
    *
    * (non-Javadoc)
    *
    * @see com.serena.zmf.sample.client.PackageManagement#getPackages()
    *****************************************************************************************/
   @Override
   public PackageInfo[] getPackages() {
      ZMFServer zServer = new ZMFServer();
      logon(zServer);
      QueryPackagesResults results = getPackagesForApprover(zServer, “someUser”);
      zServer.logoff();
      LinkedList<PackageInfo> packages = new LinkedList<PackageInfo>();
      if (results != null) {
         for (int i = 0; i < results.getResult().length; i++) {
            QueryPackagesResultsResult queryResult = results.getResult(i);
            PackageInfo queryPackage = new PackageInfo(queryResult.get_package(),
                  queryResult.getApplName(),
                  queryResult.getCreator(),
                  queryResult.getLastPromoter(),
                  queryResult.getPackageTitle(),
                  queryResult.getPackageStatus().toString());
            packages.add(queryPackage);
         }
      }
      return packages.toArray(new PackageInfo[1]);
   }
   /*****************************************************************************************
    * Approve every package passed from the frontend in the ApprovalInfo array.
    * @see com.serena.zmf.sample.client.PackageManagement
    * #approvePackages(com.serena.zmf.sample.client.common.ApprovalInfo[])
    *****************************************************************************************/
   @Override
   public Integer approvePackages(ApprovalInfo[] approvalInfo) {
      ZMFServer zServer = new ZMFServer();
      logon(zServer);
      for (ApprovalInfo approval : approvalInfo)  {
         ApprovePackageRequest request = new ApprovePackageRequest();
         request.set_package(approval.getPackageID());
         request.setApplName(approval.getAppl());
         request.setApproverEntity(approval.getEntity());
         request.setApproverAction(ApproverAction.value1);
         zServer.approvePackage(request);
      }
      zServer.logoff();
      return null;
   }
}
```

The servlet imports additional code, not shown here, for message handling. In the downloadable code sample, transport data classes are defined in the `zmf > sample > client >` common directory. Web Services RPC calls are serialized and deserialized in synchronous transmissions by member PackageManagement.java, and in asychronous transmissions by member PackageManagementAsync.java. These members can be found in the downloadable code directory `zmf > sample > client`.

ZMF WEB SERVICES INTERFACE

The PackageManagementImpl.Java viewer processing servlet calls `ZMFServer.java` to connect to the actual ZMF Web Services. This code also maintains session state for the connection to ChangeMan ZMF. Sample code for `ZMFServer.java` is shown in Figure 4. This code also resides in the `zmf > sample > server` directory of the downloadable code sample.

Each ZMF Web Service needed by the package approval checklist servlet is accessed through "stub" code shown below. One SOAP message binding stub is provided for each Web Service.

Figure 4.
ZMF Web Services Interface

```
package com.serena.zmf.sample.server;

import java.net.URL;
import java.rmi.RemoteException;

import com.serena.zmf.webservices.client.internal.Connect.*;
import com.serena.zmf.webservices.client.internal.DeveloperEnvironmentAdmin.*;
import com.serena.zmf.webservices.client.internal.PackageLifeCycle.*;
import com.serena.zmf.webservices.client.internal.PackageSearchSummary.*;

import org.apache.axis.client.Stub;
import org.apache.axis.transport.http.HTTPConstants;
import javax.xml.rpc.ServiceException;

/*****************************************************************************
 * The code below represents a basic interface to the ZMF web services required
 * to search for eligible packages and then to approve them.
 *****************************************************************************/
public class ZMFServer {

   // Host that the ZMF web server resides on (Tomcat or similar).
   private String webServicesHost = "localhost";
   // Port that ZMF web services are listening on
   private String webServicesPort = "8080";
   // We need a cookie to maintain session information after having authenticated
   private String cookie = null;
   // Name of logged in user
   private String user = null;

   public ZMFServer() {
   }

   public ZMFServer(String host, String port) {
      this.webServicesHost = host;
      this.webServicesPort = port;
   }

   /*****************************************************************************
    ** Login
    *****************************************************************************/
   public synchronized com.serena.zmf.webservices.client.internal.Connect.Response
   logon(LogonRequest request) {

      ConnectSOAPBindingStub binding = null;

      try {
         // Get the stub which implements the SDI.
         ConnectServicesLocator csl = new ConnectServicesLocator();
```

```
        // Update URL
        updateURL(csl);

        binding = (ConnectSOAPBindingStub) csl.getConnect();
        binding.setMaintainSession(true);

    } catch (ServiceException e) {
        return null;
    }

    Logon logon = new Logon(request);
    com.serena.zmf.webservices.client.internal.Connect.Response response = null;

    try {
        response = binding.logon(logon);
    } catch (RemoteException e) {
        return null;
    }

    // store cookie to maintain state

    cookie = (String) binding._getCall().getMessageContext().getProperty(
          HTTPConstants.HEADER_COOKIE);

    user = request.getUser();
    return response;
}
/*****************************************************************************
 * Logoff
 *****************************************************************************/
public synchronized void logoff() {
    ConnectSOAPBindingStub binding = this.getConnectSOAPBindingStub();

    // Create connect logoff request.

    LogoffRequest logoffReq = new LogoffRequest(user);
    Logoff logoff = new Logoff(logoffReq);

    // logoff and destroy cookie

    try {
        binding.logoff(logoff);
    } catch (RemoteException e) {
        cookie = null;
    }

    cookie = null;
}
/*****************************************************************************
 * Get Global Parameters
 *****************************************************************************/
public GetGlobalParmsResults getGlobalParms(GetGlobalParmsRequest request) {
    DeveloperEnvironmentAdminSOAPBindingStub binding =
        getDeveloperEnvironmentAdminSOAPBindingStub();
    GetGlobalParmsResults results = null;
    GetGlobalParms getGlobalParms = new GetGlobalParms(request);
    try {
        results = binding.getGlobalParms(getGlobalParms);
    } catch (RemoteException e) {
    }

    return results;
}
```

```
/*****************************************************************************************
 * Update connection address with values from preference store.
     *****************************************************************************************/
private void updateURL(ConnectServicesLocator csl) {
   String strURL = updateURL(csl.getConnectAddress());
   csl.setConnectEndpointAddress(strURL);
}

private String updateURL(String strURL) {

   try {
      URL url = new URL(strURL);

      String strProtocol = url.getProtocol();
      String strFile = url.getFile();
      url.getRef();
      Integer nPort = new Integer(webServicesPort);

      URL url2 = new URL(strProtocol, webServicesHost, nPort
            .intValue(), strFile);
      return url2.toString();

   } catch (Exception e) {
   }

   return strURL;
}

// Maintain Session

private void setCookieAndMaintainSession(Stub binding) {
   binding.setMaintainSession(true);
   binding._setProperty(HTTPConstants.HEADER_COOKIE, cookie);
}

// *****************************************************************************************
// Stub Access....
// *****************************************************************************************

// Get ConnectSoapBindingStub

private ConnectSOAPBindingStub getConnectSOAPBindingStub() {
   ConnectSOAPBindingStub binding = null;
   try {
      ConnectServicesLocator sl = new ConnectServicesLocator();
      sl.setConnectEndpointAddress(updateURL(sl.getConnectAddress()));
      binding = (ConnectSOAPBindingStub) sl.getConnect();
      setCookieAndMaintainSession(binding);
   } catch (javax.xml.rpc.ServiceException jre) {
   }
   return binding;
}

// Get PackageLifeCycleSOAPBindingStub
private PackageLifeCycleSOAPBindingStub getPackageLifeCycleSOAPBindingStub() {

   PackageLifeCycleSOAPBindingStub binding = null;
   try {
      PackageLifeCycleServicesLocator sl = new PackageLifeCycleServicesLocator();
      sl.setPackageLifeCycleEndpointAddress(updateURL(sl
```

```
                .getPackageLifeCycleAddress()));
          binding = (PackageLifeCycleSOAPBindingStub) sl
                .getPackageLifeCycle();
          setCookieAndMaintainSession(binding);
       } catch (javax.xml.rpc.ServiceException jre) {
       }
       return binding;
}
   /**********************************************************************************
    * Below are three SOAP stubs. There is one stub per web service group. The example
    * shows only those stubs which are actually used in this example.
    *********************************************************************************/
   // Get PackageSearchSummarySOAPBindingStub
   private PackageSearchSummarySOAPBindingStub getPackageSearchSummarySOAPBindingStub() {
      PackageSearchSummarySOAPBindingStub binding = null;
      try {
         PackageSearchSummaryServicesLocator sl =
                new PackageSearchSummaryServicesLocator();
         sl.setPackageSearchSummaryEndpointAddress(updateURL(sl
                .getPackageSearchSummaryAddress()));
         binding = (PackageSearchSummarySOAPBindingStub) sl
                .getPackageSearchSummary();
         setCookieAndMaintainSession(binding);
      } catch (ServiceException jre) {
      }
      return binding;
   }
   // Get DeveloperEnvironmentAdminSOAPBindingStub
   private DeveloperEnvironmentAdminSOAPBindingStub
                getDeveloperEnvironmentAdminSOAPBindingStub() {
      DeveloperEnvironmentAdminSOAPBindingStub binding = null;
      try {
         DeveloperEnvironmentAdminServicesLocator sl =
                new DeveloperEnvironmentAdminServicesLocator();
         sl.setDeveloperEnvironmentAdminEndpointAddress(updateURL(sl
                .getDeveloperEnvironmentAdminAddress()));
         binding = (DeveloperEnvironmentAdminSOAPBindingStub) sl
                .getDeveloperEnvironmentAdmin();
         setCookieAndMaintainSession(binding);
      } catch (ServiceException jre) {
         // throwCoreException(jre);
      }
      return binding;
   }
   /**********************************************************************************
    * Find packages matching a particular criterion. There is a great deal of search
    * criteria which could be specified here.
    *********************************************************************************/
   public QueryPackagesResults getPackages(QueryPackagesRequest request) {
      QueryPackagesResults results = null;
      PackageSearchSummarySOAPBindingStub binding = this
             .getPackageSearchSummarySOAPBindingStub();
      QueryPackages sfp = new QueryPackages(request);

      try {
         results = binding.queryPackages(sfp);
      } catch (RemoteException e1) {
      }
      return results;
   }
```

```
   /**************************************************************************
    * Approve a package. The request defines the package, in particular name,
    * application and entity.
    **************************************************************************/
   public void approvePackage(ApprovePackageRequest request) {
      PackageLifeCycleSOAPBindingStub binding = this
            .getPackageLifeCycleSOAPBindingStub();
      ApprovePackage ap = new ApprovePackage(request);

      try {
         binding.approvePackage(ap);
      } catch (RemoteException e1) {
      }
   }
}
```

Obtaining the ChangeMan ZMF Web Services API

The ChangeMan ZMF Web Services API is provided at no extra charge on the product media. When you unload the product media prior to transfering the host software to z/OS, the Web Services API code is saved to the following location on the installation PC:

```
c:\Program Files\Serena\ChangeMan ZMF\v.r.m\WebServices
```

where v is the version number, r is the release, and m is the modification level of ZMF. You must install this code separately on the desired Web Services application server.

Licensed customers may download the ChangeMan ZMF product media, including the Web Services API, from the Serena Customer Support Web site at http://support.serena.com.

Further information about the ZMF Web Services are available in the ChangeMan ZMF Web Services Getting Started Guide. This document may also be downloaded from Customer Support.

CONTACT

Website: www.serena.com
Phone: 800-547-7827
Email: info@serena.com