LegStar User Guide

LegStar User Guide

1.5.3

Table of Contents

List of Figures

Chapter 1. Introduction

What is LegStar?

LegStar provides development and runtime features for developers who need to integrate with mainframe programs such as those written in COBOL.

Unlike other integration solutions available, LegStar is free and open-source. It leverages the ever-increasing power of open-source software by using familiar programming patterns (visitor, strategy…), frameworks (JAXB, JAX-WS), tools (Apache Ant, Eclipse) and targeting widely used J2EE servers such as Apache Tomcat and Jetty or Enterprise Service Buses.

Features overview

Development time features

Activities involved in mainframe integration usually require:

- Mapping mainframe data structures to open world constructs such as Java classes or XML Schema.
- Mapping mainframe programs to open world processes (Web Service operations, Java methods, …).

These mapping activities occur at development time and usually produce meta-data that can later be used by runtime engines to flow requests and data between the mainframe application and the open world.

At the core of LegStar is an XML schema to COBOL binding language. This is similar in spirit to the Java to XML Schema binding language introduced by the JAXB standard. This COBOL binding language materializes as XML schema annotations or Java annotations.

LegStar COBOL binding language tries to cover all the real issues facing integration developers such as how to map COBOL weakly typed variables to Java strongly typed ones, deal with complex "REDEFINES", variable size arrays, code page conversions, numeric conversions and support for multiple input/output programs (CICS Containers).

LegStar provides tooling to support data and process mapping activities. These tools are provided as ant scripts but they are also available as a rich set of plug-ins for the Eclipse platform.

Runtime features

Runtime mainframe integration activities can be separated into:

- Data binding activities where data streams in mainframe format are transformed to, or from, open world objects such as Java classes or XML.
- Remote Procedure Call activities where mainframe programs are invoked, or call outbound open world processes.

LegStar provides runtime capabilities for IBM CICS, where CICS programs act either as servers, serving requests coming from the open world or as clients, calling remote Web or Java services.

The LegStar various modules are only loosely coupled and can be used in a large number of scenarios. For instance, one can use the data binding capabilities without using the LegStar RPC mechanisms.

Integration targets are not limited to Web Services. There are various projects using LegStar to integrate directly with major ESBs or ETLs for instance.

Common use cases

The easiest way to present the LegStar architecture is to show how it supports two common integration use cases:

1. An existing mainframe program, say a COBOL CICS program, needs to be exposed as a Web Service.

2. A mainframe application needs to execute a remote Web Service.

The first use case is very common but the importance of the second one is growing rapidly as legacy subsystems are being replaced by new applications running on J2EE and .Net platforms.

There is a large number of variations on these 2 main use cases, for instance developers might need to expose legacy functionalities as REST rather than plain Web Services, or developers might need to map complex structures to Java objects rather than XML. Developers might need to describe new structures in XML schema and then map these to Java and COBOL in support for two parallel developments (rather than integration), etc. LegStar is modular so that features can be selected and combined as necessary.

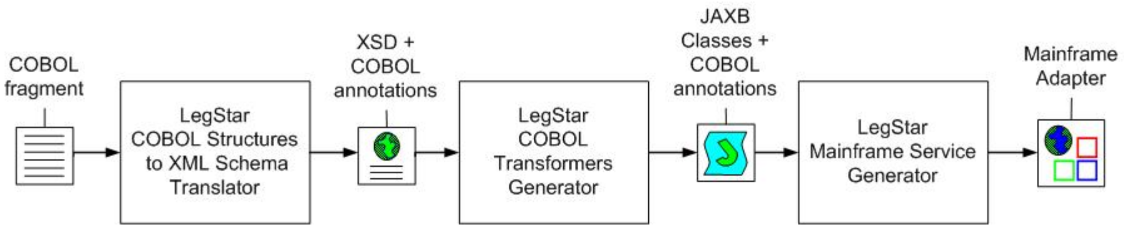
Chapter 2. Architecture

Expose a COBOL program as a Web Service

Development tools

With LegStar, developers would follow these three steps to *Service-enable* a COBOL program:

Figure 2.1. Service adapter development steps


COBOL fragment
LegStar COBOL Structures to XML Schema Translator
XSD + COBOL annotations
LegStar COBOL Transformers Generator
JAXB Classes + COBOL annotations
LegStar Mainframe Service Generator
Mainframe Adapter

In this use case, initial COBOL code fragments describe the legacy program input and output structures.

The LegStar *COBOL Structures to XML Schema Translator* takes a COBOL fragment as input and creates an XML Schema with COBOL annotations. This generated XML Schema is known as a Mapping XML Schema since the COBOL annotations form the meta-data that maps each COBOL data item to an XML element type.

Some COBOL programs accept several input structures and several output structures, each described by a different code fragment. This step can therefore be repeated as necessary for each COBOL code fragment involved.

The generated XML Schema can be customized, and further annotated, by developers using standard XML Schema editors. In particular, developers can specify custom processing to deal with complex decisions related with COBOL REDEFINES for instance.

The LegStar *COBOL Transformers Generator* takes a Mapping XML Schema as input and produces a set of Java Classes. These generated classes are all that is needed for the runtime to perform Java, XML or JSON to mainframe data marshaling/unmarshaling.

The meta data that binds java fields to COBOL data items is stored as java annotations in a set of JAXB classes.

At runtime, mainframe data described by the initial COBOL fragment is typically encoded in a mainframe character set (EBCDIC) and contain compressed numerics and other mainframe specific formatted data. Conversion is totally bi-directional and completely independent of the origin or destination of the host data.

The LegStar *Mainframe Service Generator*, maps a mainframe program to a Web Service operation or Java method. The current version of LegStar supports CICS programs either Commarea or Container driven.

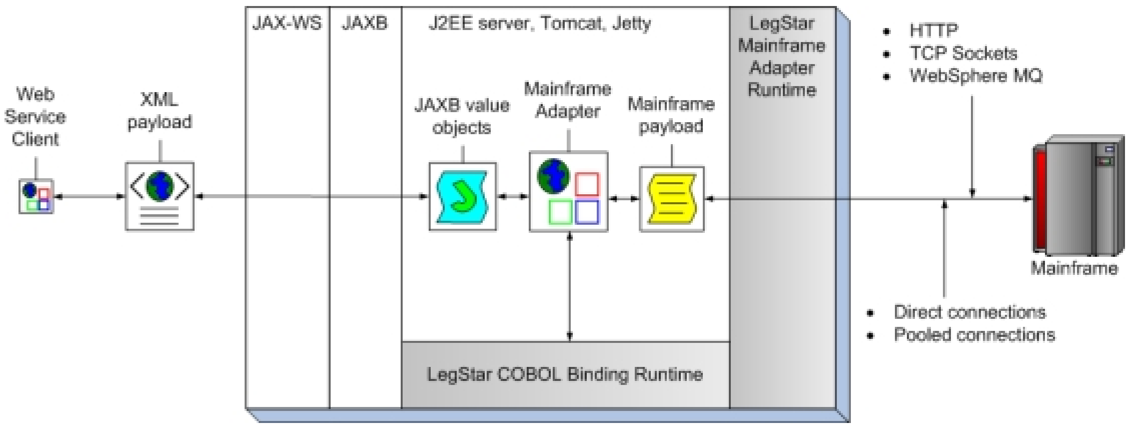
It is important to note that the tools behind each step are completely independent from each other. For instance, without an initial COBOL code fragment, developers could start from an XML Schema, edit the XML Schema to add COBOL binding annotations and then continue the remaining steps. This is not an uncommon use case, where the mainframe program is actually new and the starting point is an XML schema (An approach sometimes referred to as Contract-first).

For each of these steps, LegStar provides both Ant scripts and Eclipse plug-ins. Moving forward, the Eclipse plugins are the recommended tool as the parameter set needed by generators keeps increasing.

Runtime services

From a runtime perspective, this is how a request/reply message exchange would flow in an IBM CICS environment:

Figure 2.2. Service adapter runtime



Starting from a Web Service client, SOAP requests are first processed by a standard Web Service stack. The LegStar-generated endpoint uses the JAX-WS standard API to communicate with the SOAP stack.

The XML payload extracted from SOAP requests is handed over by JAX-WS to the standard JAXB binding framework, which uses the LegStar-generated JAXB classes to parse the XML and produce a value object. The adapter endpoint implementation uses the LegStar *COBOL Binding Runtime* to transform java value objects to a mainframe payload. This transformation includes Unicode to EBCDIC conversions, numeric conversions, REDEFINES decision-making, etc…

Once data is in mainframe format, the endpoint uses a transport independent layer called LegStar *Mainframe Adapter Runtime* to invoke a remote program.

The LegStar *Messaging Protocol* used by the Mainframe Adapter Runtime is binary and implements a request/reply exchange pattern. It is designed to reduce the payload size and supports multiple input/output named structures such as CICS containers. Alternatively, the adapter runtime can use transport specific messaging protocols such as IBM CICS MQ Bridge.

The actual transport is selected at runtime from a configuration file. The following options are available:

- Socket transport
- HTTP transport
- WebSphere MQ transport

When needed, LegStar provides z/OS modules, written in C/370 for CICS, to handle messaging on the Mainframe side (Not needed with CICS Websphere MQ Bridge). The CICS footprint of this architecture is minimal since all SOAP/XML processing occur off-host.

Any of the available transports can be used in a direct or pooled fashion. Pooling of connections, offered by LegStar *Connection Pooling Engine*, allows efficient connection reuse and enhances performances.

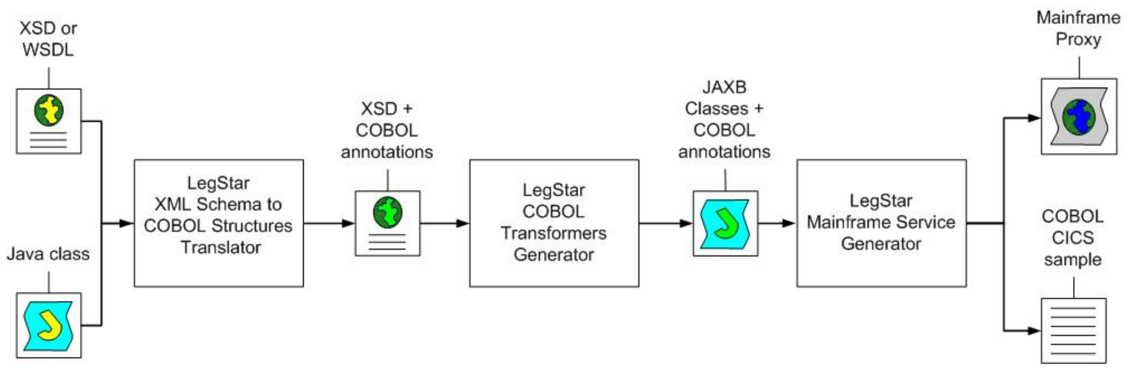
On the way back, mainframe data is converted to XML and then wrapped in a SOAP reply.

Consume a Web Service or POJO from a COBOL program

Development tools

There are three steps to achieve outbound access to Web Services or POJOs:

Figure 2.3. Service proxy development steps


XSD or WSDL
Java class
LegStar XML Schema to COBOL Structures Translator
XSD + COBOL annotations
LegStar COBOL Transformers Generator
JAXB Classes + COBOL annotations
LegStar Mainframe Service Generator
Mainframe Proxy
COBOL CICS sample

The LegStar *Xml Schema to COBOL Translator* takes a WSDL or XML Schema file as input and produces a Mapping XML Schema with COBOL annotations. Alternatively, it can use pure Java objects as input when the target is a POJO rather than a Web Service.

The developer will typically edit the resulting XML schema to adjust such things as COBOL string sizes or maximum array sizes, which cannot always be inferred from XML Schema.

The second step, using the LegStar *COBOL Transformers Generator* is the same tool used for adapters. The result is a set of java classes, which provides the conversion capabilities from mainframe data to XML, JSON or Java.

The third step, the LegStar *Mainframe Service Generator*, also used for adapters, is used here to produce a Mainframe Proxy and COBOL CICS sample program. The Mainframe Proxy acts as an intermediary between the mainframe client program and the target Web Service or POJO at runtime. It can be deployed as a Servlet.

The COBOL CICS sample program generated can be used to jump start your own mainframe client programs.

Runtime services

LegStar provides a *Mainframe Proxy Runtime* to support incoming requests from the mainframe. The HTTP Transport is the only one available at the moment.

On the mainframe side, you can use CICS DFHWBCLI or EXEC WEB API to send the payload to the Mainframe Proxy Runtime. All that is required is that the request is an HTTP POST and that the body is binary (not translated to ASCII) signaled by an application/octet-stream MIME type.

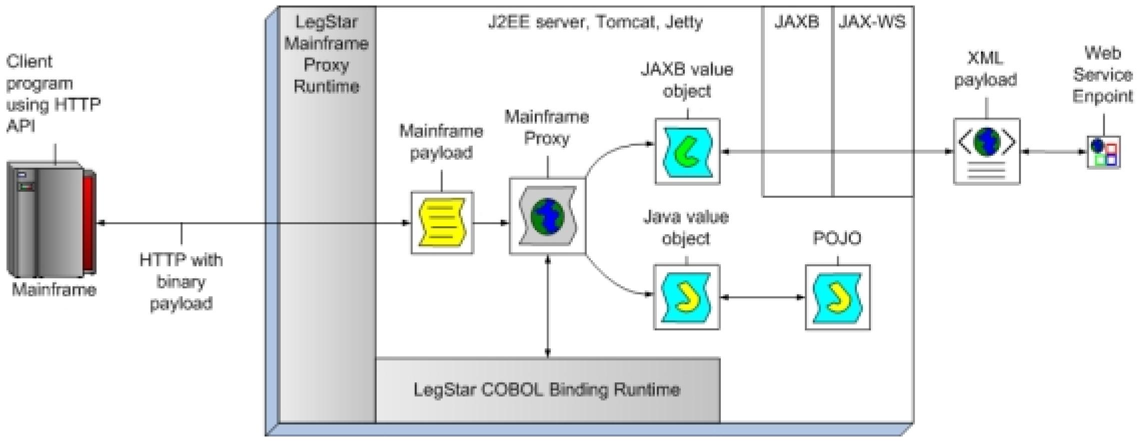
For older versions of CICS without DFHWBCLI or EXEC WEB API, LegStar provides a simple HTTP client API written in C/370.

The CICS program does not directly call the target Web Service/POJO. Rather, the generated Mainframe Proxy receives the request, which is still in host (EBCDIC) format at this stage. Again, no conversion occurs on the host significantly reducing the mainframe footprint of this solution.

Transformation from mainframe format to XML is performed by the LegStar *COBOL Binding Runtime*.

The request would flow as depicted in the following diagram:

Figure 2.4. Service proxy runtime



The Mainframe Proxy uses the standard JAX-WS Client API to perform the call to the target Web Service. Alternatively, when the target is a POJO, the Proxy invokes the POJO method directly. Observe that in the case of POJOs, there is no need for JAXB at runtime, the only constraint is that the Proxy be able to locate the POJO in the J2EE server classpath.

Chapter 3. Installation

Installing LegStar

Pre-requisites

Java 1.5+ and ANT 1.6.5+ are both prerequisites for LegStar.

LegStar requires a JDK, or an international version of the JRE, that includes charsets.jar.

Make sure JAVA_HOME and ANT_HOME environment variables are set and that $JAVA_HOME/bin (%JAVA_HOME%/bin on Windows) and $ANT_HOME/bin (%ANT_HOME%/bin on Windows) are both in your system path.

The mainframe Service Generator requires JAX-WS 2.1 (JSR 224). The Sun's JAX-WS reference implementation is shipped with LegStar. You can replace this implementation with any JAX-WS 2.1 compliant provider.

Since Service Generator is modular, there might be more prerequisites depending on your target choice (Adapter or Proxy) and, if you chose Adapters, what transport you want to use. For each of these choices, documentation is available either online or in the docs folder.

Installing

Warning: make sure you completely uninstall any previous version before proceeding with a new install.

Download the latest distribution [http://www.legsem.com/legstar/legstar-distribution].

Unzip the binary distribution package into the directory of your choice, referred to as <installDir> in the following steps.

The directory tree should look like this:

```
<installDir>
 |-->LICENSE
 |-->readme.html
 |--><docs>
     |-->*-README
 |--><lib>
     |-->*.jar
 |--><samples>
     |--><quickstarts>
         |--><adapter_lsfileae>
             |-->build-*.xml
             |--><cobol>
                 |-->lsfileae
             |--><schema>
                 |-->lsfileae.xsd
             |--><src>
```

```
                |-->**/*.java
        |--><chttprt>
            |-->build.xml
            |-->legstar-invoker-config.xml
            |-->lsfileae.properties
            |-->readme.txt
            |--><src>
        |--><cmockrt>
            |-->legstar-invoker-config.xml
            |-->readme.txt
        |--><cmqrt>
            |-->build.xml
            |-->legstar-invoker-config.xml
            |-->lsfileae.properties
            |-->readme.txt
            |--><src>
        |--><csokrt>
            |-->build.xml
            |-->legstar-invoker-config.xml
            |-->lsfileae.properties
            |-->readme.txt
            |--><src>
        |--><pooling>
            |-->legstar-engine-config.xml
            |-->legstar-pooling-config.xml
        |--><proxy_pojo_jvmquery>
            |-->build-*.xml
            |--><java>
                |-->*.jar
            |--><jcl>
                |-->COBCJVMQ
            |--><schema>
                |-->jvmquery.xsd
            |--><src>
                |-->**/*.java
        |--><proxy_ws_cultureinfo>
                |-->build-*.xml
                |--><jcl>
                    |-->COBCCULT
                |--><schema>
                    |-->cultureinfo.xsd
                |--><src>
                    |-->**/*.java
                |--><webapp>
                    |--><jaxws-cultureinfo>
                        |--><WEB-INF>
                            |-->web.xml
                            |-->sun-jaxws.xml
|--><war>
    |-->legstar-engine.war
|--><zos>
    |--><C370>
        |-->*.C
        |-->*.H
```

```
|--><cobol>
    |-->*.cbl
|--><JCL>
    |-->*
|--><docs>
    |-->*-README
```

if you plan on using the Service Generation capabilities, copy the content of <installDir>/lib to your J2EE container shared libraries (ex $CATALINA_BASE/shared/lib).

Uninstalling

To uninstall, remove the <installDir> folder.

Running the ant samples

Service Adapter

From the samples/quickstarts/adapter_lsfileae folder, run command *ant -f build-cob2xsd.xml*. This should create a schema folder with a generated XML schema from the sample cobol source in the cobol folder.

Run command *ant -f build-coxb.xml* from the samples/quickstarts/adapter_lsfileae folder and check the result. This should create a src folder containing generated JAXB classes and Transformers from the sample XML Schema in the schema folder.

From the samples/quickstarts/adapter_lsfileae folder, run command *ant -f build-jws2cixs.xml*. This should add a set of java files to the src folder, these implement a JAX-WS endpoint. This also creates a webapp and an ant folders.

From the samples/quickstarts/adapter_lsfileae/ant folder, run command *ant -f build-war.xml*. This should generate a war file in the dist folder, ready for deployment.

Deploy the generated war file from the war folder into your J2EE container deployment folder (ex $CATALINA_BASE/webapps).

See the FAQ [http://www.legsem.com/legstar/legstar-cixsgen/legstar-jaxws-generator/faq.html] for common deployment issues.

You will notice that a jar file is also generated in the dist folder. As an alternative to JAX-WS RI (Metro) you can deploy this jar to Apache AXIS2, servicejars folder.

Optionally, you can use build-jws-client.xml to generate a Web service client using JAX-WS.

In order to actually run the deployed Web Service, you need to select a transport and put a customized version of the legstar-invoker-config.xml configuration file in a location such as $CATALINA_HOME/shared/classes.

There are sample invoker configuration files for each of the transports supported in:

- samples/quickstarts/chttprt for HTTP transport
- samples/quickstarts/cmqrt for WebSphere MQ transport

- samples/quickstarts/csokrt for Socket transport

Service Proxy

There are 2 proxy samples, one that consumes a POJO and one that consumes a Web Service.

Service Proxy to a POJO

Change directory to samples/quickstarts/proxy_pojo_jvmquery/.

The distribution contains a simple POJO called jvmquery. It is provided as a jar file in the samples/quickstarts/proxy_pojo_jvmquery/java folder and as source are in samples/quickstarts/proxy_pojo_jvmquery/src.

Run command *ant -f build-java2cob.xml* which generates a COBOL-annotated XML schema named jvmquery.xsd in the schema folder.

Each java type, from the jvmquery source, maps to an XML schema complex type. Each XML schema element has special Cobol annotations with default attributes, such as maximum character string sizes.

Run command *ant -f build-coxb.xml* from the samples/quickstarts/proxy_pojo_jvmquery folder and check the result. This should add to the src folder generated JAXB classes and Transformers from the sample XML Schema in the schema folder.

From the samples/quickstarts/proxy_pojo_jvmquery folder, run command *ant -f build-cixs2jws.xml*. This should generate a web.xml file in the webapp/cixs-jvmquery/WEB-INF folder, an ant script in the ant folder and a sample COBOL program in the cobol folder.

From the samples/quickstarts/proxy_pojo_jvmquery/ant folder, run command *ant -f build-war.xml*. This should generate a war file in the dist folder, ready for deployment.

Deploy the generated war file from the dist folder into your J2EE container deployment folder (ex $CATALINA_BASE/webapps).

The sample COBOL program is an almost complete working sample of a proxy client. You can follow instructions in the code to add the missing instructions or just run *ant -f build-jvmquery-cobol-src.xml* that automatically does that.

Edit the JVMQUERY.cbl source and check the W00-SERVICE-URI, make sure it points to the machine where you deployed the proxy. You can now upload that program to z/OS, compile and run it. A sample compilation jcl is in the jcl folder.

Service Proxy to a Web Service

Change directory to samples/quickstarts/proxy_ws_cultureinfo/.

The distribution contains a simple Web Service called cultureinfo. It is provided as a war file in the samples/quickstarts/proxy_ws_cultureinfo/war folder.

The war file contains a simple JAX-WS endpoint. If the JAX-WS libraries are not available in your target J2EE container, you can get them from <installDir>/lib.

Deploy this war file to your J2EE container deployment folder (ex $CATALINA_BASE/webapps).

Check the build-xsd2cob.xml inputXsdUri parameter. It assumes the J2EE container you deployed the target Web Service to, is listening on localhost, port 8080. You might need to customize this.

You can now run command *ant -f build-xsd2cob.xml* which generates a COBOL-annotated XML schema named cultureinfo.xsd in the schema folder. It does so by reading the target Web Service WSDL.

Each complex type and element from the source WSDL also exists in the generated mapping XML schema. Each element also has special Cobol annotations with default attributes, such as maximum character string sizes.

Run command *ant -f build-coxb.xml* from the samples/quickstarts/proxy_pojo_cultureinfo folder and check the result. This should create a src folder containing generated JAXB classes and Transformers from the sample XML Schema in the schema folder.

From the samples/quickstarts/proxy_pojo_cultureinfo folder, run command *ant -f build-cixs2jws.xml*. This should generate a web.xml file in the webapp/cixs-cultureinfo/WEB-INF folder, an ant script in the ant folder and a sample COBOL program in the cobol folder.

From the samples/quickstarts/proxy_pojo_cultureinfo/ant folder, run command *ant -f build-war.xml*. This should generate a war file in the dist folder, ready for deployment.

Deploy the generated war file from the dist folder into your J2EE container deployment folder (ex $CATALINA_BASE/webapps).

The sample COBOL program is an almost complete working sample of a proxy client. You can follow instructions in the code to add the missing instructions or just run *ant -f build-cultureinfo-cobol-src.xml* that automatically does that.

Edit the CULTUREI.cbl source and check the W00-SERVICE-URI, make sure it points to the machine where you deployed the proxy. You can now upload that program to z/OS, compile and run it. A sample compilation jcl is in the jcl folder.

Installing LegStar modules for z/OS

Pre-requisites

LegStar mainframe programs are mostly written in C/370, therefore Language Environment (LE) is mandatory.

Some COBOL sample programs are part of the delivery. They were compiled using Enterprise COBOL for z/OS 3.3.0.

All CICS programs are translated with CICS 6.4.0 COMMAND LANGUAGE TRANSLATOR.

Installing

Download the latest LegStar distribution for z/OS [http://www.legsem.com/legstar/legstar-transport/legstar-distribution-zos/download.html].

Unzip the binary distribution package into the directory of your choice, referred to as <installDir> in the following steps.

The directory tree should look like this:

```
<installDir>
```

```
        |---->LEGSTAR.version.C370.XMIT
        |---->LEGSTAR.version.CNTL.XMIT
        |---->LEGSTAR.version.COBOL.XMIT
        |---->LEGSTAR.version.H370.XMIT
        |---->LEGSTAR.version.LOAD.XMIT
        |---->LEGSTAR.version.OBJ.XMIT
```

Upload the XMIT files into sequential files on z/OS.

It is very important that:

- You use a *BINARY* transfer type
- Use *RECFM=FB LRECL=80 PS* files as targets

With FTP, you can use command such as:

```
   QUOTE SITE RECFM=FB LRECL=80 BLKSIZE=27920
```

to force new uploaded files to have an acceptable format for the TSO XMIT/RECEIVE protocol.

Under TSO, you can now RECEIVE the XMIT files into their target PDSs using a JCL such as:

```
//P390XMIT   JOB  (ACCT#),'USERID',
//*          RESTART=REFRESH,
//           MSGCLASS=X,NOTIFY=&SYSUID,PRTY=14,REGION=0M
//*********************************************************************
//*  RESTORE XMIT FILES                                               *
//*********************************************************************
//SUBMIT   EXEC PGM=IKJEFT1A
//SYSTSPRT DD   SYSOUT=*
//SYSTSIN  DD   *
  RECEIVE INDATASET('hlq.LEGSTAR.version.LOAD.XMIT')
       DSNAME('hlq.LEGSTAR.version.LOAD')
  RECEIVE INDATASET('hlq.LEGSTAR.version.OBJ.XMIT')
       DSNAME('hlq.LEGSTAR.version.OBJ')
  RECEIVE INDATASET('hlq.LEGSTAR.version.C370.XMIT')
       DSNAME('hlq.LEGSTAR.version.C370')
  RECEIVE INDATASET('hlq.LEGSTAR.version.H370.XMIT')
       DSNAME('hlq.LEGSTAR.version.H370')
  RECEIVE INDATASET('hlq.LEGSTAR.version.CNTL.XMIT')
       DSNAME('hlq.LEGSTAR.version.CNTL')
  RECEIVE INDATASET('hlq.LEGSTAR.version.COBOL.XMIT')
       DSNAME('hlq.LEGSTAR.version.COBOL')
/*
```

Finally in the target CICS region:

- Add hlq.LEGSTAR.version.LOAD to the RPL list
- Customize and run CSD updates from hlq.LEGSTAR.version.CNTL(CICSCSDU)

Compile and link-edit LegStar programs

If, for some reason, le load modules shipped with the LegStar distribution are not suitable, you can chose to either re-link edit or recompile all the modules.

You might have noticed that the LegStar delivery contains object modules in the hlq.LEGSTAR.version.OBJ library. So you can re-link without recompiling. If you elect to recompile, these object modules will be overridden with yours.

if you decide to compile, make sure you have a C/370 compiler.

The hlq.LEGSTAR.version.CNTL library contains a set of cataloged procedures that you need to customize.

If you elect to recompile, you need to customize:

- CTRNC370: Translates and compiles a CICS C/370 program and pre-links into an object module.
- BCMPC370: Compiles a C/370 program and pre-links into an object module.

for link-editing, you need to customize:

- CLNKC370: Link-edits a CICS object module.
- BLNKC370: Link-edits an object module.

Once the cataloged procedures are ready, you can use the following JCL's to process all LegStar modules at once:

- BUILDOBJ: Compiles and pre-links all programs. After you run this job, you need to run BUILDLOA.
- BUILDLOA: Link-edits all programs. You can use this job on the object modules shipped with LegStar or after you ran BUILDOBJ.

Installing LegStar plugins for Eclipse

Pre-requisites

LegStar Eclipse plugins require Eclipse version 3.2.1 and above.

Any previous version of the LegStar Eclipse plugins must be uninstalled before installing this version.

At development time, the Eclipse plugins are standalone and don't necessitate that you install the core product. When you deploy your Transformers and Services though, you will need an easy access to the LegStar libraries and will find it easier to get them from the core LegStar product that you can get from this link [http://www.legsem.com/legstar/legstar-distribution/].

Installing

Within Eclipse, create a remote site using:

```
  Help --> Install New Software... --> Add
```

or

```
Help-->Software Updates-->Find and Install-->Search for new features-->New Remot
```

depending on your Eclipse version.

Name the remote site LegStar and have the URL point to:

```
http://www.legsem.com/legstar/eclipse/update
```

Select all the features.

Uncomment the "Contact all update sites during install to find required software" check box. If you keep it checked, it would still work but calculating requirements and dependencies takes a very long time.

If you get complaints about MD5 checksum being incorrect, you probably have a proxy issue. As an alternative, you can download the update site from this link [http://www.legsem.com/legstar/maven/repo/com/legsem/legstar/legstar-site/1.5.3/legstar-site-1.5.3.zip], add a local site to Eclipse and install from there.

Once installation is complete, restart Eclipse and check that the LegStar plugins show up in:

```
Help-->About Eclipse SDK-->Installation Details-->Installed Software
```

From within Eclipse, setup your preferences:

```
Window-->Preferences...-->LegStar
```

Uninstalling

To uninstall, use:

```
Help-->About Eclipse SDK-->Installation Details-->Installed Software.
```

then right-click on the feature you wish to uninstall.

Chapter 4. Using LegStar

We show 2 use cases in the following sections.

The Adapter use case is one where you start from a commarea-driven COBOL CICS program (LSFILEAE) and turn it a Web Service.

The Proxy use case is one where you started from a WSDL (Microsoft LIVE search) and access the LIVE Web Service from a COBOL CICS program.

You have the choice between using ant scripts and Eclipse plugins.

Using ant scripts

Expose a COBOL program as a Web Service

COBOL structures to XML Schema translation

The process starts by translating COBOL Structures to XML Schema.

You can use the *build-cob2xsd.xml* ant script from the samples/quickstarts/adapter_lsfileae folder as your starting point.

The sample script translates the lsfileae COBOL program from the cobol folder to an XML schema called lsfileae.xsd in the schema folder.

The script executes a single ant task: cob2xsd.

Generally, there are 4 things you will need to check in *build-cob2xsd.xml* in order to adapt it to your needs:

- Make sure the classpath setting for the cob2xsd task is correct. The fileset should point to the location where you installed LegStar.
- The target parameter should point to the location where you want the translated XML Schemas to go.
- The targetNamespace parameter will become the translated XML Schema target namespace. It must be a valid URI.
- The fileset parameter of the cob2xsd ant task must designate one, or more, COBOL source fragments that you want to translate to XML Schema.

For a complete list of options for the cob2xsd task, you can refer to CobolStructureToXsdTask [http://www.legsem.com/legstar/legstar-cob2xsd/apidocs/com/legstar/cob2xsd/task/CobolStructureToXsdTask.html].

After you run the script, you should get XML Schema files such as lsfileae.xsd which are named after the COBOL fragments translated.

Open one of these files and notice that each COBOL data structure in the LSFILEAE COBOL source has been mapped to an XML Schema type. Each Schema type has COBOL annotations that serves as mapping meta-data for other tools.

COBOL Transformers Generation for an Adapter

You can use the *build-coxb.xml* ant script from the samples/quickstarts/adapter_lsfileae folder as your starting point. It generates Transformers for the LSFILEAE COBOL program which takes the same input and output structure called Dfhcommarea.

Generally, there are 4 things you will need to change in *build-coxb.xml* in order to adapt it to your needs:

- Make sure the classpath setting is correct. The fileset should point to the location where you installed LegStar.
- The xsdFile parameter of the jaxbgen ant task must point to the location of you XML Schema with COBOL annotations.
- The jaxbPackageName parameter, for both jaxbgen and coxbgen ant tasks, must be set to the same value conforming to your naming standards.
- The jaxbRootClass parameter of the coxbgen ant task must designate one, or more, JAXB classes that you want to turn into Transformers.

build-coxb.xml executes 3 steps (or targets in ant parlance):

- The first target, generateJAXB, runs the jaxbgen ant task and turns an XML Schema with COBOL annotations into JAXB classes with COBOL annotations.

  For a complete list of options for the jaxbgen task, you can refer to CobolJAXBGenerator [http://www.legsem.com/legstar/legstar-core/legstar-jaxbgen/apidocs/com/legstar/jaxb/gen/CobolJAXBGenerator.html].

- The second target, compileJAXB, is a regular java compilation step for the JAXB classes previously generated.
- The third target, generateCOXB, runs the coxbgen ant task and produces the actual Transformers. The generation process reflects on the JAXB classes compiled at the previous step.

  You will notice that the coxbgen ant task takes one or more jaxbRootClass elements. These are needed to designate which JAXB class (or classes) should become a Transformer. You would generally pickup the higher classes in the hierarchy but you don't have to.

  By default you get Java to Host Transformers only. In addition, you can get Host to XML and Host to JSON Transformers (use the xmlTransformers and jsonTransformers options).

  For a complete list of options for the jaxbgen task, you can refer to CoxbBindingGenerator [http://www.legsem.com/legstar/legstar-core/legstar-coxbgen/apidocs/com/legstar/coxb/gen/CoxbBindingGenerator.html].

The Using generated tranformers section gives examples of code you could write to run Transformers.

Mainframe Adapter generation

The final step in the process is to generate a Mainframe Adapter, which is a JAX-WS Endpoint.

You can use the *build-jws2cixs.xml* ant script from the samples/quickstarts/adapter_lsfileae folder as your starting point.

The sample script produces several artifacts and is primarily a single step script running the jaxws2cixsgen ant task.

Let us review the important parameters that you will need to change in *build-jws2cixs.xml* in order to adapt it to your needs:

- Make sure the classpath setting for the jaxws2cixsgen task is correct. The fileset should point to the location where you installed LegStar.

  It is also important that the JAXB and Transformers previously generated be on the classpath.

- jaxws2cixsgen takes a series of file system locations as parameters. Follow the comments in the ant script to select the correct locations.
- The cixsJaxwsService element is where the actual Adapter is described. The name and packageName parameters are up to you but must be valid java identifier and package name respectively.
- The cixsOperation element is there because you can implement more than one operation for a Service. You just need a different name (a valid java identifier) for each operation.
- The cicsProgramName parameter must exactly match an existing CICS program. The generated service will attempt to run that program.
- The input and output elements must refer to JAXB types and package names as generated by the COBOL Transformers generator. In the case of LSFILEAE, the JAXB type is Dfhcommarea for both input and output.
- Finally the webServiceParameters element is JAX-WS specific. It specifies the way the service will be exposed via WSDL.

For a complete list of options for the jaxws2cixsgen task, you can refer to Jaxws2CixsGenerator [http://www.legsem.com/legstar/legstar-cixsgen/legstar-jaxws-generator/apidocs/com/legstar/cixs/jaxws/gen/Jaxws2CixsGenerator.html].

After you run the script, you should get a new set of java classes which implement the JAX-WS endpoint.

Besides the JAX-WS java classes, you will also find ant scripts that were generated to help you with deploying your endpoint. build-jar.xml bundles the classes in a jar archive that you could deploy to AXIS2 for instance. build-war.xml bundles a war file suitable for Sun's JAX-WS RI (Metro).

Once deployed, you can use the Web Service with any SOAP client such as soapUI [http://www.soapui.org/].

Consume a Web Service from a COBOL program

WSDL/XSD Structures Mapping

The process starts by mapping XML schema types, from the target Web Service WSDL, to COBOL data items.

You can use the *build-xsd2cob.xml* ant script from the samples/quickstarts/proxy_ws_cultureinfo folder as your starting point.

The script executes a single ant task, xsd2cob, which can process WSDL or XML Schema as input to produce a new XML Schema with COBOL annotations.

Assuming you want to customize *build-xsd2cob.xml* for the Microsoft LIVE search SOAP service. Follow these steps:

- Make sure the classpath setting for the xsd2cob task is correct. The fileset should point to the location where you installed LegStar.
- The inputXsdUri is the location of your input WSDL or XML Schema. In the case of Microsoft LIVE Web Service you can use: http://soap.search.msn.com/webservices.asmx?wsdl
- The targetXsdFile parameter should point to the file name for the generated XML Schema with COBOL annotations (Can also be a folder name in which case the file name is derived from inputXsdUri).

  The targetCobolFile parameter should point to the file name for the generated COBOL copybook (Can also be a folder name in which case the file name is derived from inputXsdUri).

For a complete list of options for the xsd2cob task, you can refer to Xsd2CobTask [http://www.legsem.com/legstar/legstar-xsd2cob-pom/legstar-xsd2cob/apidocs/com/legstar/xsd/def/Xsd2CobTask.html].

After you run the script, you should get a new XML Schema file with the name you specified. You should also get a COBOL copybook.

Open this file and notice that each root Elements and Complex Types in the WSDL source has been mapped to a COBOL data item. Each Schema type has COBOL annotations that serves as mapping meta-data for other tools.

Since we started from a WSDL, a certain number of default COBOL data attributes were assigned. For instance, all character strings are 32 characters long. While this might be an acceptable default, it is not always the case. In our situation, the application ID (AppID) must be 40 characters long. We need to change the COBOL picture from X(32) to X(40).

COBOL Transformers Generation for a Proxy

You can use the *build-coxb.xml* ant script from the samples/quickstarts/proxy_ws_cultureinfo folder as your starting point. It generates Transformers for the cultureinfo Web Service which takes the GetInfo structure as its input and the GetInfoResponse structure as its output.

Refer to COBOL Transformers Generation for an Adapter for a description of what you need to customize in *build-coxb.xml* and how it works.

Assuming you want to customize *build-coxb.xml* for the Microsoft LIVE search SOAP service. Follow these steps:

- Change xsdFile parameter of the jaxbgen ant task to point to the location the XML Schema that was produced with the WSDL/XSD Structures Mapping tool.
- Change the jaxbPackageName parameter, for both jaxbgen and coxbgen, to something like *com.microsoft.schemas.msnsearch*.
- Change the jaxbRootClass parameters to Search and SearchResponse, which are the wrapper elements expected and produced by the target LIVE Web Service.

if you execute the ant script as is, you will get a JAXB error:

```
  [ERROR] A class/interface with the same name "com.microsoft.schemas.msnsearch.Se
```

This is because Microsoft uses the same names for Complex Types and Elements in their XML Schemas which confuses JAXB. To solve this, you can use the typeNameSuffix parameter on the jaxbgen task. The Task should now look like this:

```
<jaxbgen
    xsdFile="LIVESearch.xsd"
    targetDir="src"
    jaxbPackageName="com.microsoft.schemas.msnsearch"
    generateIsSetMethod="true"
    serializableUid="1"
    typeNameSuffix="Type"
/>
```

You can now execute the ant script.

The Using generated tranformers section gives examples of code you could write to run Transformers.

Mainframe Proxy generation

The final step in the process is to generate a Mainframe Proxy, which is a JAX-WS Client that responds to Mainframe requests and mediates the call to the target Web Service, for instance the Microsot LIVE Search Web Service.

You can use the *build-cixs2jws.xml* ant script from the samples/quickstarts/proxy_ws_cultureinfo folder as your starting point.

The sample script produces several artifacts and is primarily a single step script running the cixs2jaxwsgen ant task.

Let us review the important parameters that you will need to change in *build-cixs2jws.xml* in order to adapt it to your needs:

- Make sure the classpath setting for the cixs2jaxwsgen task is correct. The fileset should point to the location where you installed LegStar.

  It is also important that the JAXB and Transformers previously generated be on the classpath.

- cixs2jaxwsgen takes a series of file system locations as parameters. Follow the comments in the ant script to select the correct locations.
- The hostCharset parameter is your mainframe character set (for instance IBM01140). It must be one of the available java character sets.
- The proxyTargetType parameter is either POJO or WEBSERVICE. In the case of LIVE Search it must be WEBSERVICE.
- The sampleCobolHttpClientType parameter is the type of COBOL sample that you would like to generate. This COBOL/CICS code uses HTTP to access the Proxy and therefore your choice depends on the APIs available to your CICS region. There are 3 possibilities: DFHWBCLI, WEBAPI and LSHTTAPI.
- The cixsJaxwsService element is where the actual Proxy is described. The name parameter is up to you but must be valid java identifier.
- The cixsOperation element is there because you can implement more than one operation for a Service. You just need a different name (a valid java identifier) for each operation.
- The cicsProgramName parameter designates the sample COBOL client that will be generated.

- The input and output elements must refer to JAXB types and package names as generated by the COBOL Transformers generator. In the case of LIVE Search, the JAXB types are Search and SearchResponse on input and output respectively.
- Finally the webServiceTargetParameters element gives the datail on how to access the target Web Service. In the case of LIVE search, it should look like this:

```
<webServiceTargetParameters
        wsdlUrl="http://soap.search.msn.com/webservices.asmx?wsdl"
        wsdlTargetNamespace="http://schemas.microsoft.com/MSNSearch/2
        wsdlServiceName="MSNSearchService"
        wsdlPortName="MSNSearchPort"
    />
```

For a complete list of options for the cixs2jaxwsgen task, you can refer to Cixs2JaxwsGenerator [http://www.legsem.com/legstar/legstar-cixsgen/legstar-jaxws-generator/apidocs/com/legstar/cixs/jaxws/gen/Cixs2JaxwsGenerator.html].

After you run the script, you should get a new set of artifacts which implement a JAX-WS client.

The Mainframe proxy is a Servlet to be deployed in a J2EE container. The implementation uses Sun's JAX-WS RI (Metro) as a Web Service client. The build-war.xml ant file that is generated allows you to bundle the servlet ready for deployment.

The generator also creates a sample COBOL program that behaves as an HTTP client.

The sample COBOL source contains TODO comments to help you locate where you should set values for the search request and where you can display the results. The LIVE search service requires a developer ID that you can get for free and enter in the AppID field.

This is an example of code to set the search structures properly:

```
           MOVE ZERO TO Flags--C OF COM-REQUEST.
           MOVE ZERO TO SortBy--C OF COM-REQUEST.
           MOVE ZERO TO ResultFields--C OF COM-REQUEST.
           MOVE ZERO TO R-string--C OF COM-REQUEST.
           MOVE 1 TO SourceRequest--C OF COM-REQUEST.
      *  You should specify your own Microsoft LIVE application ID
           MOVE '5588C3ACE949317B3ECAADDQ908611BDF5D8D5ZA'
             TO AppID OF COM-REQUEST.
           MOVE 'Mainframe' TO Query OF COM-REQUEST.
           MOVE 'en-US' TO CultureInfo OF COM-REQUEST.
           MOVE 'Moderate' to SafeSearch OF COM-REQUEST.
           MOVE ZERO TO Latitude OF COM-REQUEST.
           MOVE ZERO TO Longitude OF COM-REQUEST.
           MOVE ZERO TO Radius OF COM-REQUEST.
           MOVE 'Web' TO R-Source OF COM-REQUEST(1).
           MOVE ZERO TO Offset OF COM-REQUEST(1).
           MOVE 1 TO R-Count OF COM-REQUEST(1).
           MOVE SPACES TO FileType OF COM-REQUEST(1).
```

And these are lines to display the result:

```
STRING 'INVOKE-SERVICE success. First hit is '
       DELIMITED BY SIZE
       Description OF COM-REPLY(1, 1)
       DELIMITED BY SIZE
       INTO ERROR-MESSAGE.
EXEC CICS SEND TEXT FROM(ERROR-MESSAGE) FREEKB END-EXEC.
DISPLAY 'Response data length=' WBCLI-RESPONSE-BODY-LEN.

DISPLAY 'SourceResponse--C ='
         SourceResponse--C OF COM-REPLY.
DISPLAY 'R-Source(1)=' R-Source OF COM-REPLY(1).
DISPLAY 'Total(1)=' Total OF COM-REPLY(1).
DISPLAY 'R-Title(1, 1)=' R-Title OF COM-REPLY(1, 1).
DISPLAY 'Description(1, 1)='
         Description OF COM-REPLY(1, 1).
```

After you add these lines of code, you should be able to upload the source onto your mainframe and get it compiled and defined to your CICS region. Please note that this program calls the CICS DFHWBCLI program defined in the CICS standard DFHWEB group. Alternatively, LegStar supports the new EXEC CICS WEB API or even supports older version of CICS with its own HTTP library.

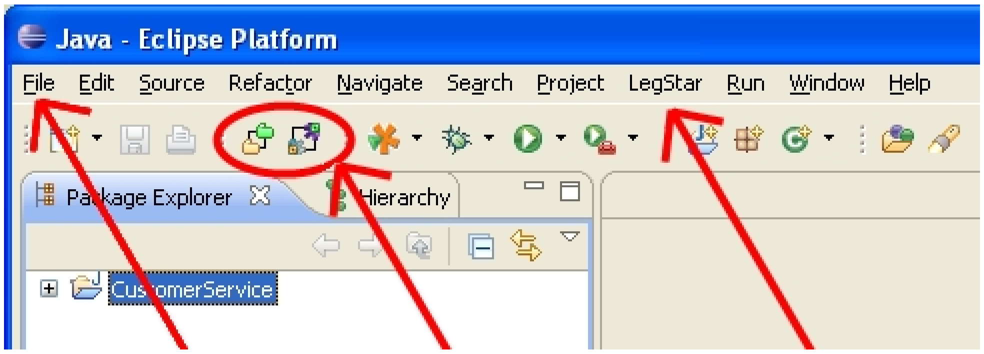
Using Eclipse plugins

Expose a COBOL program as a Web Service

Start by creating a new standard Java Eclipse Project named *CustomerService*. It is important that the project be of a Java nature.

The LegStar options are available from the File->New->Other.. ->LegStar dialog or directly from the LegStar menu or toolbar buttons:

Figure 4.1. LegStar menu and toolbar options

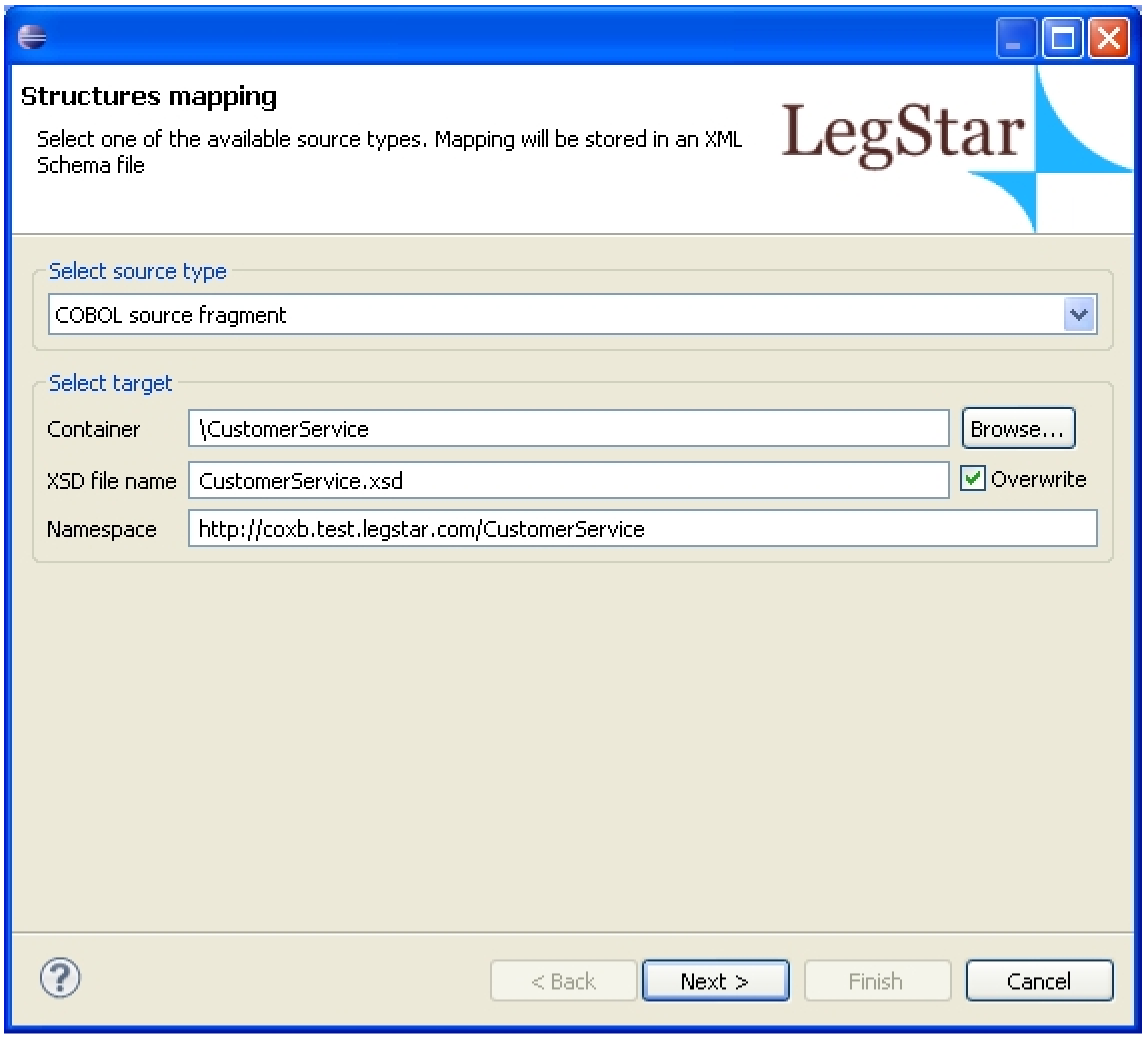

Java - Eclipse Platform
File Edit Source Refactor Navigate Search Project LegStar Run Window Help
Package Explorer
Hierarchy
CustomerService

COBOL structures to XML Schema translation

The process starts by translating COBOL Structures to XML Schema. This is option LegStar->New structures mapping…

On the *Structures Mapping* plug-in first page, type an XML Schema file name making sure the extension is xsd. The source type will be COBOL for this use case:

Figure 4.2. Adapter structures mapping screen select source


Structures mapping

Select one of the available source types. Mapping will be stored in an XML Schema file

Select source type: COBOL source fragment

Select target

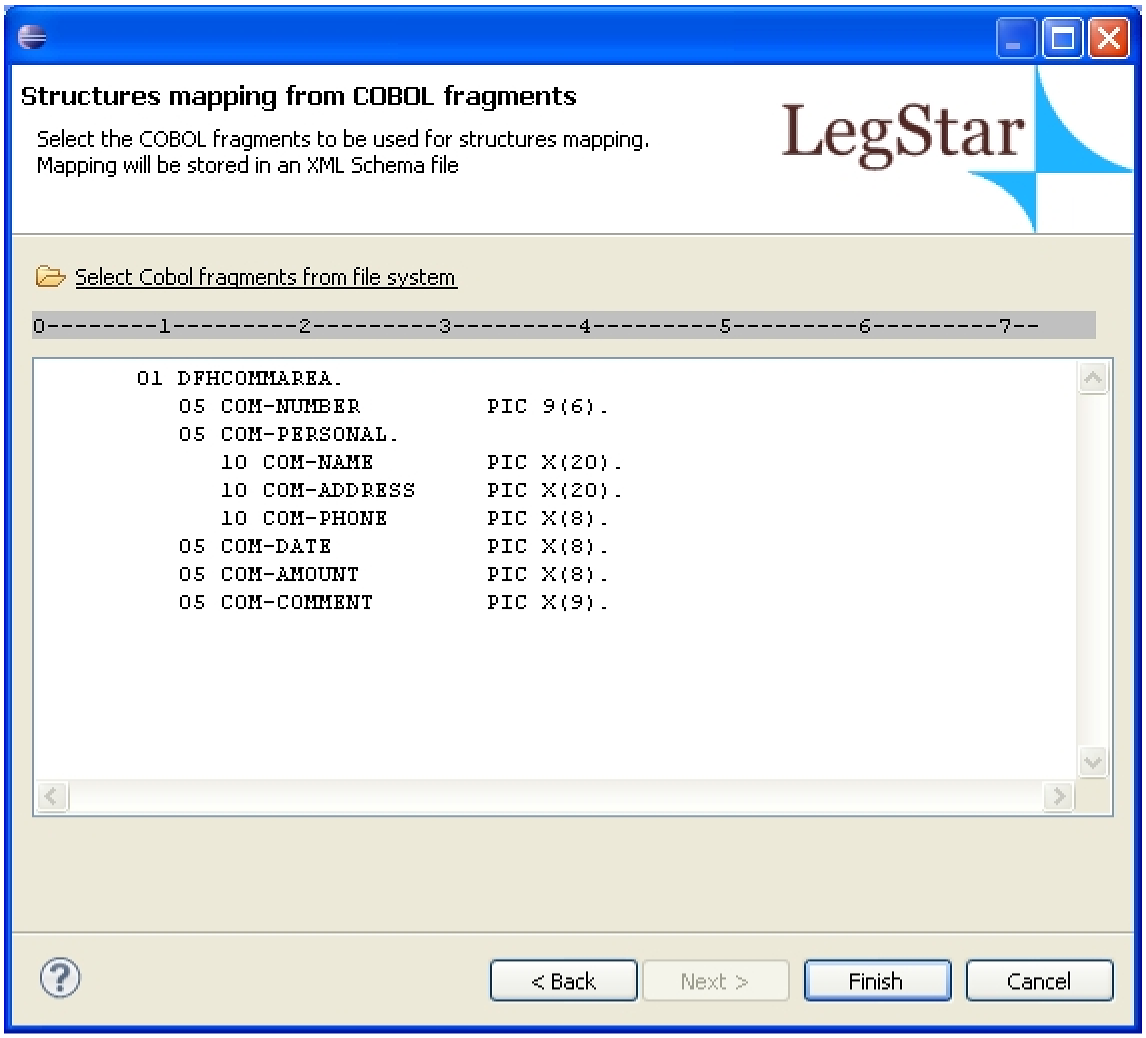
Container: \CustomerService

XSD file name: CustomerService.xsd [✓ Overwrite]

Namespace: http://coxb.test.legstar.com/CustomerService

On the next page, you can either paste COBOL code copied from somewhere else or select a file containing COBOL source code from the file system. Make sure this is valid COBOL as the mapping generator is not a full featured COBOL syntax checker:

Figure 4.3. Adapter structures mapping screen editor

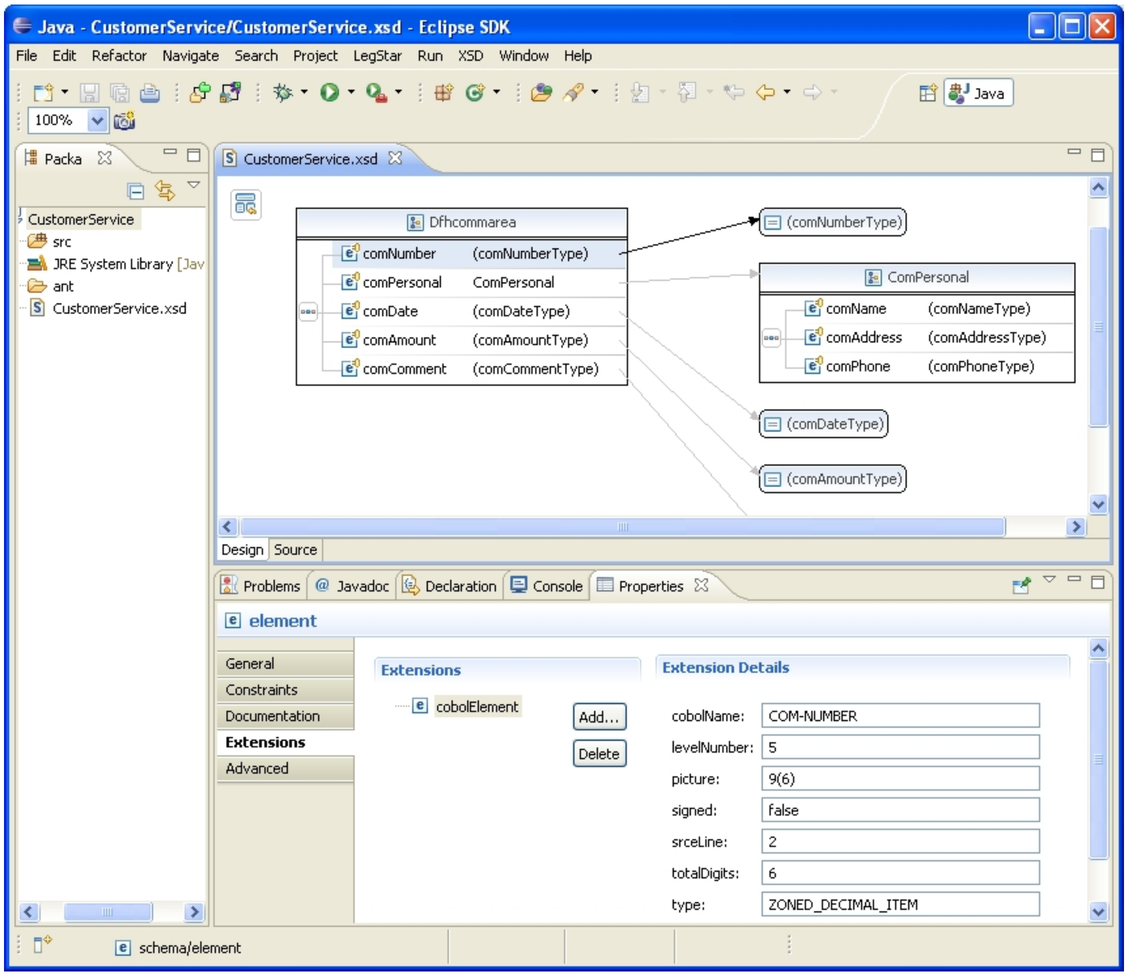

Structures mapping from COBOL fragments
Select the COBOL fragments to be used for structures mapping.
Mapping will be stored in an XML Schema file
LegStar
Select Cobol fragments from file system
0--------1---------2---------3---------4---------5---------6---------7--
< Back
Next >
Finish
Cancel

```
       01 DFHCOMMAREA.
          05 COM-NUMBER         PIC 9(6).
          05 COM-PERSONAL.
             10 COM-NAME        PIC X(20).
             10 COM-ADDRESS     PIC X(20).
             10 COM-PHONE       PIC X(8).
          05 COM-DATE           PIC X(8).
          05 COM-AMOUNT         PIC X(8).
          05 COM-COMMENT        PIC X(9).
```

By default, your COBOL code needs to be of a fixed format, between columns 8 and 72. You can change to free format in the *Window->Preferences…->LegStar->Structures mapping* dialog.

After you click finish, an ant script with a name similar to build-schemagen-CustomerService.xsd.xml is generated and launched. This script generates a new XML Schema and then the Eclipse standard XML Schema editor is opened and you can check the mapping (COBOL annotations) that was automatically generated:

Figure 4.4. Adapter structures mapping screen results



Dfhcommarea

comNumber	(comNumberType)
comPersonal	ComPersonal
comDate	(comDateType)
comAmount	(comAmountType)
comComment	(comCommentType)

(comNumberType)

ComPersonal

comName	(comNameType)
comAddress	(comAddressType)
comPhone	(comPhoneType)

(comDateType)

(comAmountType)

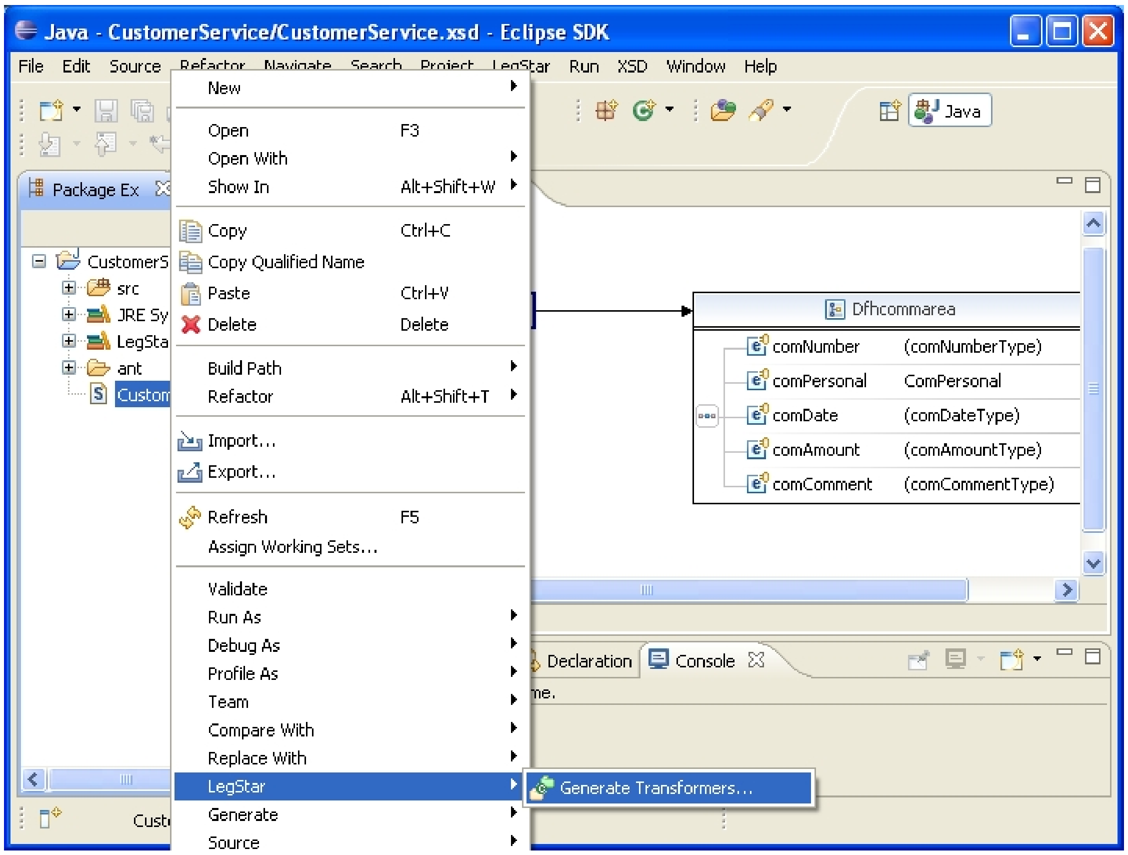
cobolName:	COM-NUMBER
levelNumber:	5
picture:	9(6)
signed:	false
srceLine:	2
totalDigits:	6
type:	ZONED_DECIMAL_ITEM

Notice the extensions used to annotate the XML Schema elements with COBOL meta-data.

COBOL Transformers Generation for an Adapter

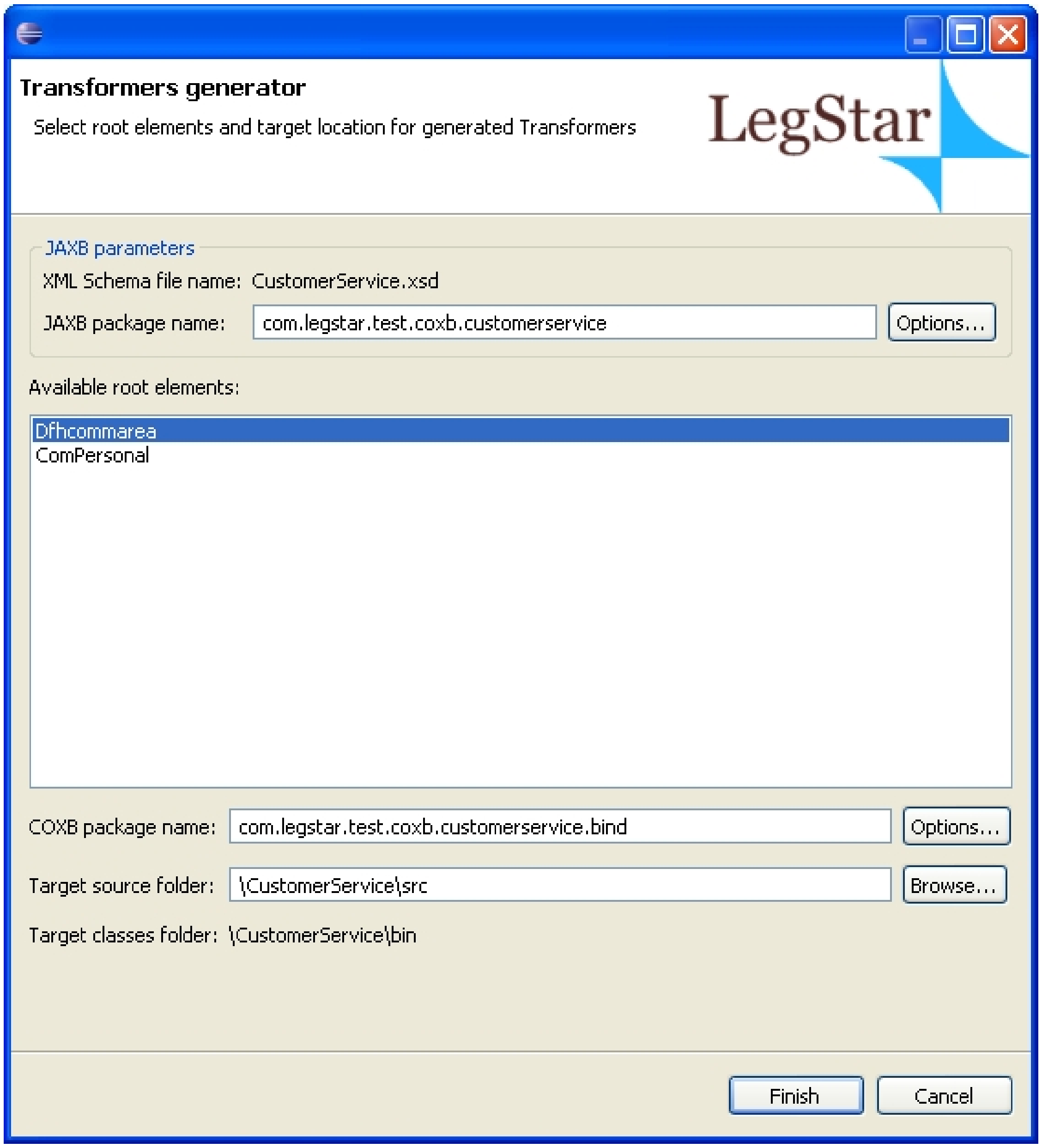
The wizard is started from the package explorer, by right clicking on the previously generated XML Schema:

Figure 4.5. Adapter COBOL Transformers generation menu



The next page allows you to specify which elements from the source XML Schema will need to be bound. All elements are displayed here but if you select a parent element, this will automatically select all children for you, so all you need to do is to select root elements:

Figure 4.6. Adapter COBOL Transformers generation parameters screen



Transformers generator

Select root elements and target location for generated Transformers

JAXB parameters

XML Schema file name: CustomerService.xsd

JAXB package name: com.legstar.test.coxb.customerservice

Available root elements:

Dfhcommarea
ComPersonal

COXB package name: com.legstar.test.coxb.customerservice.bind

Target source folder: \CustomerService\src

Target classes folder: \CustomerService\bin

In the Adapter case, the mainframe program expects a Dfhcommarea and also produces a Dfhcommarea so that's the only element we need to select.

The first options button allows you to customize the JAXB classes that will be generated.

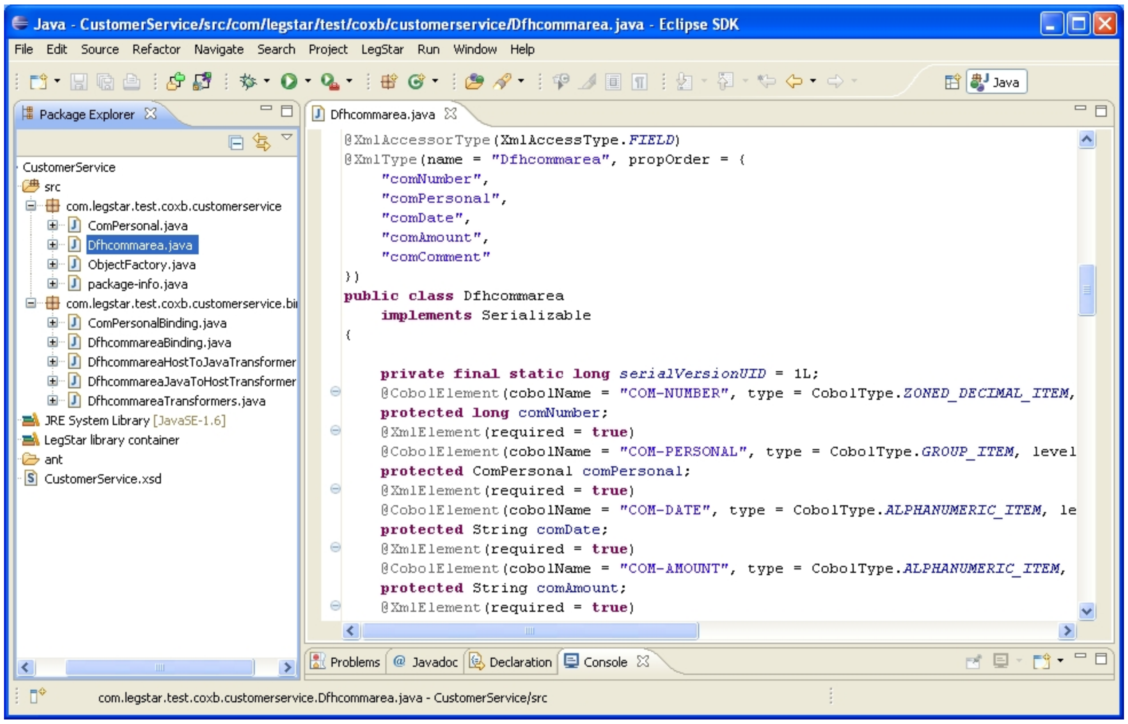
The second options button allows you to specify additional Transformers to be generated such as XML and JSON Transformers.

When you click the finish button, an ant script with a name similar to build-coxb-CustomerService.xsd.xml is generated and launched. There are two different java packages that are generated by the ant script:

- *com.legstar.test.coxb.customerservice* contains JAXB classes as generated by Sun's JAXB XJC utility but with special COBOL annotations as shown on the next screen.

- *com.legstar.test.coxb.customerservice.bind* contains the Transformers classes that can be used for fast marshaling/unmarshaling. Using these classes, there is no need for reflection on the JAXB classes to get the COBOL meta-data at runtime.

Figure 4.7. Adapter COBOL Transformers generation screen results


Java - CustomerService/src/com/legstar/test/coxb/customerservice/Dfhcommarea.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project LegStar Run Window Help
Java
Package Explorer
CustomerService
src
com.legstar.test.coxb.customerservice
ComPersonal.java
Dfhcommarea.java
ObjectFactory.java
package-info.java
com.legstar.test.coxb.customerservice.bi
ComPersonalBinding.java
DfhcommareaBinding.java
DfhcommareaHostToJavaTransformer
DfhcommareaJavaToHostTransformer
DfhcommareaTransformers.java
JRE System Library [JavaSE-1.6]
LegStar library container
ant
CustomerService.xsd
Dfhcommarea.java
Problems
Javadoc
Declaration
Console
com.legstar.test.coxb.customerservice.Dfhcommarea.java - CustomerService/src

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Dfhcommarea", propOrder = {
    "comNumber",
    "comPersonal",
    "comDate",
    "comAmount",
    "comComment"
})
public class Dfhcommarea
    implements Serializable
{

    private final static long serialVersionUID = 1L;
    @CobolElement(cobolName = "COM-NUMBER", type = CobolType.ZONED_DECIMAL_ITEM,
    protected long comNumber;
    @XmlElement(required = true)
    @CobolElement(cobolName = "COM-PERSONAL", type = CobolType.GROUP_ITEM, level
    protected ComPersonal comPersonal;
    @XmlElement(required = true)
    @CobolElement(cobolName = "COM-DATE", type = CobolType.ALPHANUMERIC_ITEM, le
    protected String comDate;
    @XmlElement(required = true)
    @CobolElement(cobolName = "COM-AMOUNT", type = CobolType.ALPHANUMERIC_ITEM,
    protected String comAmount;
    @XmlElement(required = true)
```

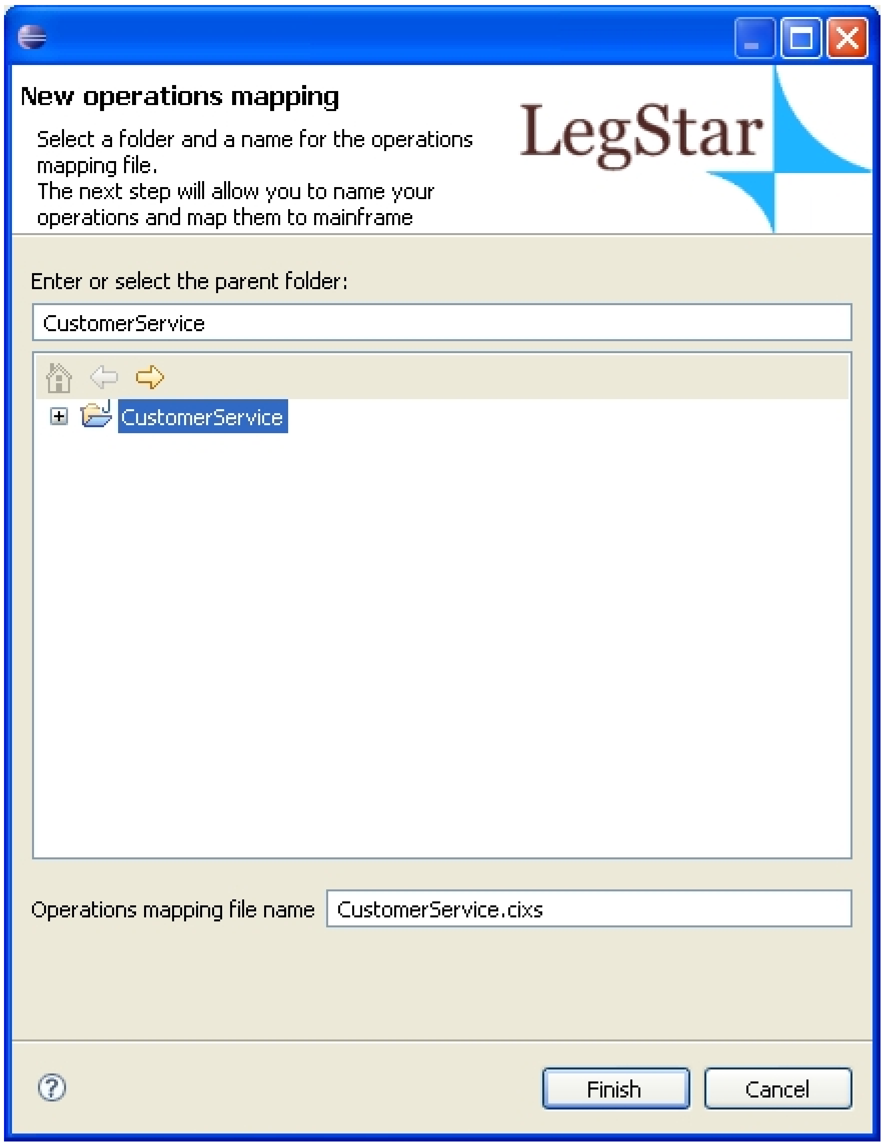
The Using generated tranformers section gives examples of code you could write to run Transformers.

Mainframe Adapter generation

The final step in the process is to generate a Mainframe Adapter, which is a JAX-WS Endpoint.

The first stage is to perform a mapping between the target mainframe program and a Java side operation. This is option *LegStar->New operations mapping…* On the first page you select a name and location for the mapping file. Operations mapping files are XML files with the *cixs* extension.

Figure 4.8. Adapter service generation screen new operation



New operations mapping

Select a folder and a name for the operations mapping file.
The next step will allow you to name your operations and map them to mainframe

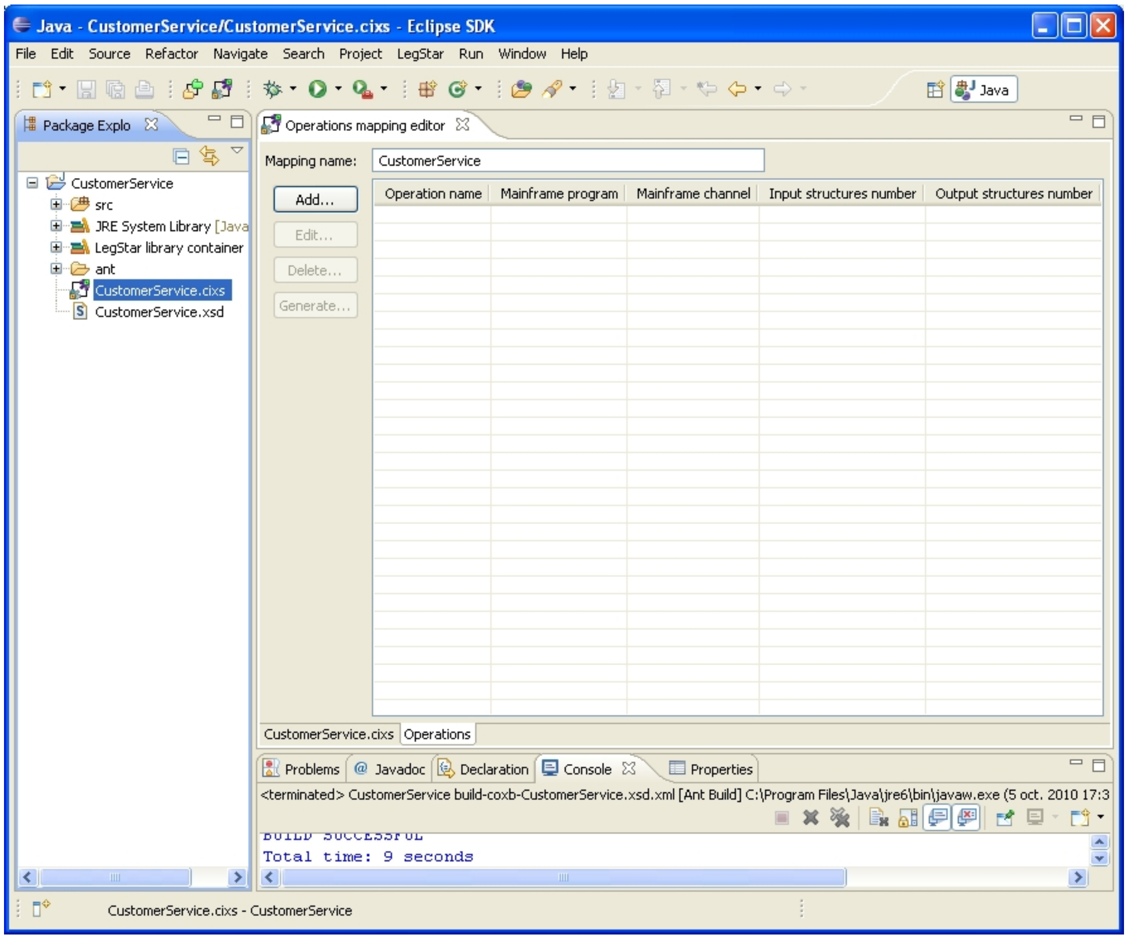
Enter or select the parent folder: CustomerService

CustomerService

Operations mapping file name: CustomerService.cixs

Clicking on Finish creates the operations mapping file and then opens up a special editor associated with files with cixs extension:

Figure 4.9. Adapter service generation screen editor


Java - CustomerService/CustomerService.cixs - Eclipse SDK
File Edit Source Refactor Navigate Search Project LegStar Run Window Help
Package Explo
CustomerService
src
JRE System Library [Java
LegStar library container
ant
CustomerService.cixs
CustomerService.xsd
Operations mapping editor
Add...
Edit...
Delete...
Generate...
CustomerService.cixs
Operations
Problems
Javadoc
Declaration
Console
Properties
CustomerService.cixs - CustomerService

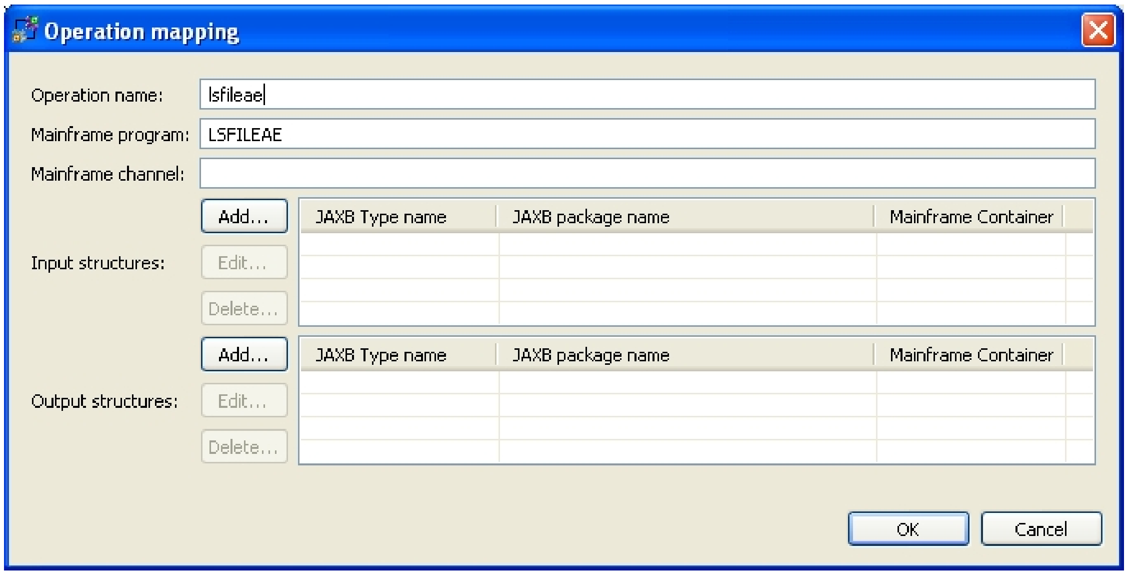
Mapping name: CustomerService

Operation name	Mainframe program	Mainframe channel	Input structures number	Output structures number

```
<terminated> CustomerService build-coxb-CustomerService.xsd.xml [Ant Build] C:\Program Files\Java\jre6\bin\javaw.exe (5 oct. 2010 17:3
BUILD SUCCESSFUL
Total time: 9 seconds
```

Click on the add button to start the operations mapping dialog:

Figure 4.10. Adapter service generation screen operation mapping



Operation name: lsfileae

Mainframe program: LSFILEAE

Mainframe channel:

Input structures:

JAXB Type name	JAXB package name	Mainframe Container

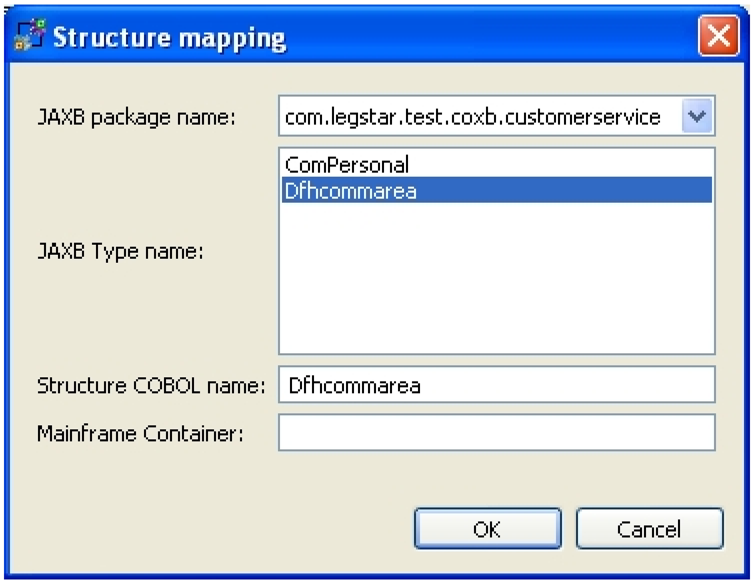
Output structures:

JAXB Type name	JAXB package name	Mainframe Container

Type in an operation name and then enter the target mainframe program name. This must correspond to an actual mainframe program.

The next step is to specify input and output structures. You will have to use the add button again. You are then presented with the JAXB classes that you generated previously:

Figure 4.11. Adapter service generation screen structures mapping



JAXB package name: com.legstar.test.coxb.customerservice

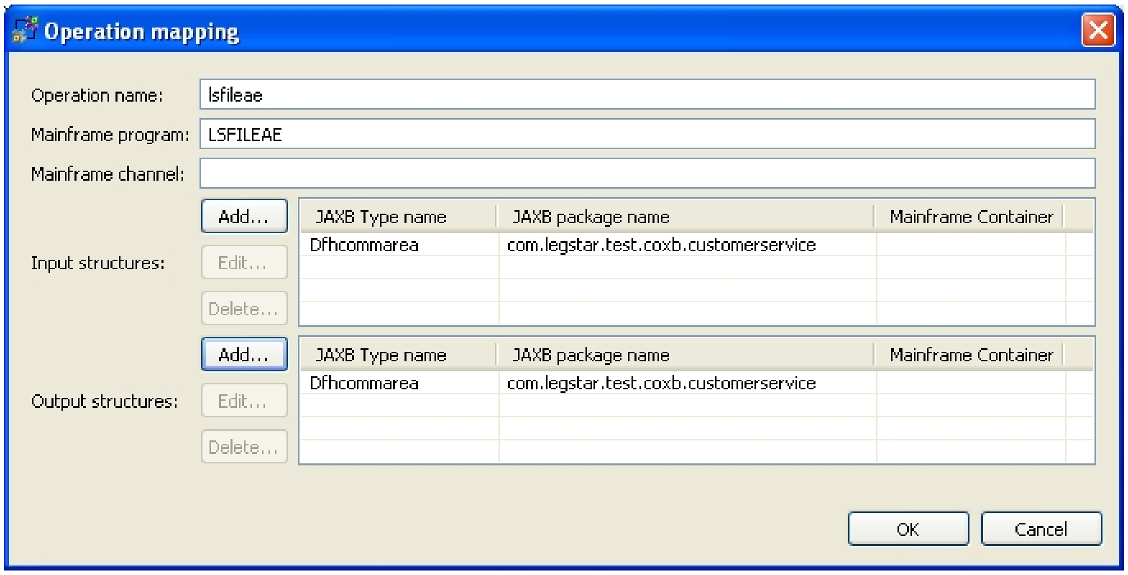
JAXB Type name: ComPersonal, Dfhcommarea (selected)

Structure COBOL name: Dfhcommarea

Mainframe Container:

The target LSFILEAE CICS program is commarea-driven, there is a single input and a single output which both happen to be described by the same COBOL structure. So all we have to do is to select Dfhcommarea both for input and output.

Figure 4.12. Adapter service generation screen operation mapping done



Operation name: lsfileae

Mainframe program: LSFILEAE

Mainframe channel:

Input structures:

JAXB Type name	JAXB package name	Mainframe Container
Dfhcommarea	com.legstar.test.coxb.customerservice	

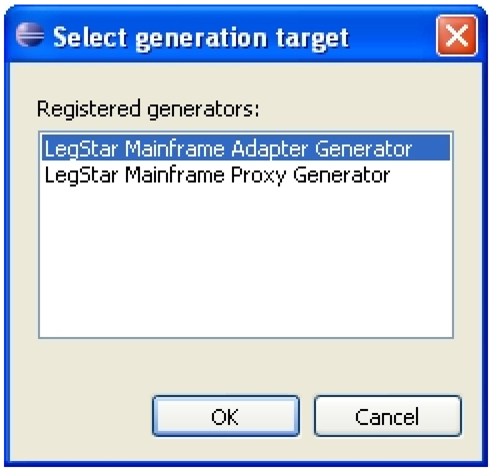
Output structures:

JAXB Type name	JAXB package name	Mainframe Container
Dfhcommarea	com.legstar.test.coxb.customerservice	

This dialog allows you to specify a different input and output structures if needed. It also allows you to specify more than one input and more than one output as it would be the case for a target container-driven program (using CICS channel/containers) for instance.

Back on the editor, you can now click on the generate button and should see the following dialog:

Figure 4.13. Adapter service generation screen select target



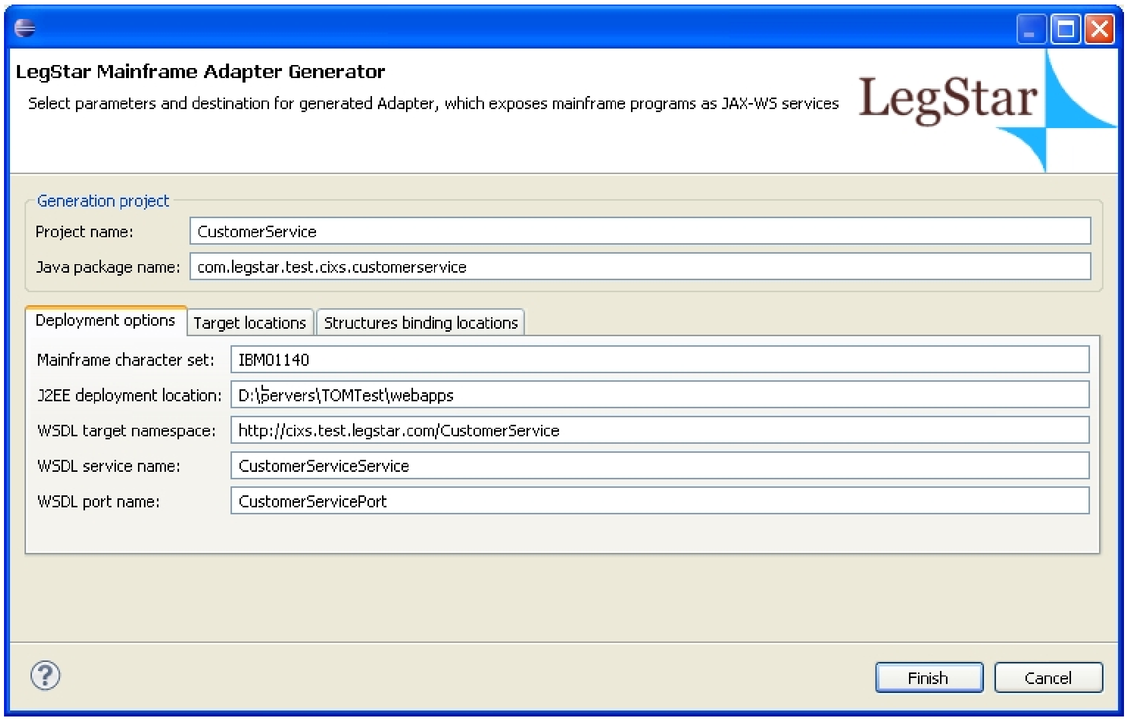
Select generation target

Registered generators:

LegStar Mainframe Adapter Generator
LegStar Mainframe Proxy Generator

The operations mapping editor can be used with different kinds of generators, which are registered dynamically on your machine. Depending on your configuration, you might have more than one possible generation target. In our case, we want to generate a *Mainframe Adapter*. When you select that target and click the OK button you get this final dialog:

Figure 4.14. Adapter service generation screen parameters



LegStar Mainframe Adapter Generator

Select parameters and destination for generated Adapter, which exposes mainframe programs as JAX-WS services

Generation project

Project name:	CustomerService
Java package name:	com.legstar.test.cixs.customerservice

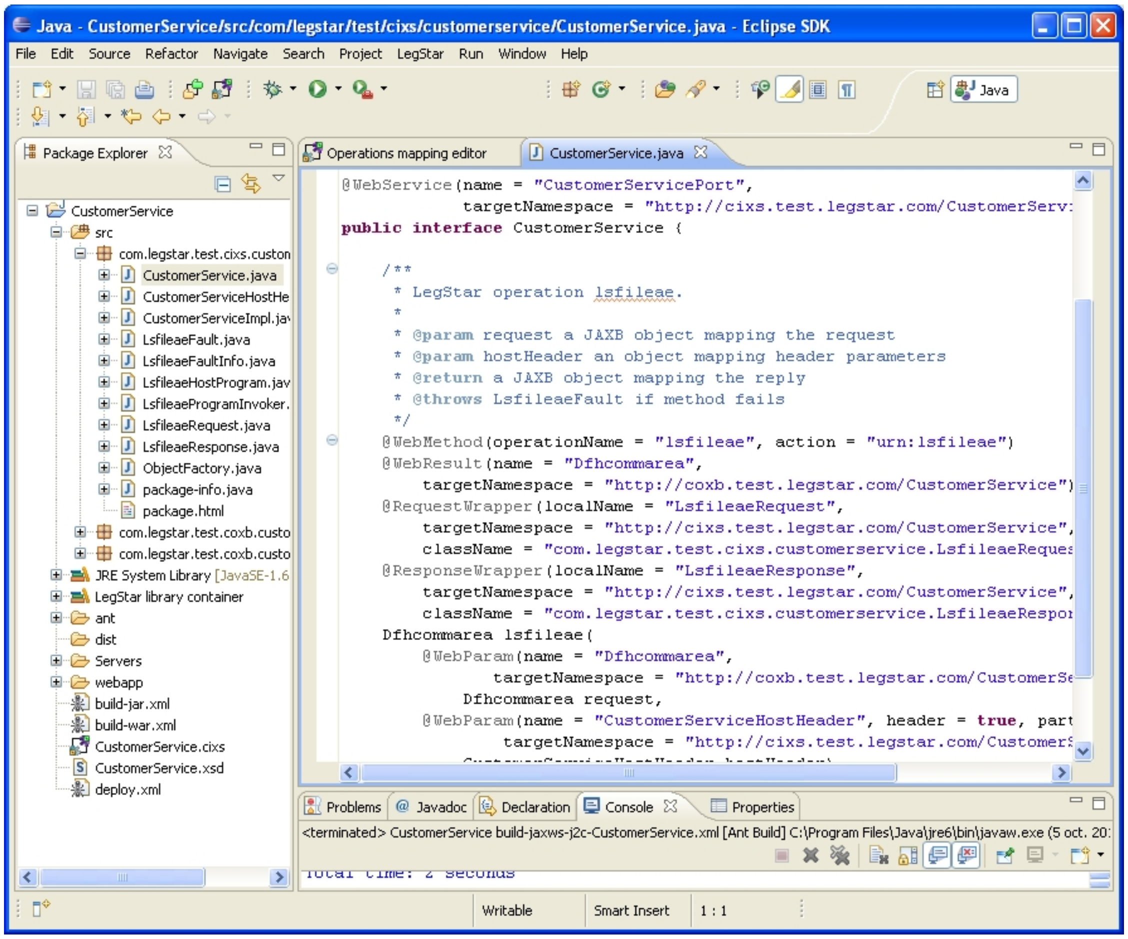
Deployment options | Target locations | Structures binding locations

Mainframe character set:	IBM01140
J2EE deployment location:	D:\Servers\TOMTest\webapps
WSDL target namespace:	http://cixs.test.legstar.com/CustomerService
WSDL service name:	CustomerServiceService
WSDL port name:	CustomerServicePort

The generation process needs to create various artifacts, including Java classes that implement a JAX-WS endpoint bundled in a J2EE war file. This dialog allows you to select the target locations and other options. Most of these options have default values derived from your preferences. You can change the proposed parameters as you see fit.

Again, the Finish button creates an ant script, which actually generates the artifacts. The ant script has a name similar to build-jaxws-j2c-CustomerService.xml.

Figure 4.15. Adapter service generation screen results



```
@WebService(name = "CustomerServicePort",
            targetNamespace = "http://cixs.test.legstar.com/CustomerServ
public interface CustomerService {

    /**
     * LegStar operation lsfileae.
     *
     * @param request a JAXB object mapping the request
     * @param hostHeader an object mapping header parameters
     * @return a JAXB object mapping the reply
     * @throws LsfileaeFault if method fails
     */
    @WebMethod(operationName = "lsfileae", action = "urn:lsfileae")
    @WebResult(name = "Dfhcommarea",
        targetNamespace = "http://coxb.test.legstar.com/CustomerService")
    @RequestWrapper(localName = "LsfileaeRequest",
        targetNamespace = "http://cixs.test.legstar.com/CustomerService",
        className = "com.legstar.test.cixs.customerservice.LsfileaeReques
    @ResponseWrapper(localName = "LsfileaeResponse",
        targetNamespace = "http://cixs.test.legstar.com/CustomerService",
        className = "com.legstar.test.cixs.customerservice.LsfileaeRespo
    Dfhcommarea lsfileae(
        @WebParam(name = "Dfhcommarea",
              targetNamespace = "http://coxb.test.legstar.com/CustomerSe
            Dfhcommarea request,
        @WebParam(name = "CustomerServiceHostHeader", header = true, part
               targetNamespace = "http://cixs.test.legstar.com/CustomerS
```

```
<terminated> CustomerService build-jaxws-j2c-CustomerService.xml [Ant Build] C:\Program Files\Java\jre6\bin\javaw.exe (5 oct. 20:
Total time: 2 seconds
```

Besides the JAX-WS java classes, you will also find ant scripts that were generated to help you with deploying your endpoint. build-jar.xml bundles the classes in a jar archive that you could deploy to AXIS2 for instance. build-war.xml bundles a war file suitable for Sun's JAX-WS RI (Metro).

Once deployed, you can use the Web Service with any SOAP client such as soapUI [http://www.soapui.org/].

Consume a Web Service from a COBOL program

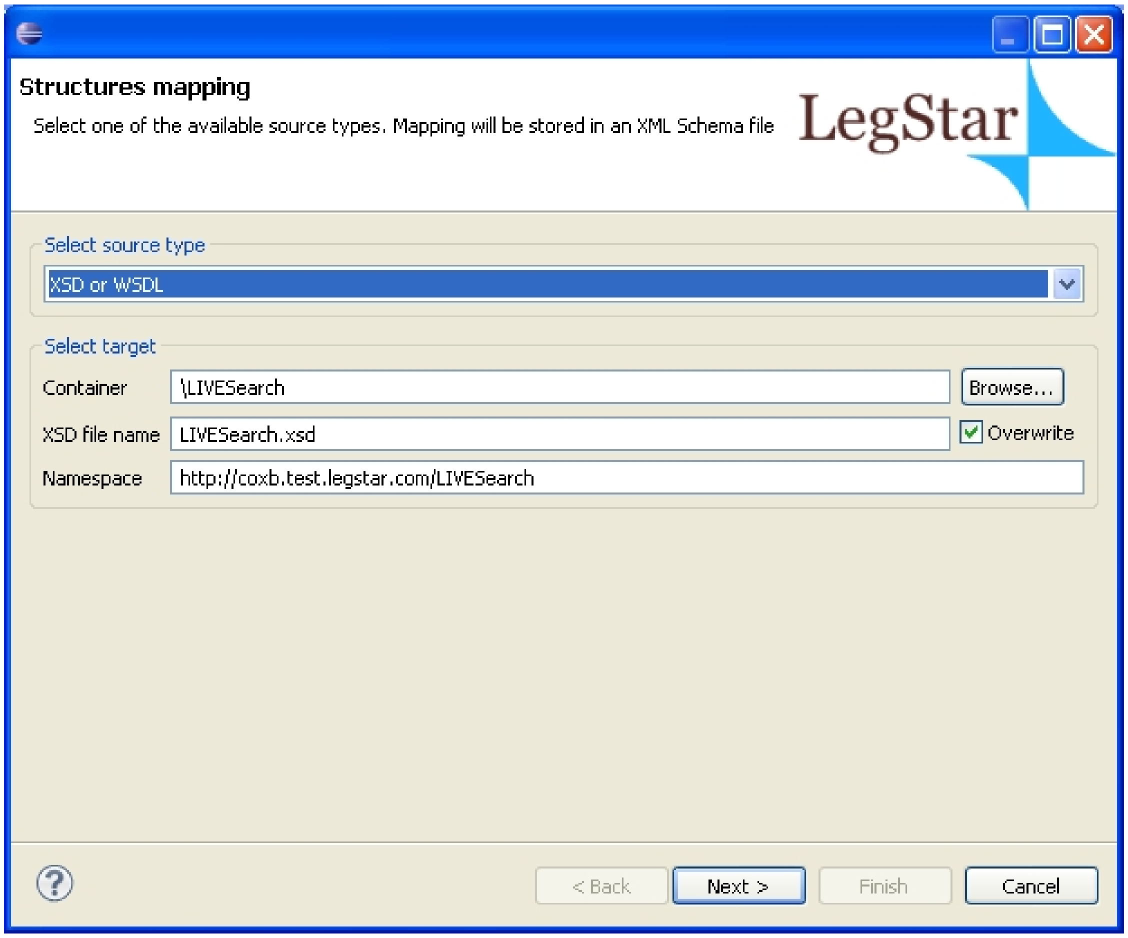
In this use case we will give a CICS program access to a Web Service (You can also access a POJO in a similar fashion). The target Web service will be the LIVE Search API [http://soap.search.msn.com/webservices.asmx?wsdl].

Start by creating a new standard Java Eclipse Project named *LIVESearch*. It is important that the project be of a Java nature.

WSDL Structures Mapping

The process starts by mapping XML schema types, from the target Web Service WSDL, to COBOL data items. This is option LegStar->New structures mapping…

Figure 4.16. Proxy WSDL structures mapping screen select source


Structures mapping

Select one of the available source types. Mapping will be stored in an XML Schema file

Select source type: XSD or WSDL

Select target

Container: \LIVESearch

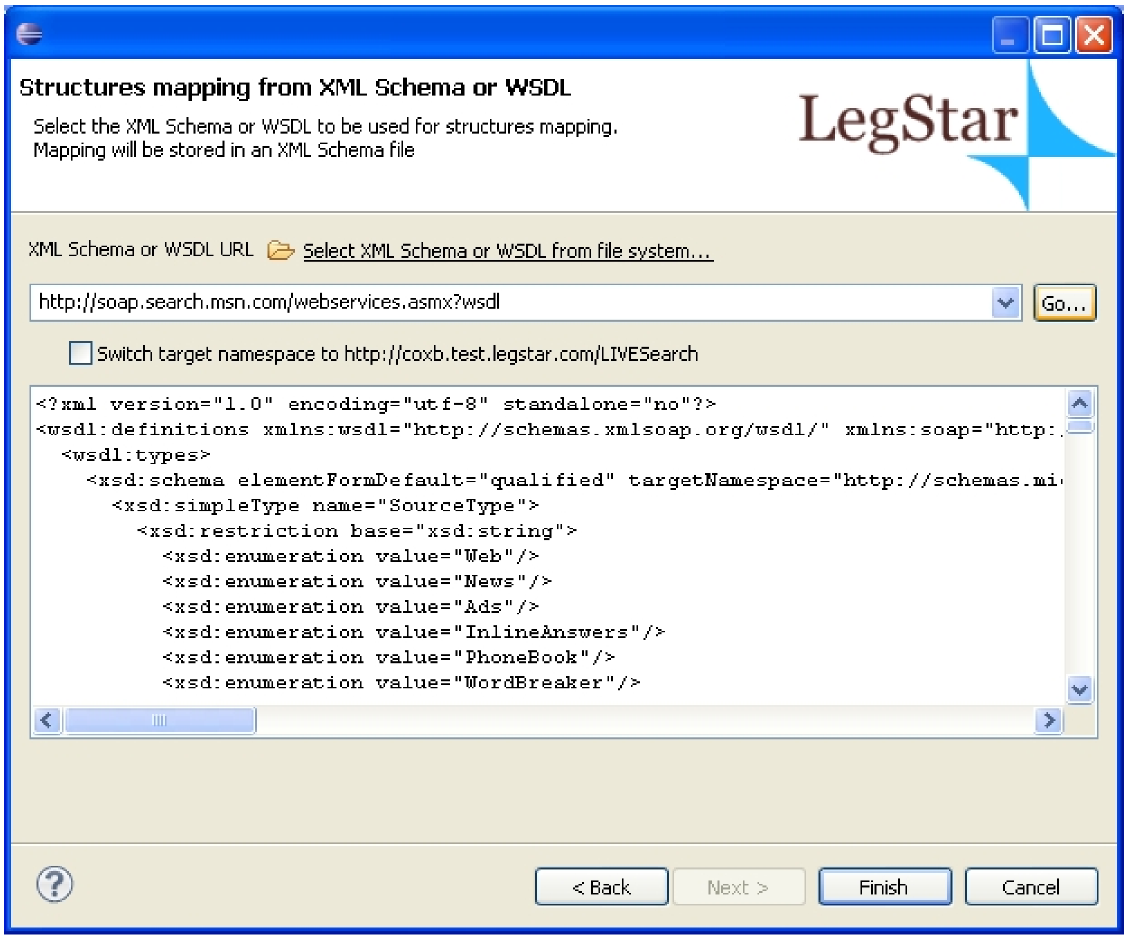
XSD file name: LIVESearch.xsd

Overwrite: ✓

Namespace: http://coxb.test.legstar.com/LIVESearch

In this use case we select the XSD or WSDL source type since our starting point is a WSDL. As a result, the next page will allow you to select a file from your file system or to fetch it directly from the internet, which we do here by typing the URL for the Microsoft LIVE Web Service: http://soap.search.msn.com/webservices.asmx?wsdl and clicking on the go button:

Figure 4.17. Proxy XSD structures mapping screen editor



Structures mapping from XML Schema or WSDL

Select the XML Schema or WSDL to be used for structures mapping.
Mapping will be stored in an XML Schema file

XML Schema or WSDL URL Select XML Schema or WSDL from file system...

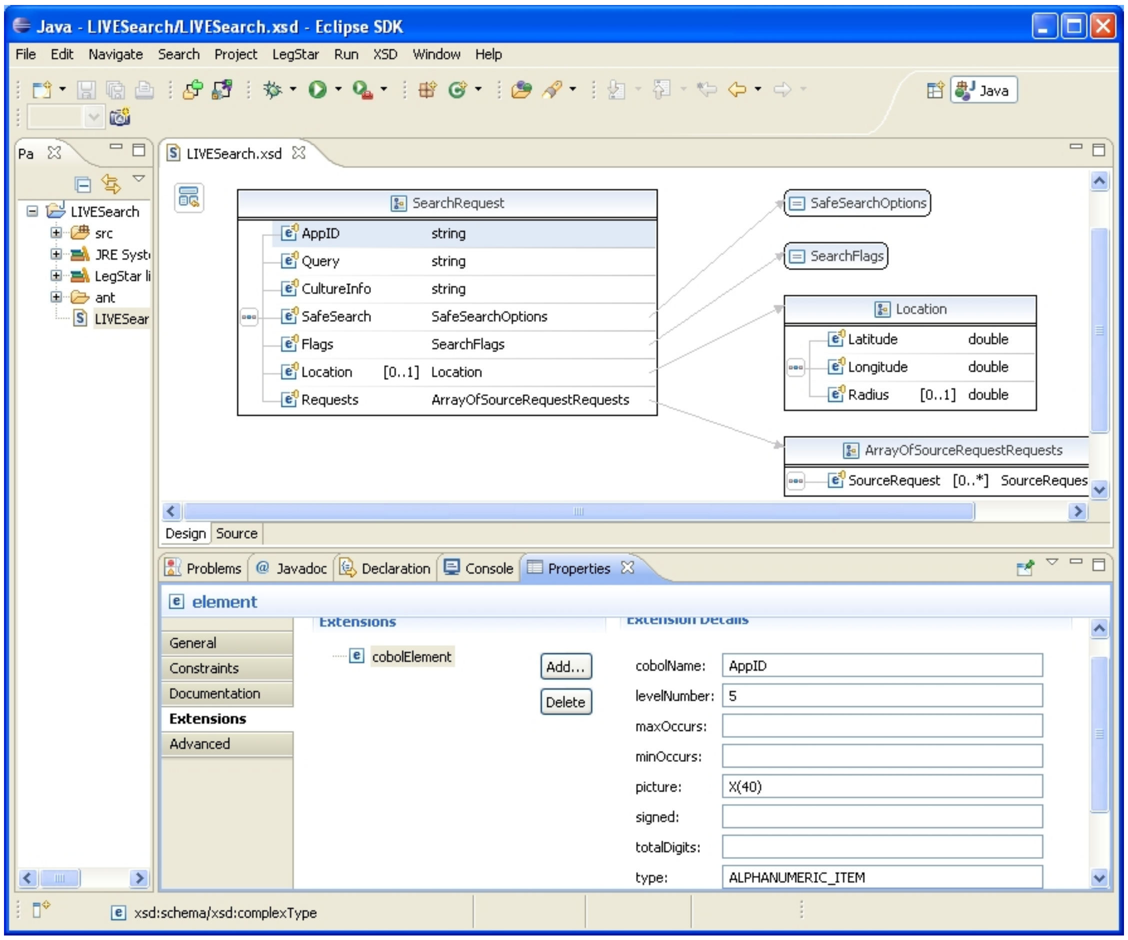
http://soap.search.msn.com/webservices.asmx?wsdl

Switch target namespace to http://coxb.test.legstar.com/LIVESearch

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:soap="http:
  <wsdl:types>
    <xsd:schema elementFormDefault="qualified" targetNamespace="http://schemas.mi
      <xsd:simpleType name="SourceType">
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="Web"/>
          <xsd:enumeration value="News"/>
          <xsd:enumeration value="Ads"/>
          <xsd:enumeration value="InlineAnswers"/>
          <xsd:enumeration value="PhoneBook"/>
          <xsd:enumeration value="WordBreaker"/>
```

At this stage, we are ready to click on the finish button and then edit the generated mapping XML Schema:

Figure 4.18. Proxy XSD structures mapping screen results


Java - LIVESearch/LIVESearch.xsd - Eclipse SDK
File Edit Navigate Search Project LegStar Run XSD Window Help
Java
LIVESearch.xsd
Design Source
Problems Javadoc Declaration Console Properties
element
General
Constraints
Documentation
Extensions
Advanced
Add...
Delete
xsd:schema/xsd:complexType

SearchRequest

AppID		string
Query		string
CultureInfo		string
SafeSearch		SafeSearchOptions
Flags		SearchFlags
Location	[0..1]	Location
Requests		ArrayOfSourceRequestRequests

SafeSearchOptions

SearchFlags

Location

Latitude		double
Longitude		double
Radius	[0..1]	double

ArrayOfSourceRequestRequests

SourceRequest	[0..*]	SourceReques

Extensions: cobolElement

Extension Details

cobolName:	AppID
levelNumber:	5
maxOccurs:	
minOccurs:	
picture:	X(40)
signed:	
totalDigits:	
type:	ALPHANUMERIC_ITEM

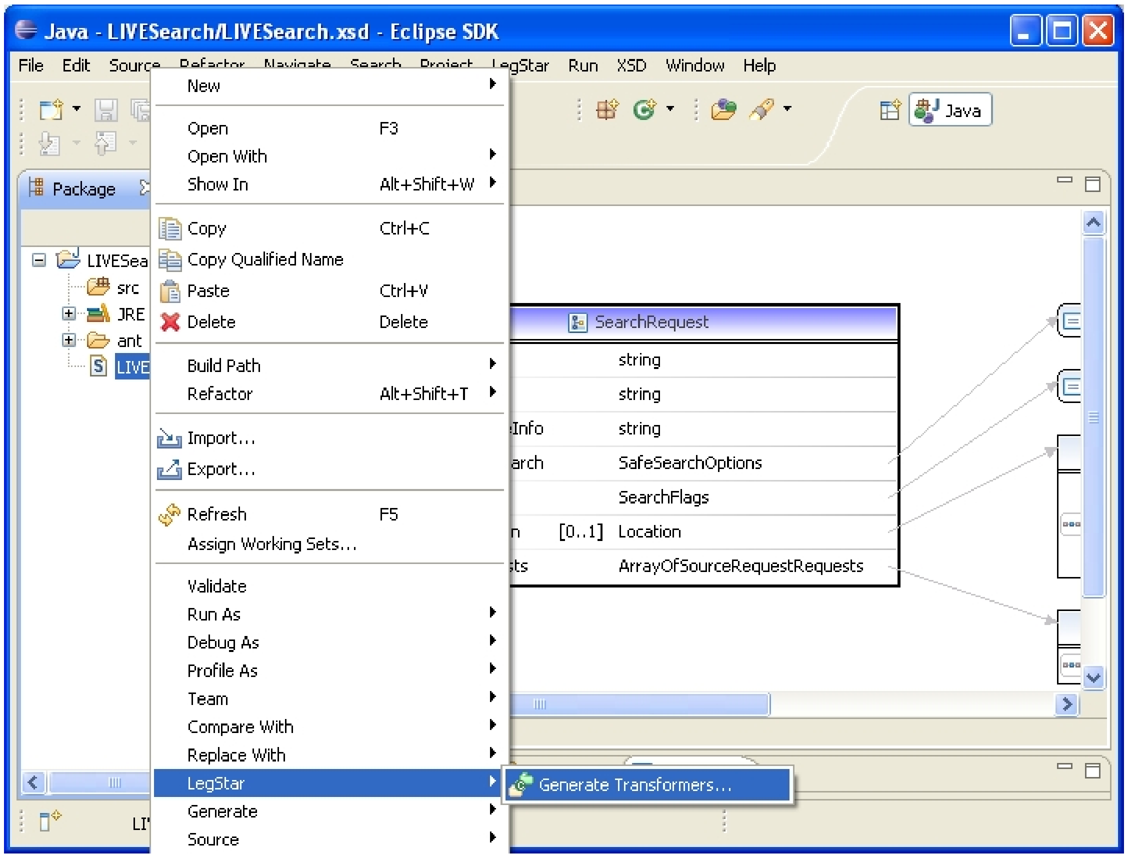
If you have the Eclipse XML Schema Editor installed, you can open the Properties view and select the extensions element to show and edit the COBOL attributes..

Since we started from a WSDL, a certain number of default COBOL data attributes were assigned. For instance, all character strings are 32 characters long. While this might be an acceptable default, it is not always the case. In our situation, the application ID (AppID in the SearchRequest complex type) must be 40 characters long. We need to change the COBOL picture from X(32) to X(40).

COBOL Transformers Generation for a Proxy

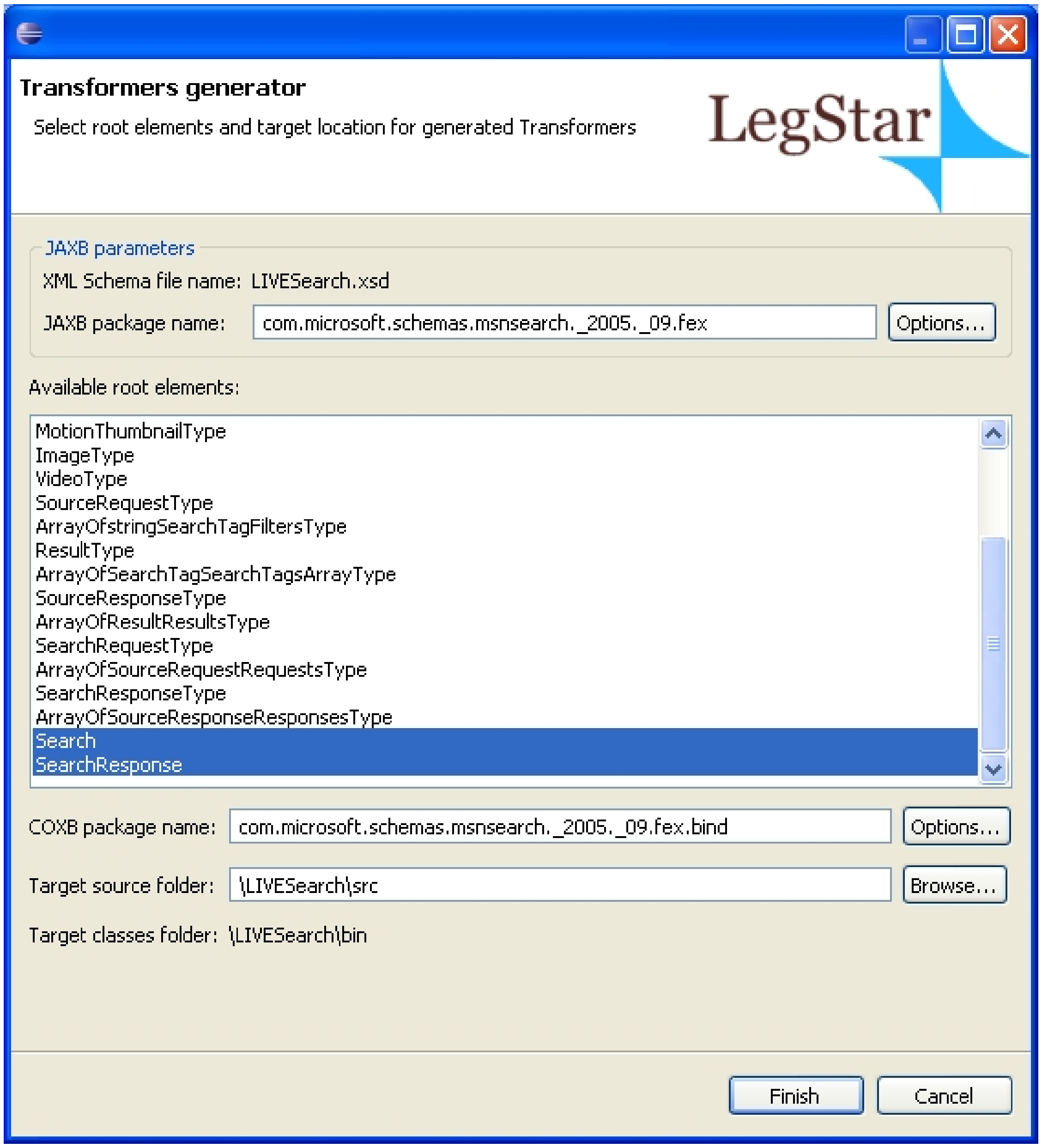
The wizard is started from the package explorer, by right clicking on a previously generated XML Schema and then selecting LegStar->Generate Transformers:

Figure 4.19. Proxy COBOL Transformers generation menu



In this case, the root structures we are interested in are Search and SearchResponse, which are the wrapper elements expected and produced by the target Web Service. We select them both.

Figure 4.20. Proxy COBOL Transformers generation screen parameters



Transformers generator

Select root elements and target location for generated Transformers

JAXB parameters

XML Schema file name: LIVESearch.xsd

JAXB package name: com.microsoft.schemas.msnsearch._2005._09.fex

Available root elements:

MotionThumbnailType
ImageType
VideoType
SourceRequestType
ArrayOfstringSearchTagFiltersType
ResultType
ArrayOfSearchTagSearchTagsArrayType
SourceResponseType
ArrayOfResultResultsType
SearchRequestType
ArrayOfSourceRequestRequestsType
SearchResponseType
ArrayOfSourceResponseResponsesType
Search
SearchResponse

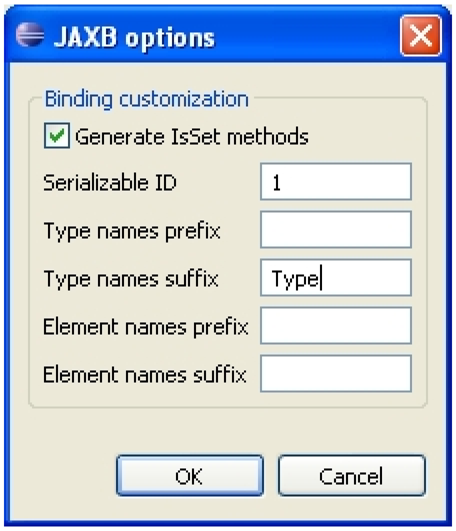
COXB package name: com.microsoft.schemas.msnsearch._2005._09.fex.bind

Target source folder: \LIVESearch\src

Target classes folder: \LIVESearch\bin

Because Microsoft uses the same names for both Elements and Complex Types, JAXB may complain about name conflicts. To avoid this, you can customize JAXB using the upper options button and specify that all Complex Types should be suffixed with characters "Type":

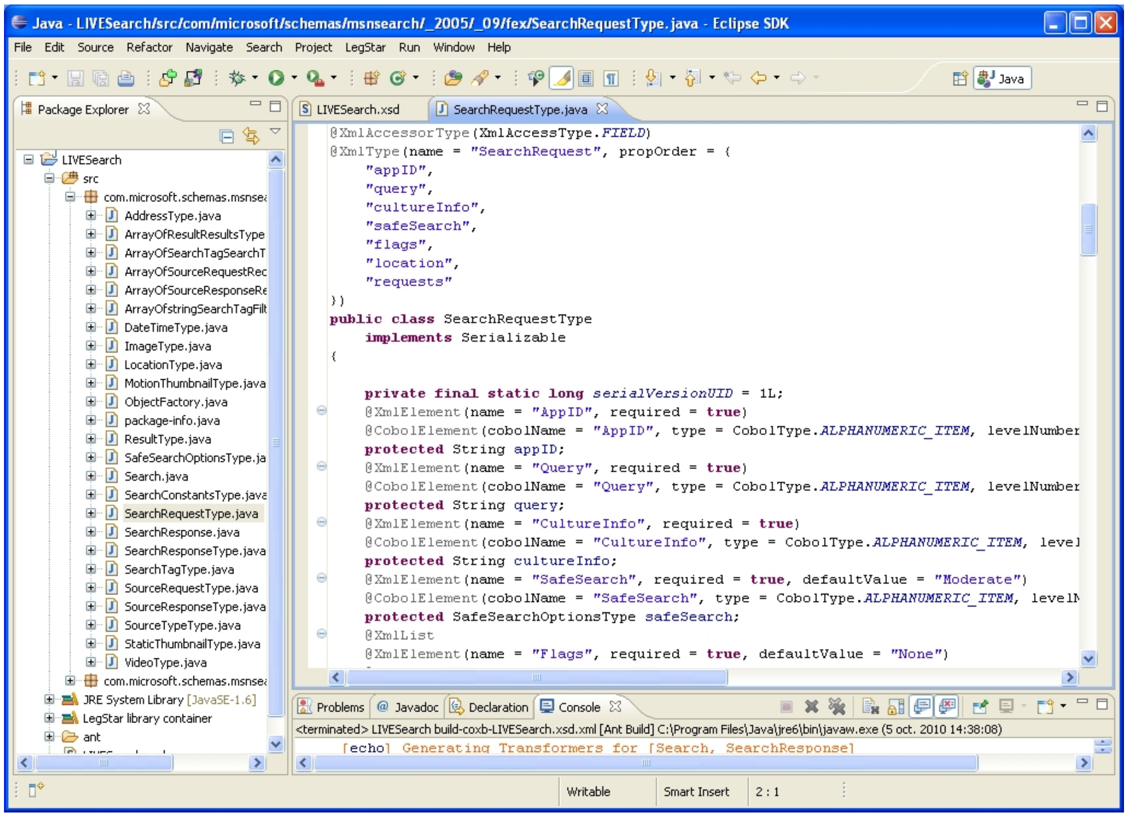
Figure 4.21. Proxy COBOL Transformers generation screen JAXB options



Generate IsSet methods	✓
Serializable ID	1
Type names prefix	
Type names suffix	Type
Element names prefix	
Element names suffix	

After you click finish, two Java packages are created, one for JAXB classes with COBOL annotations and one for the optimized Transformers classes.

Figure 4.22. Proxy COBOL Transformers generation screen results



```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "SearchRequest", propOrder = {
    "appID",
    "query",
    "cultureInfo",
    "safeSearch",
    "flags",
    "location",
    "requests"
})
public class SearchRequestType
    implements Serializable
{

    private final static long serialVersionUID = 1L;
    @XmlElement(name = "AppID", required = true)
    @CobolElement(cobolName = "AppID", type = CobolType.ALPHANUMERIC_ITEM, levelNumber
    protected String appID;
    @XmlElement(name = "Query", required = true)
    @CobolElement(cobolName = "Query", type = CobolType.ALPHANUMERIC_ITEM, levelNumber
    protected String query;
    @XmlElement(name = "CultureInfo", required = true)
    @CobolElement(cobolName = "CultureInfo", type = CobolType.ALPHANUMERIC_ITEM, level
    protected String cultureInfo;
    @XmlElement(name = "SafeSearch", required = true, defaultValue = "Moderate")
    @CobolElement(cobolName = "SafeSearch", type = CobolType.ALPHANUMERIC_ITEM, levelN
    protected SafeSearchOptionsType safeSearch;
    @XmlList
    @XmlElement(name = "Flags", required = true, defaultValue = "None")
```

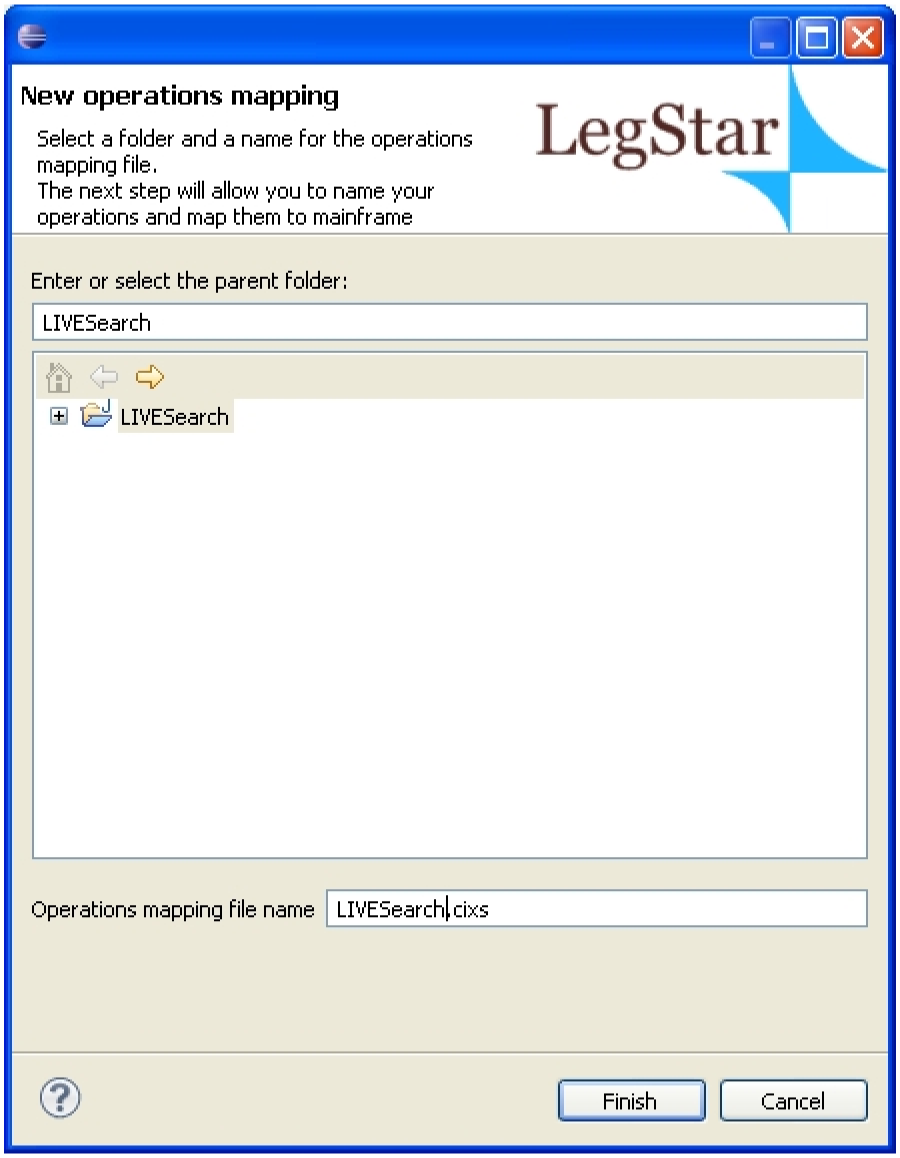
```
<terminated> LIVESearch build-coxb-LIVESearch.xsd.xml [Ant Build] C:\Program Files\Java\jre6\bin\javaw.exe (5 oct. 2010 14:38:08)
     [echo] Generating Transformers for [Search, SearchResponse]
```

The Using generated tranformers section gives examples of code you could write to run Transformers.

Mainframe Proxy generation

The last step is also similar to the previous use case. We start by creating a mapping to the Web Service operation using the LegStar->New operations mapping… option:

Figure 4.23. Proxy service generation screen new operation



New operations mapping

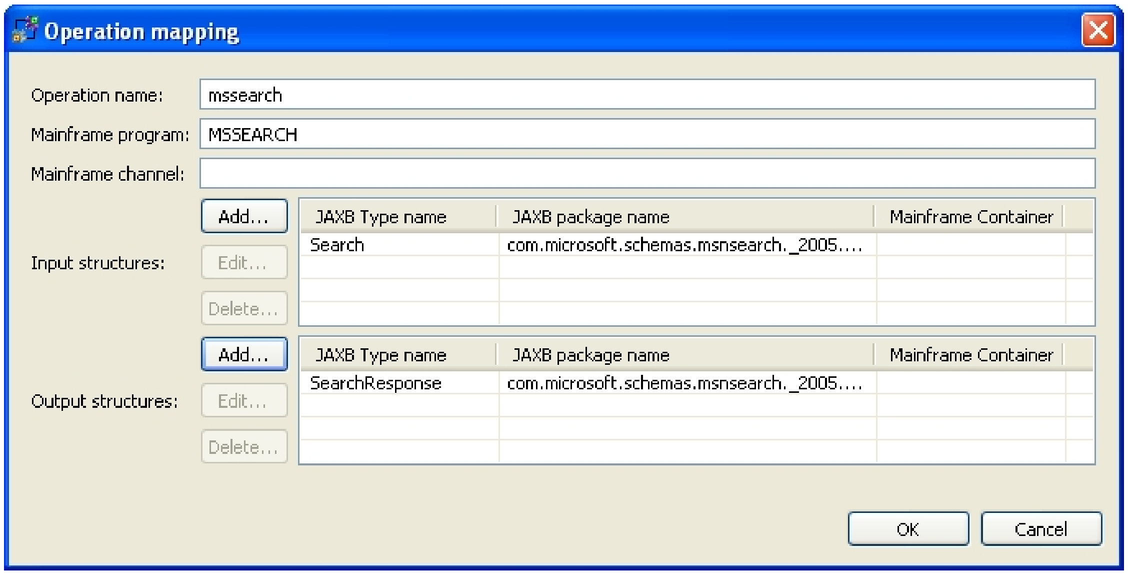
Select a folder and a name for the operations mapping file.
The next step will allow you to name your operations and map them to mainframe

Enter or select the parent folder: LIVESearch

Operations mapping file name: LIVESearch.cixs

We now add an operation with the following characteristics:

Figure 4.24. Proxy service generation screen operation mapping done



Operation name: mssearch

Mainframe program: MSSEARCH

Mainframe channel:

Input structures:

JAXB Type name	JAXB package name	Mainframe Container
Search	com.microsoft.schemas.msnsearch._2005....	

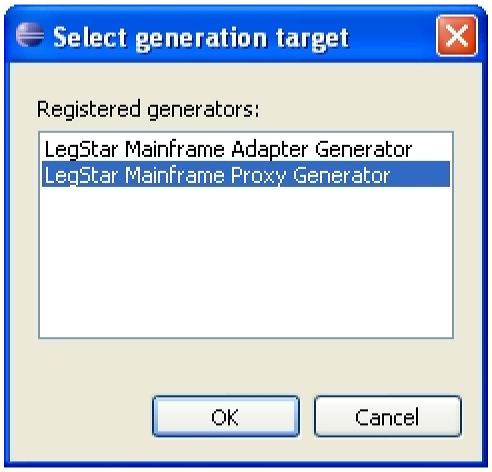
Output structures:

JAXB Type name	JAXB package name	Mainframe Container
SearchResponse	com.microsoft.schemas.msnsearch._2005....	

The mainframe program in this case is a sample COBOL CICS program that will be generated with the name that you specify here.

Once you are back on the operations mapping editor, you can click on generate. You can now select the mainframe proxy option:

Figure 4.25. Proxy service generation screen select target

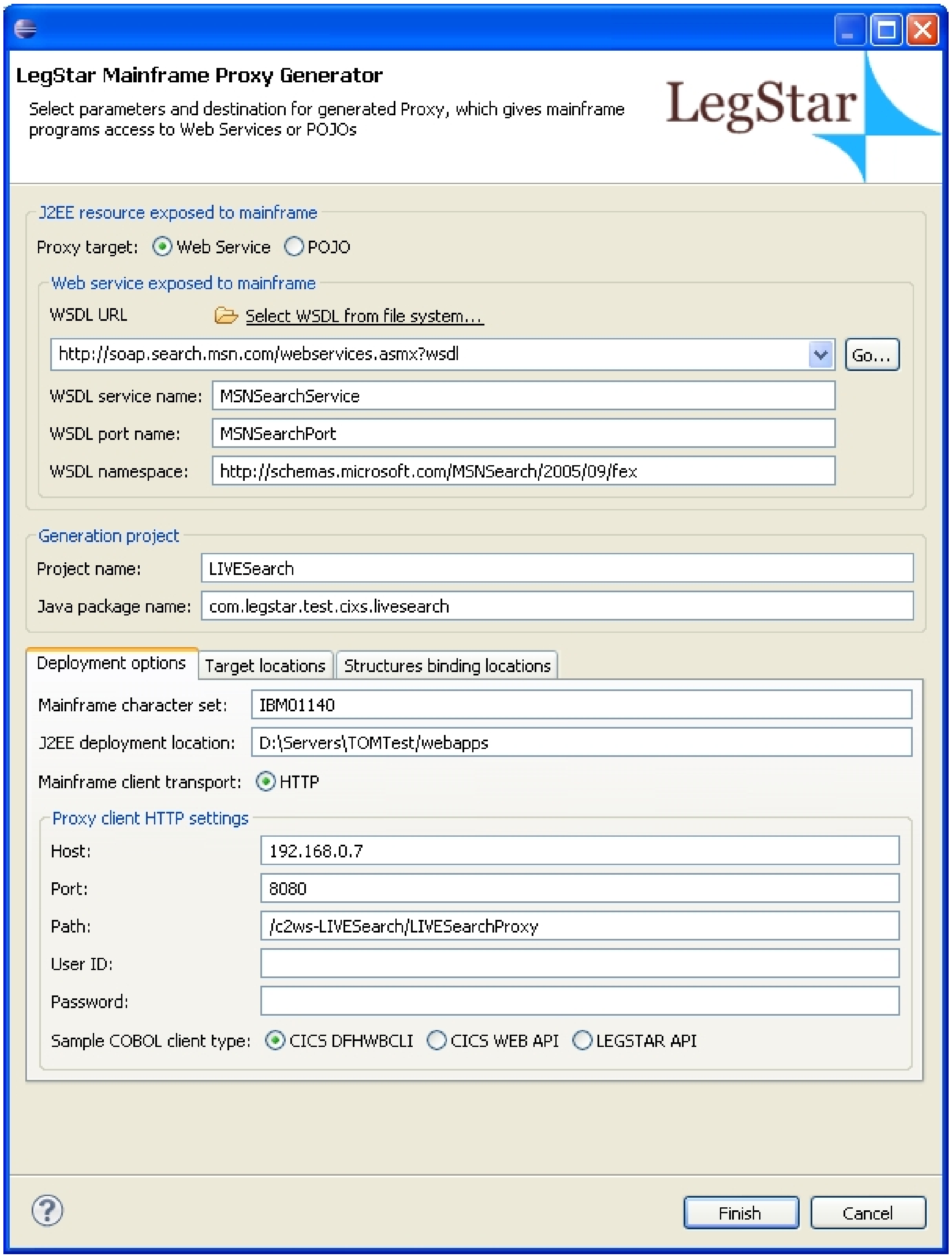


Registered generators:

LegStar Mainframe Adapter Generator
LegStar Mainframe Proxy Generator

The generator dialog will ask you for the target Web Service runtime characteristics. This is needed to allow the proxy to address the target Web Service at runtime. You can query these parameters from the WSDL again by entering the URL and clicking on go. If the target WSDL has more than one service or port, you will have to select one:

Figure 4.26. Proxy service generation screen parameters



LegStar Mainframe Proxy Generator

Select parameters and destination for generated Proxy, which gives mainframe programs access to Web Services or POJOs

J2EE resource exposed to mainframe

Proxy target: [✓ Web Service] POJO

Web service exposed to mainframe

WSDL URL — Select WSDL from file system...

http://soap.search.msn.com/webservices.asmx?wsdl

WSDL service name: MSNSearchService

WSDL port name: MSNSearchPort

WSDL namespace: http://schemas.microsoft.com/MSNSearch/2005/09/fex

Generation project

Project name: LIVESearch

Java package name: com.legstar.test.cixs.livesearch

Deployment options | Target locations | Structures binding locations

Mainframe character set: IBM01140

J2EE deployment location: D:\Servers\TOMTest/webapps

Mainframe client transport: [✓ HTTP]

Proxy client HTTP settings

Host: 192.168.0.7

Port: 8080

Path: /c2ws-LIVESearch/LIVESearchProxy

User ID:

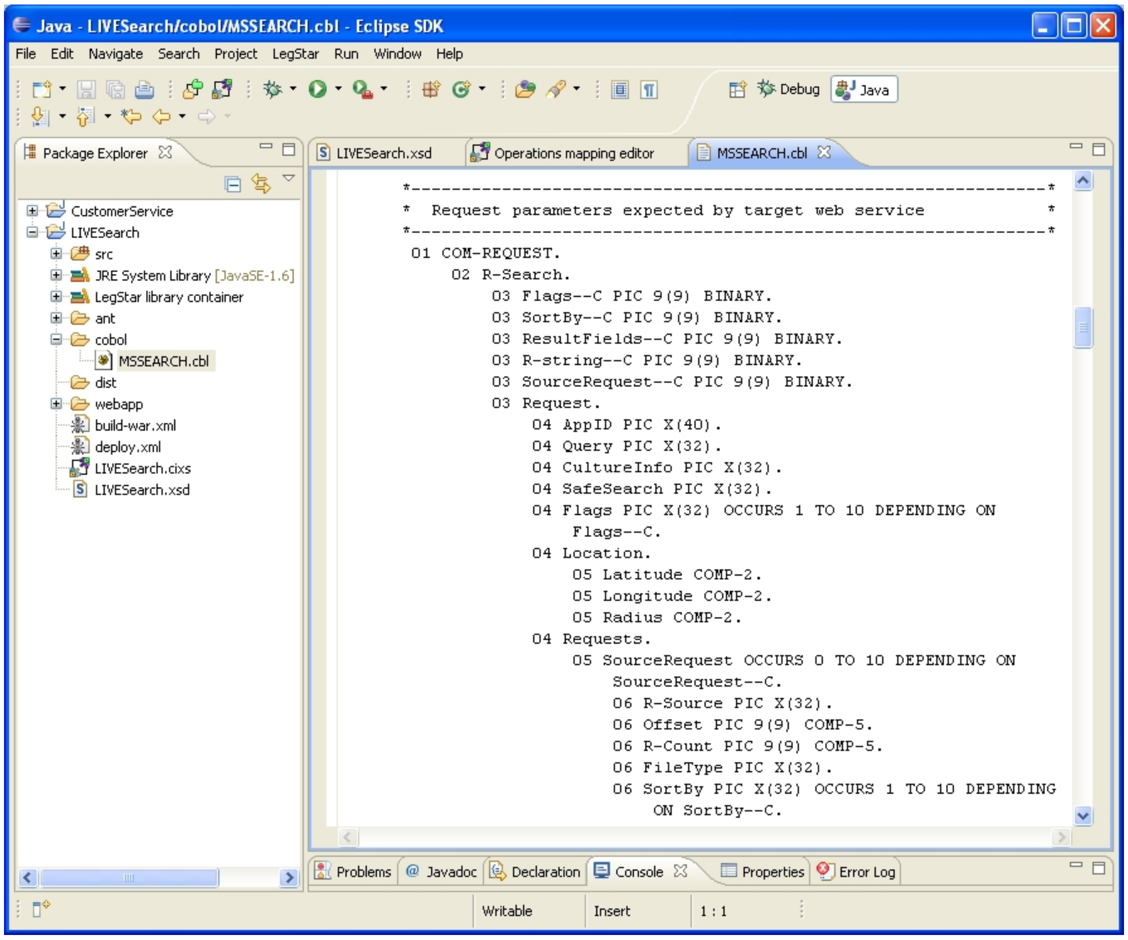
Password:

Sample COBOL client type: [✓ CICS DFHWBCLI] CICS WEB API LEGSTAR API

Clicking the finish button will create various artifacts. The Mainframe proxy is Servlet to be deployed in a J2EE container. The implementation uses Sun's JAX-WS RI (Metro) as a Web Service client. The build-war.xml ant file that is generated allows you to bundle the servlet ready for deployment.

The generator also creates a sample COBOL program that behaves as an HTTP client. The search structure shows up in the working storage section:

Figure 4.27. Proxy service generation screen results



```
      *-----------------------------------------------------------------*
      *  Request parameters expected by target web service              *
      *-----------------------------------------------------------------*
       01 COM-REQUEST.
           02 R-Search.
               03 Flags--C PIC 9(9) BINARY.
               03 SortBy--C PIC 9(9) BINARY.
               03 ResultFields--C PIC 9(9) BINARY.
               03 R-string--C PIC 9(9) BINARY.
               03 SourceRequest--C PIC 9(9) BINARY.
               03 Request.
                   04 AppID PIC X(40).
                   04 Query PIC X(32).
                   04 CultureInfo PIC X(32).
                   04 SafeSearch PIC X(32).
                   04 Flags PIC X(32) OCCURS 1 TO 10 DEPENDING ON
                       Flags--C.
                   04 Location.
                       05 Latitude COMP-2.
                       05 Longitude COMP-2.
                       05 Radius COMP-2.
                   04 Requests.
                       05 SourceRequest OCCURS 0 TO 10 DEPENDING ON
                           SourceRequest--C.
                           06 R-Source PIC X(32).
                           06 Offset PIC 9(9) COMP-5.
                           06 R-Count PIC 9(9) COMP-5.
                           06 FileType PIC X(32).
                           06 SortBy PIC X(32) OCCURS 1 TO 10 DEPENDING
                               ON SortBy--C.
```

The sample COBOL source contains TODO comments to help you locate where you should set values for the search request and where you can display the results. The LIVE search service requires a developer ID that you can get for free and enter in the AppID field.

This is an example of code to set the search structures properly:

```
           MOVE ZERO TO Flags--C OF COM-REQUEST.
           MOVE ZERO TO SortBy--C OF COM-REQUEST.
           MOVE ZERO TO ResultFields--C OF COM-REQUEST.
           MOVE ZERO TO R-string--C OF COM-REQUEST.
           MOVE 1 TO SourceRequest--C OF COM-REQUEST.
      *  You should specify your own Microsoft LIVE application ID
           MOVE '5588C3ACE949317B3ECAADDQ908611BDF5D8D5ZA'
             TO AppID OF COM-REQUEST.
           MOVE 'Mainframe' TO Query OF COM-REQUEST.
           MOVE 'en-US' TO CultureInfo OF COM-REQUEST.
           MOVE 'Moderate' to SafeSearch OF COM-REQUEST.
           MOVE ZERO TO Latitude OF COM-REQUEST.
           MOVE ZERO TO Longitude OF COM-REQUEST.
           MOVE ZERO TO Radius OF COM-REQUEST.
           MOVE 'Web' TO R-Source OF COM-REQUEST(1).
           MOVE ZERO TO Offset OF COM-REQUEST(1).
           MOVE 1 TO R-Count OF COM-REQUEST(1).
           MOVE SPACES TO FileType OF COM-REQUEST(1).
```

And these are lines to display the result:

```
           STRING 'INVOKE-SERVICE success. First hit is '
                  DELIMITED BY SIZE
                  Description OF COM-REPLY(1, 1)
                  DELIMITED BY SIZE
                  INTO ERROR-MESSAGE.
           EXEC CICS SEND TEXT FROM(ERROR-MESSAGE) FREEKB END-EXEC.
           DISPLAY 'Response data length=' WBCLI-RESPONSE-BODY-LEN.

           DISPLAY 'SourceResponse--C ='
                    SourceResponse--C OF COM-REPLY.
           DISPLAY 'R-Source(1)=' R-Source OF COM-REPLY(1).
           DISPLAY 'Total(1)=' Total OF COM-REPLY(1).
           DISPLAY 'R-Title(1, 1)=' R-Title OF COM-REPLY(1, 1).
           DISPLAY 'Description(1, 1)='
                    Description OF COM-REPLY(1, 1).
```

After you add these lines of code, you should be able to upload the source onto your mainframe and get it compiled and defined to your CICS region. Please note that this program calls the CICS DFHWBCLI program defined in the CICS standard DFHWEB group. Alternatively, LegStar supports the new EXEC CICS WEB API or even supports older version of CICS with its own HTTP library.

Chapter 5. Wrap up

This document has described some of the most common use cases for LegStar. The product being modular, there are many possibilities to combine the features to satisfy more use cases.

LegStar is a community effort and we encourage you to send your feedback on the mailing list [http://groups.google.com/group/legstar-user].

The source code for LegStar is on Google Code [http://code.google.com/p/legstar/]. You can freely use SVN to access the code base.

You can search bug reports [http://code.google.com/p/legstar/issues/list] and create new ones when needed.

There is more detailed info on the wiki pages [http://code.google.com/p/legstar/w/list] that you can contribute to enhance.

Appendix A. Code snippets

Using generated tranformers

Running Host/Java Transformers

The COBOL Transformers Generator produces a set of java classes that you can easily use to turn mainframe payloads to java data objects.

This is sample code showing how you would use a generated mainframe to java transformer assuming you just generated a transformer class called com.legstar.test.coxb.lsfileae.DfhcommareaTransformers.

```
/**
 * Transform host data and test java data object result.
 *
 * @param hostBytes a byte array holding the mainframe payload
 * @throws HostTransformException if transforming fails
 */
public void hostToJavaTransform(final byte[] hostBytes)
        throws HostTransformException {

    DfhcommareaTransformers transformers = new DfhcommareaTransformers();
    Dfhcommarea dfhcommarea = transformers.toJava(hostBytes);
    assertEquals(100, dfhcommarea.getComNumber());
    assertEquals("TOTO", dfhcommarea.getComPersonal().getComName().trim());
    assertEquals("LABAS STREET", dfhcommarea.getComPersonal()
            .getComAddress().trim());
    assertEquals("88993314", dfhcommarea.getComPersonal().getComPhone()
            .trim());
    assertEquals("100458", dfhcommarea.getComDate().trim());
    assertEquals("00100.35", dfhcommarea.getComAmount().trim());
    assertEquals("A VOIR", dfhcommarea.getComComment().trim());
}
```

Conversely, you would produce a byte array with mainframe data from a java data object with code similar to this:

```
/**
 * Creates a java data object and returns the host data result.
 *
 * @return a byte array holding the mainframe payload
 * @throws HostTransformException if transforming fails
 */
public byte[] javaToHostTransform() throws HostTransformException {
    Dfhcommarea dfhcommarea = new Dfhcommarea();
    dfhcommarea.setComNumber(100L);
    ComPersonal comPersonal = new ComPersonal();
    comPersonal.setComName("TOTO");
    comPersonal.setComAddress("LABAS STREET");
    comPersonal.setComPhone("88993314");
```

```
        dfhcommarea.setComPersonal(comPersonal);
        dfhcommarea.setComDate("100458");
        dfhcommarea.setComAmount("00100.35");
        dfhcommarea.setComComment("A VOIR");
        DfhcommareaTransformers transformers = new DfhcommareaTransformers();
        return transformers.toHost(dfhcommarea);
    }
```

Generated transformers use the default IBM01140 US EBCDIC character set for conversions.

Methods toHost and toJava also accept a character set name as a second parameter if you need to use a different one (just make sure your JRE charsets.jar supports your character set).

Running Host/XML Transformers

In addition to Host/Java transformers, you can generate Host/XML transformers by turning the xmlTransformers generation option on.

Using these transformers, this is sample code to turn host data to XML:

```
    /**
     * Transform host data and test XML result.
     *
     * @param hostBytes a byte array holding the mainframe payload
     * @throws HostTransformException if transforming fails
     */
    public void hostToXmlTransform(final byte[] hostBytes)
            throws HostTransformException {

        DfhcommareaXmlTransformers transformers =
                new DfhcommareaXmlTransformers();
        StringWriter writer = new StringWriter();
        transformers.toXml(hostBytes, writer);
        assertEquals(
                "<?xml version=\"1.0\" encoding=\"UTF-8\" "
                        + "standalone=\"yes\"?>"
                        + "<Dfhcommarea xmlns="
                        + "\"http://legstar.com/test/coxb/lsfileae\">"
                        + "<ComNumber>100</ComNumber>"
                        + "<ComPersonal>"
                        + "<ComName>TOTO</ComName>"
                        + "<ComAddress>LABAS STREET</ComAddress>"
                        + "<ComPhone>88993314</ComPhone>"
                        + "</ComPersonal>"
                        + "<ComDate>100458</ComDate>"
                        + "<ComAmount>00100.35</ComAmount>"
                        + "<ComComment>A VOIR</ComComment>"
                        + "</Dfhcommarea>", writer.toString());
    }
```

This is code to turn XML into host data:

```
/**
 * Turns an XML into host data.
 *
 * @return a byte array holding the mainframe payload
 * @throws HostTransformException if transforming fails
 */
public byte[] xmlToHostTransform() throws HostTransformException {
    StringReader reader = new StringReader(
            "<?xml version=\"1.0\" encoding=\"UTF-8\" "
                    + "standalone=\"yes\"?>"
                    + "<Dfhcommarea xmlns="
                    + "\"http://legstar.com/test/coxb/lsfileae\">"
                    + "<ComNumber>100</ComNumber>"
                    + "<ComPersonal>"
                    + "<ComName>TOTO</ComName>"
                    + "<ComAddress>LABAS STREET</ComAddress>"
                    + "<ComPhone>88993314</ComPhone>"
                    + "</ComPersonal>"
                    + "<ComDate>100458</ComDate>"
                    + "<ComAmount>00100.35</ComAmount>"
                    + "<ComComment>A VOIR</ComComment>"
                    + "</Dfhcommarea>");
    DfhcommareaXmlTransformers transformers =
            new DfhcommareaXmlTransformers();
    return transformers.toHost(new StreamSource(reader));
}
```

Running Host/JSON Transformers

In addition to Host/Java transformers, you can generate Host/JSON transformers by turning the jsonTransformers generation option on.

Using these transformers, this is sample code to turn host data to JSON:

```
/**
 * Transform host data and test JSON result.
 *
 * @param hostBytes a byte array holding the mainframe payload
 * @throws HostTransformException if transforming fails
 */
public void hostToJsonTransform(final byte[] hostBytes)
        throws HostTransformException {

    DfhcommareaJsonTransformers transformers =
            new DfhcommareaJsonTransformers();
    StringWriter writer = new StringWriter();
    transformers.toJson(hostBytes, writer);
    assertEquals("{\"ComNumber\":100,"
            + "\"ComPersonal\":"
            + "{\"ComName\":\"TOTO\","
```

```
                + "\"ComAddress\":\"LABAS STREET\","
                + "\"ComPhone\":\"88993314\"},"
                + "\"ComDate\":\"100458\","
                + "\"ComAmount\":\"00100.35\","
                + "\"ComComment\":\"A VOIR\"}",
                writer.toString());
    }
```

This is code to turn JSON into host data:

```
    /**
     * Turns JSON into host data.
     *
     * @return a byte array holding the mainframe payload
     * @throws HostTransformException if transforming fails
     */
    public byte[] jsonToHostTransform() throws HostTransformException {
        StringReader reader = new StringReader(
                "{\"ComNumber\":100,"
                        + "\"ComPersonal\":"
                        + "{\"ComName\":\"TOTO\","
                        + "\"ComAddress\":\"LABAS STREET\","
                        + "\"ComPhone\":\"88993314\"},"
                        + "\"ComDate\":\"100458\","
                        + "\"ComAmount\":\"00100.35\","
                        + "\"ComComment\":\"A VOIR\"}");
        DfhcommareaJsonTransformers transformers =
                new DfhcommareaJsonTransformers();
        return transformers.toHost(reader);
    }
```