

LegSem
November 2008

LegStar

Presentation

I. Introduction

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.

LegStar 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 runtimes such as Apache Tomcat or Jetty.

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 on 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 mainframe integration activities can be separated into:

- Data binding activities where data streams in mainframe format are converted 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 for instance.

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 sub-systems 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, 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.

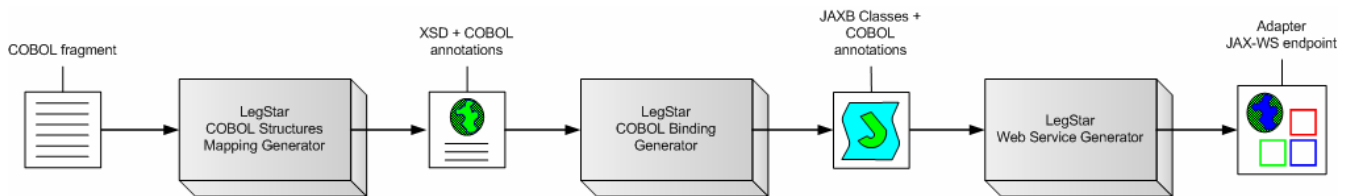
LegStar is modular so that features can be selected and combined as necessary.

II. Architecture

Expose a COBOL program as a Web Service

Development

Developers would follow these three steps to "Service-enable" a COBOL program:



In this use case, initial COBOL code fragments describe the legacy program input and output structures. When the COBOL program is a CICS Container-driven program there can actually be several such input structures and several output structures each described by a different code fragment. This step is repeated as necessary for each COBOL code fragment involved.

The LegStar **COBOL Structures Mapping Generator** 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.

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 Binding Generator** takes a Mapping XML Schema as input and produces annotated Java Classes (using Java annotations introduced in J2SE 1.5). More precisely, LegStar uses the standard Java to XML Binding (JAXB) framework during this phase.

These generated classes are all that is needed for the runtime to perform XML to mainframe data marshaling/unmarshaling. Mainframe data, described by the initial COBOL fragment, is typically encoded in a mainframe character set (EBCDIC). Conversion is totally bi-directional and completely independent of the origin or destination of the host data.

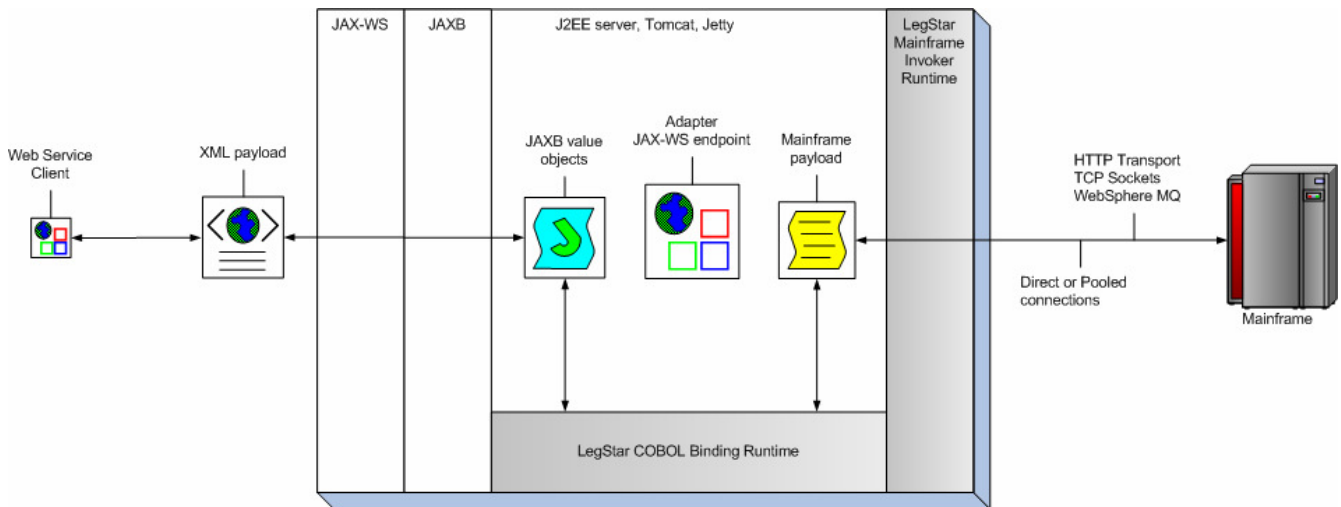
The LegStar **Web Service Generator**, maps a mainframe program to a Web Service operation. The current version of LegStar supports CICS programs either Commarea or Container driven.

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 (or use LegStar Complex Types Mapping Generator) 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. Ant scripts are for pure java developers and plug-ins are for the Eclipse IDE. Moving forward, the Eclipse plug-ins are the recommended tool as the parameter set needed by generators keeps increasing.

Runtime

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



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 convert java value objects to a mainframe payload. This conversion 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 Invoker Runtime** to invoke a remote program.

The LegStar **Messaging Protocol** used by the Mainframe Invoker is binary and implements a request/reply exchange pattern. It is designed to reduce the payload size and support multiple input/output named structures such as CICS containers.

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

- Socket connectivity
- HTTP connectivity
- WebSphere MQ connectivity

Any of these transport options 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.

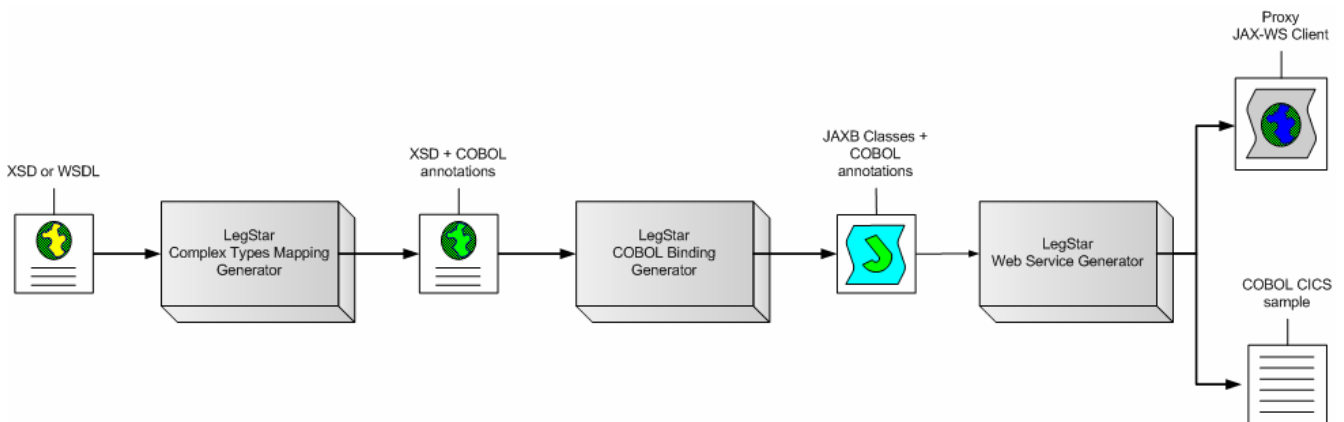
On the way back, mainframe data is converted to XML and then wrapped in a SOAP reply.

The CICS footprint of this architecture is minimal since all SOAP/XML processing occur off-host.

Consume a Web Service from a COBOL program

Development

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



The LegStar **Complex Types Mapping Generator** takes a WSDL or XML Schema file as input and produces a Mapping XML Schema with COBOL annotations.

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 Binding Generator** is the same as the one we went through in the Web Service generation use-case. The result is a set of annotated JAXB classes, which provides the conversion capabilities from mainframe data to XML.

The third step, the LegStar **Web Service Generator**, already seen in the previous use-case is used here to produce a Servlet Proxy and COBOL CICS sample program. This Proxy acts as an intermediary between the mainframe client program and the target Web Service at runtime.

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

Runtime

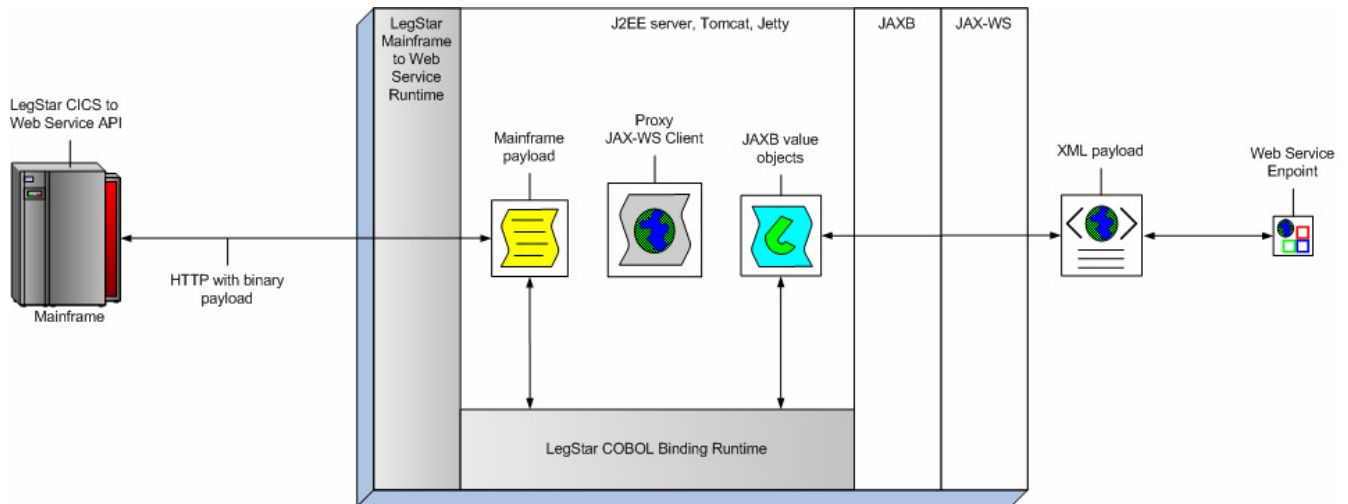
LegStar provides a CICS to Web Service API as part of the **Mainframe to Web Service Runtime** that programmers can use from their COBOL programs to invoke the remote Web Service. The generated COBOL sample highlights how to use that API.

The same LegStar **Messaging Protocol** as the previous use case is used. The difference is that the HTTP Transport is the only one available for now.

The CICS program does not directly call the target Web Service. Rather, the generated Servlet 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.

Conversion from mainframe format to XML is performed by the LegStar **COBOL Binding runtime**.

The request would flow as depicted in the following diagram:



The Servlet Proxy uses the standard JAX-WS Client API to perform the call to the target Web Service. Again, use of a standard such as JAX-WS shields LegStar from the ever-increasing complexity of SOA.

III. Getting started

Installation

From the LegStar Web site at <http://www.legsem.com/legstar>, follow the Distribution menu option.

From the Distribution page:

- The Download menu option gets you the latest LegStar release.
- The Release Notes menu option gives installation instructions.

The core LegStar installation does not contain the Eclipse plugins.

To install the Eclipse plug-ins, follow the standard Eclipse update mechanism using this URL:

<http://www.legsem.com/legstar/eclipse/update>.

First development

Once the core product is installed, you can start developing. You will have to choose between the ant scripts and the Eclipse plug-ins.

The following chapter gives a good overview of the Eclipse plug-ins, which are the recommended development path.

The ant scripts can be found in the ant sub folder under your installation folder. Ant scripts are documented under each module release notes.

IV. Using the LegStar Eclipse plug-ins

This chapter gives examples of how to use the LegStar Eclipse plug-ins for the two important use cases already mentioned.

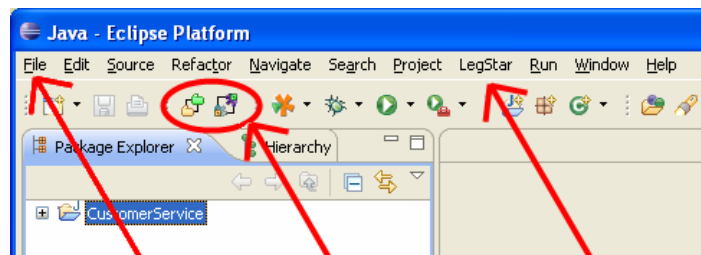
It assumes you successfully installed the core LegStar product and the Eclipse plug ins. It also assumes that you checked the Window→Preferences...→LegStar options.

Expose a COBOL program as a Web Service

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

Let's assume the project name is CustomerService.

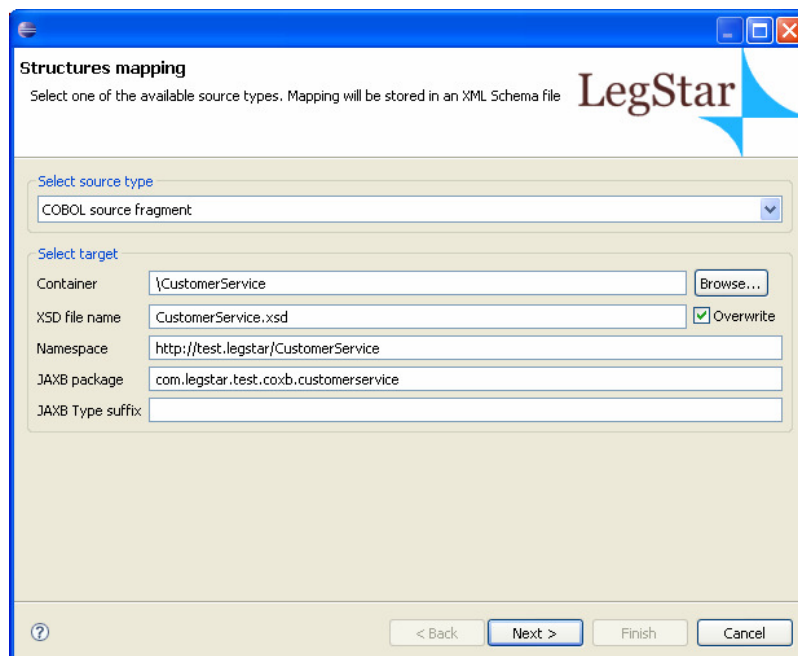
The LegStar options are available from the File→New→Other.. →LegStar or directly from the LegStar menu or toolbar buttons:



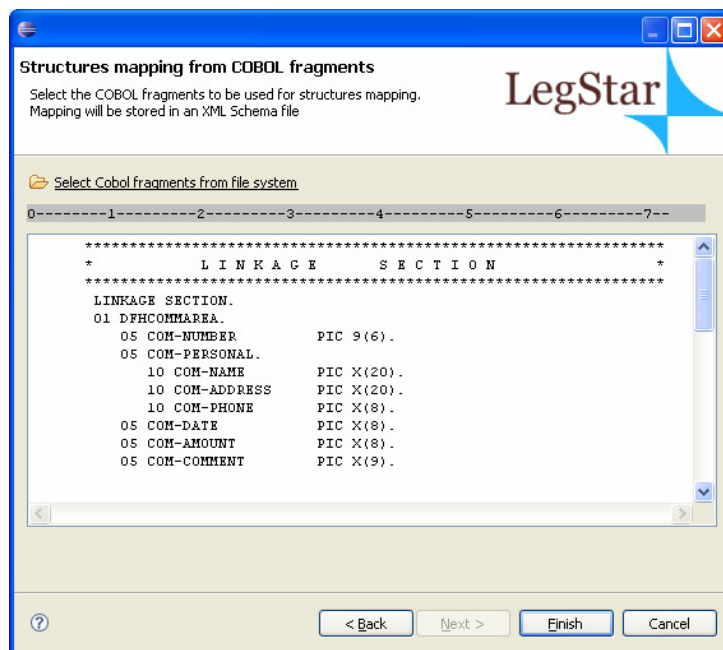
Structures Mapping:

The process starts by mapping COBOL Structures to XML Schema. This is option LegStar→New structures mapping...

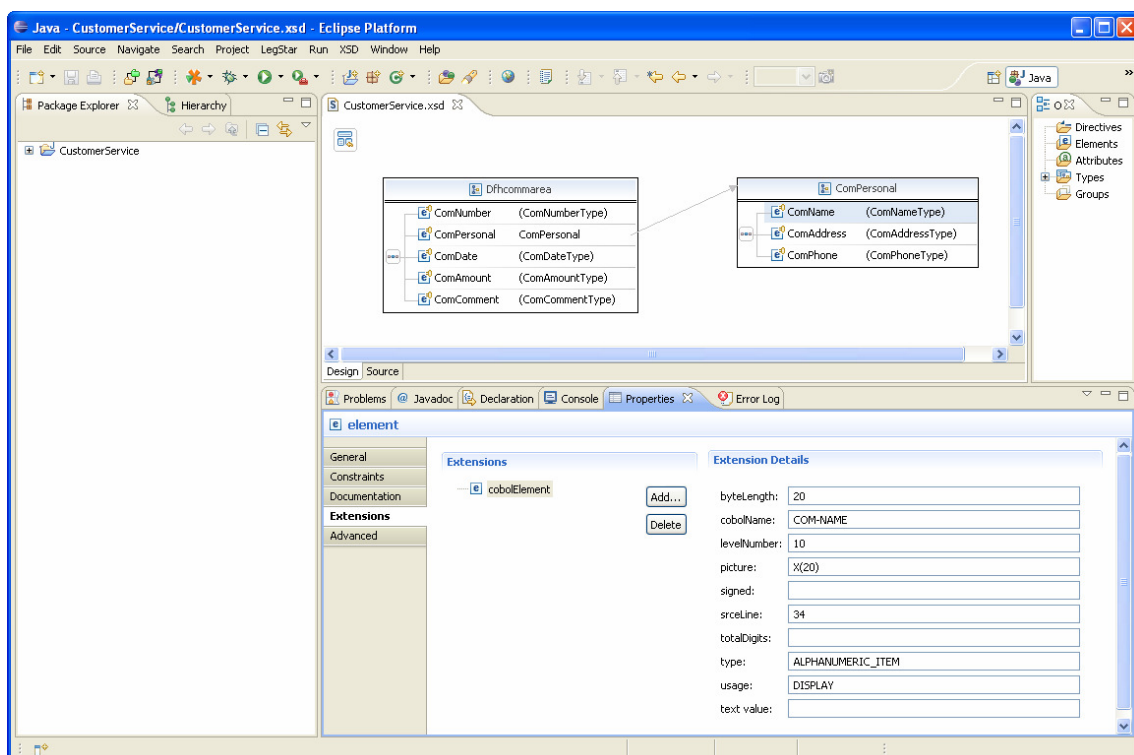
On the **Structures Mapping** plug-in first page, type an XML Schema file name making sure the extension is xsd. The source type will be COBOL for this use case:



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:



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:

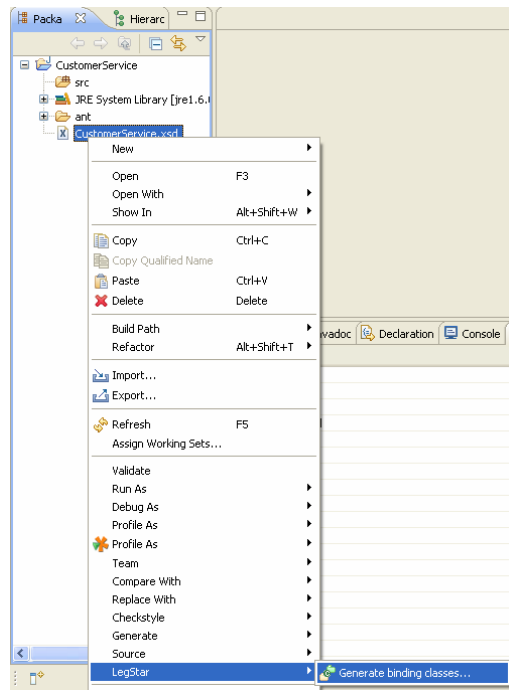


Observe the extensions used to annotate the XML Schema elements with COBOL meta-data.

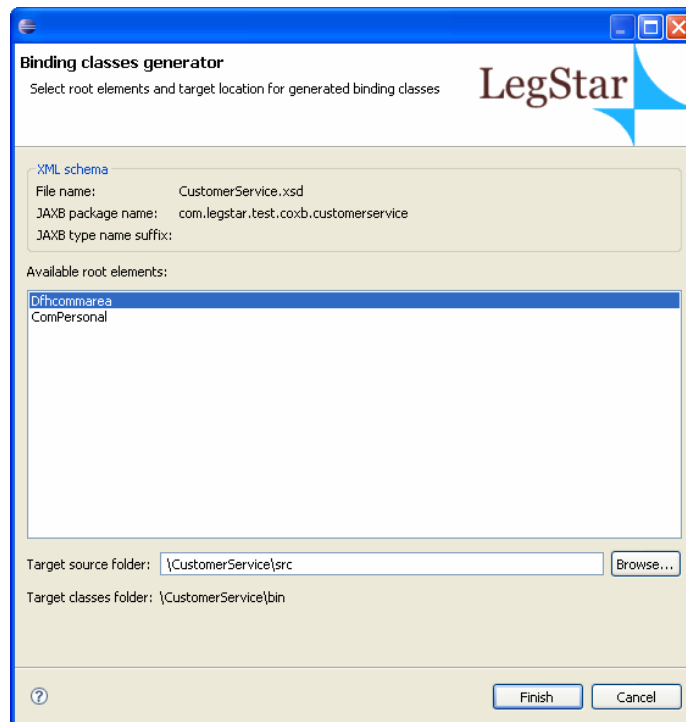
COBOL Binding classes generation:

The next step is to generate binding classes from the Mapping XML Schema. These Java classes will be responsible for marshaling/un-marshaling XML into a mainframe data.

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



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:

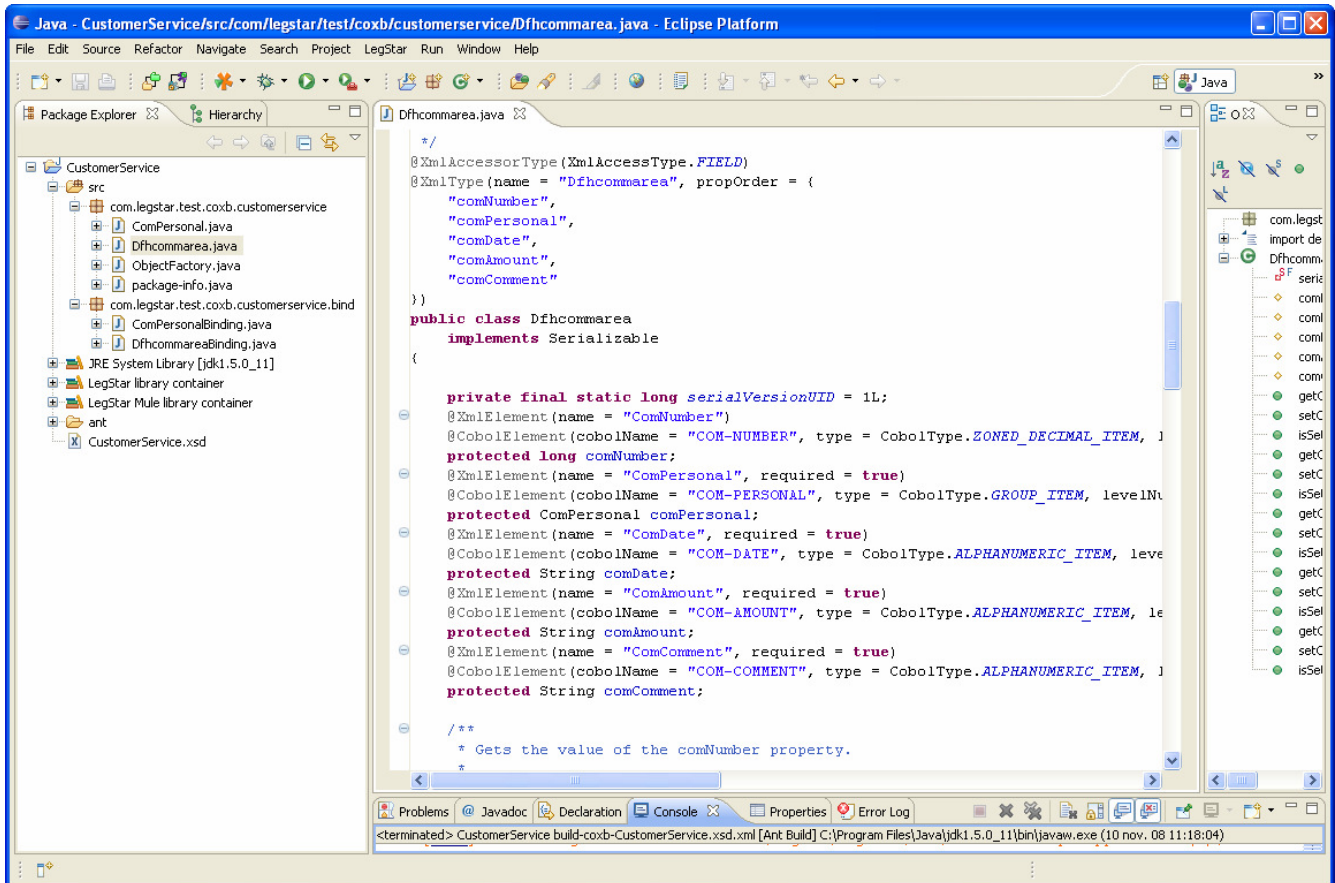


In our case, the mainframe program expects a Dfhcommarea and also produces a Dfhcommarea so that's the only element we need to select.

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 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 here:



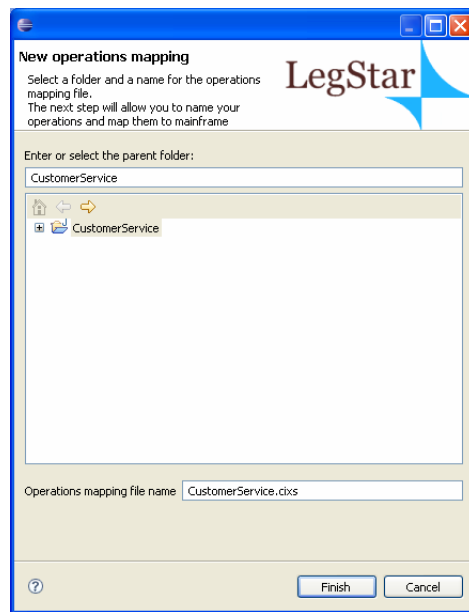
- The "com.legstar.test.coxb.CustomerService.bind" package contains runtime binding classes that can be used for fast marshaling/unmarshaling. Using these classes, there is no need to use reflection on the JAXB classes to get the COBOL meta-data.

Web Service Adapter generation:

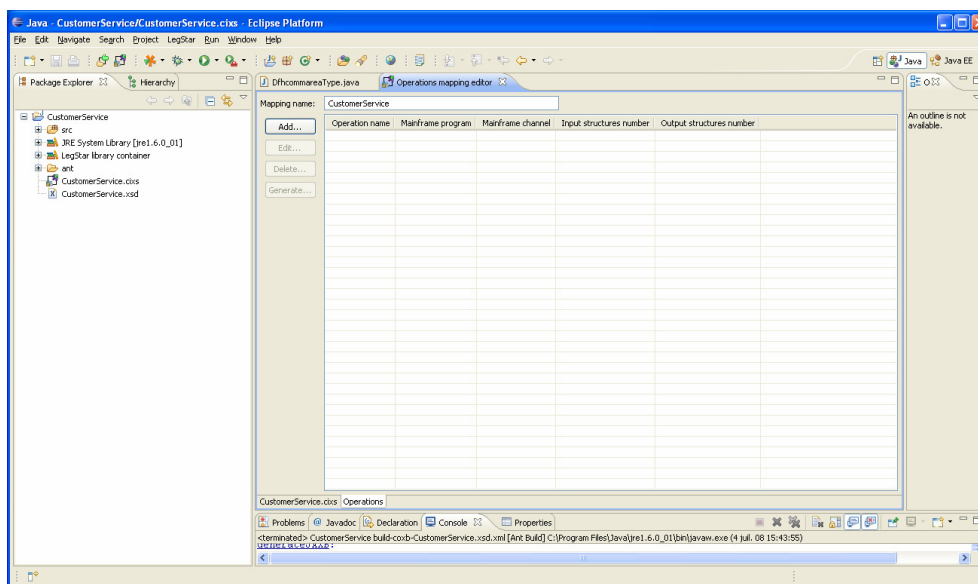
The final step in the process is to generate a JAX-WS Endpoint.

The first stage is to perform a mapping between the target mainframe program and a Web Service operation. This is option LegStar→New operations mapping...

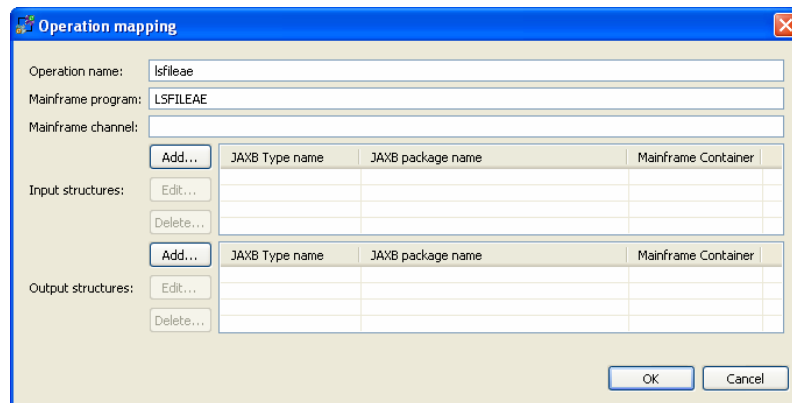
On the first page you select a name and location for the mapping file. Operations mapping files are XML files with the **cixs** extension.



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



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

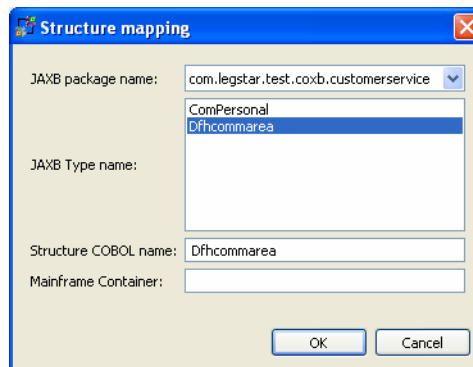


The 'Operation mapping' dialog box contains the following fields and controls:

- Operation name:
- Mainframe program:
- Mainframe channel:
- Input structures:
 -
 -
 -
 - Table with columns: JAXB Type name, JAXB package name, Mainframe Container
- Output structures:
 -
 -
 -
 - Table with columns: 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 in the previous paragraph:



The 'Structure mapping' dialog box contains the following fields and controls:

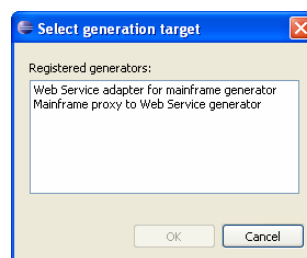
- JAXB package name:
- JAXB Type name:
- Structure COBOL name:
- Mainframe Container:
-

In our case, there is a single input and a single output so all we have to do is to select Dfhcommarea.

You can ignore the COBOL name, which will not be used in this use case.

You can select multiple input and output structures that can be mapped to CICS Containers.

Once you have specified input and output structures you can click on the generate button:

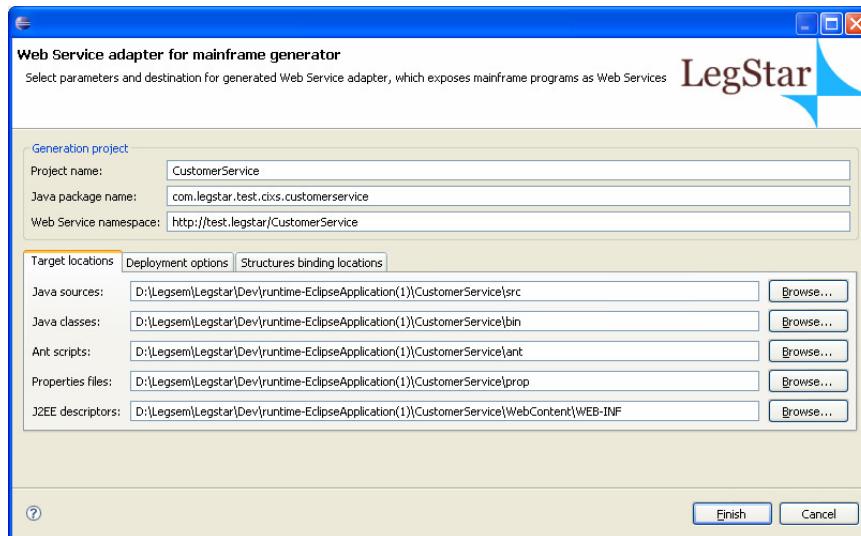


The 'Select generation target' dialog box contains the following fields and controls:

- Registered generators:
 - Web Service adapter for mainframe generator
 - Mainframe proxy to Web Service 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 targets.

In our case, we want to generate a “Web Service adapter”. When you select that target and click the OK button you get this final dialog:



The generation process needs to create various artifacts, including Java classes that implement a JAX-WS endpoint. This dialog allows you to select the target locations and other options.

Most of these options have default values derived from your preferences.

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.

An important artifact is another ant script called build.xml that you will find under the ant folder. The script allows you to deploy your endpoint to a J2EE server such as Tomcat and start testing your new Web Service adapter.

Consume a Web Service from a COBOL program

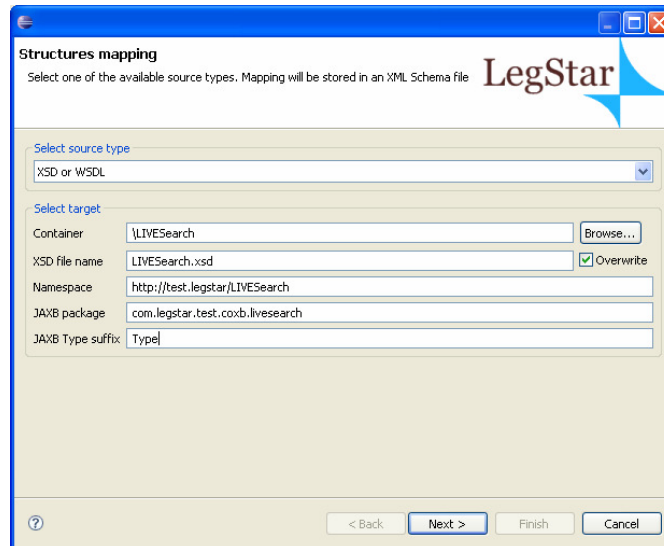
In this use case we will give a CICS program access to a Web Service.

The target Web service will be the LIVE Search API.

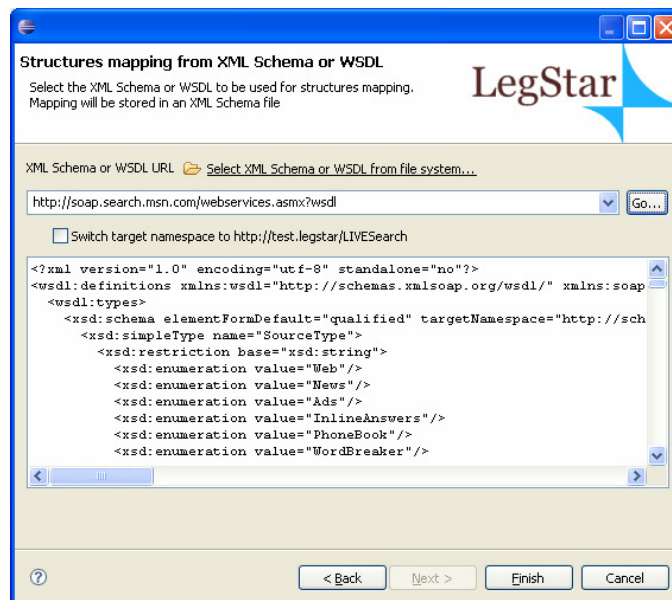
The first step is again to create a Project with a Java nature in Eclipse. We call this project LIVEsearch.

Structures Mapping:

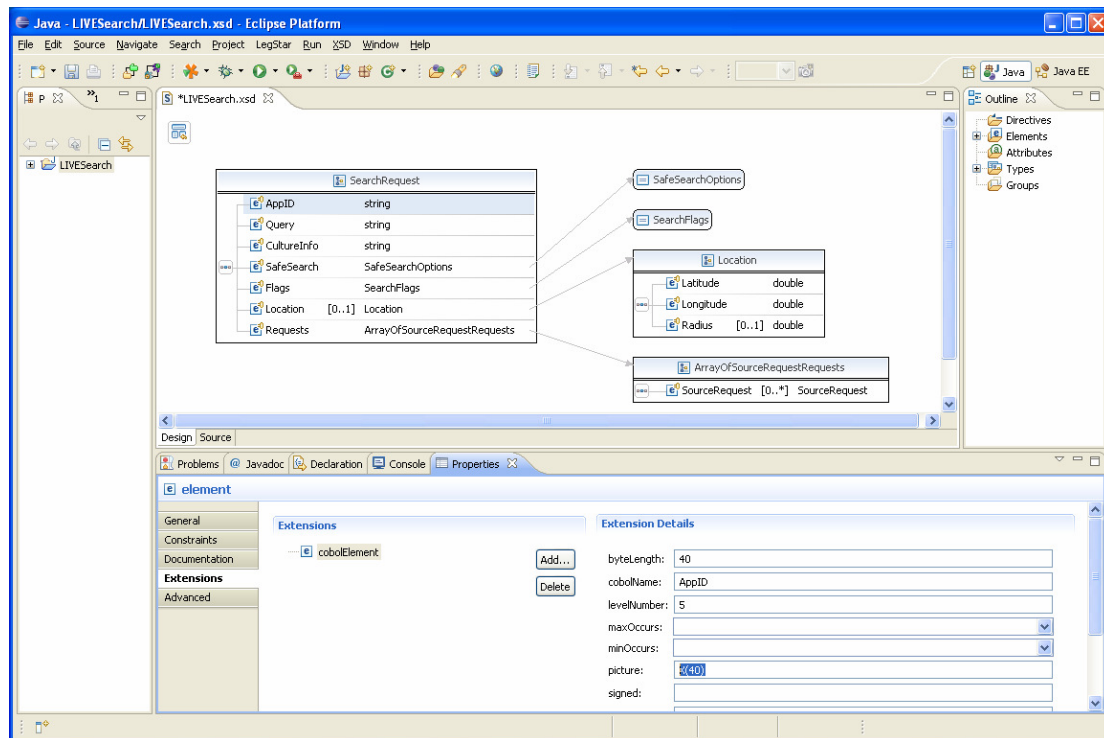
The process starts by mapping WSDL types descriptions to COBOL data items. This is option LegStar→New structures mapping...



The difference with the previous use case is that we select the XSD or WSDL source type. We also elect to have the JAXB class names suffixed with "Type". 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 and clicking on the go button:



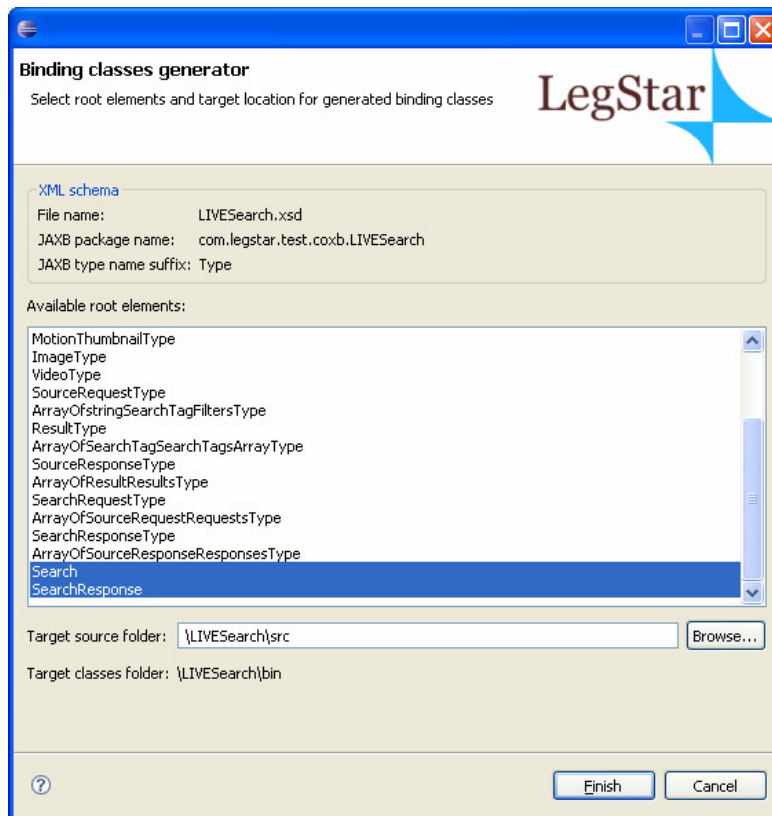
At this stage, we are ready to click on the finish button and then edit the generated mapping XML Schema:



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 case, the application ID must be 40 characters long. We need to enter 40 both for the byteLength and picture attributes (remember this is a standard XML Schema editor, it will not perform integrity checking).

COBOL Binding classes generation:

Exactly like the previous case, the next step is to generate binding classes from the Mapping XML Schema. The wizard is started from the package explorer, by right clicking on a previously generated XML Schema and then selecting LegStar→Generate Binding classes:

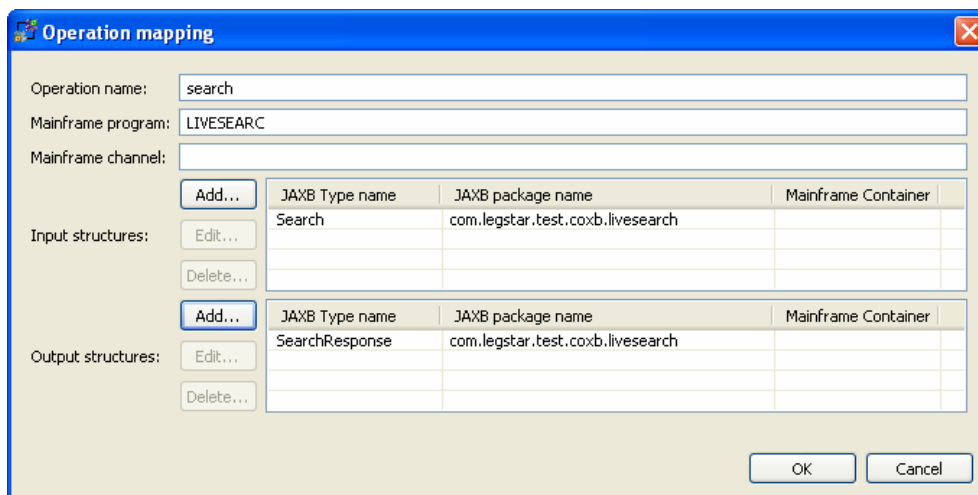


In this case, the root structures we are interested in are Search and SearchResponse, which are the wrapper elements expected and produced by the target Web Service. We select them both and click finish.

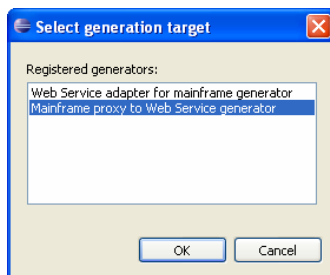
Again, two Java packages are created, one for JAXB classes and one for the optimized binding classes.

Mainframe Proxy generation:

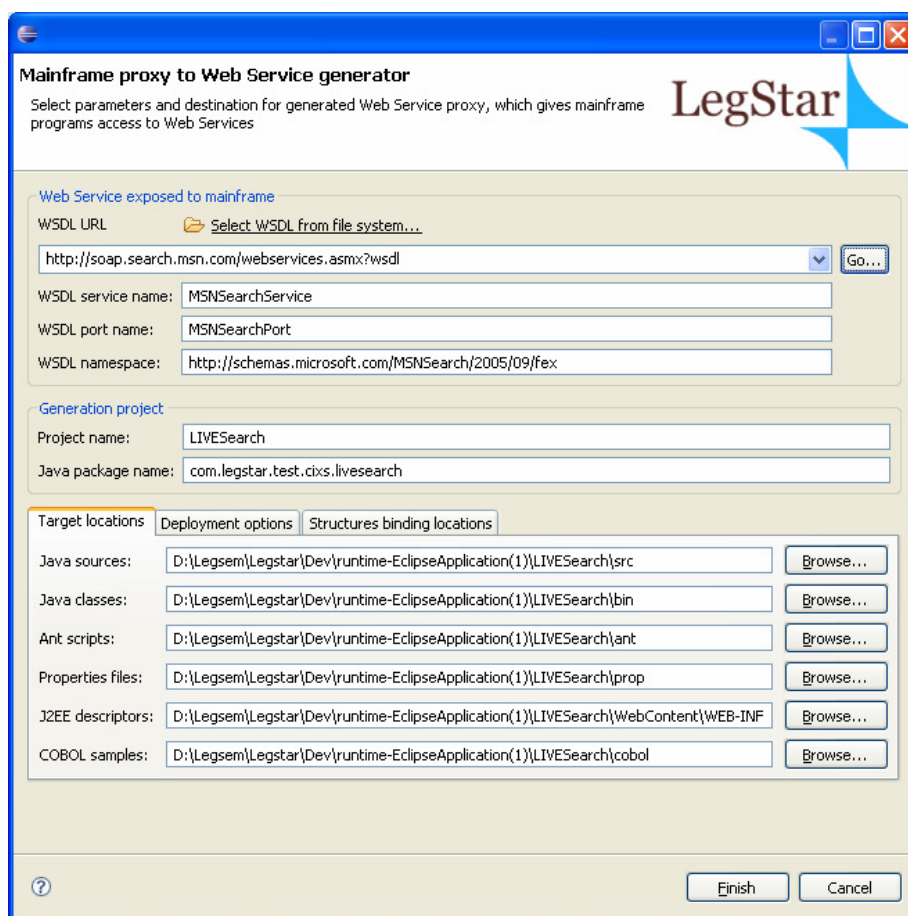
The last step is also similar to the previous use case. We start by creating a mapping to the Web Service operation we intend to consume, (this step is not shown here) and then add an operation with the following characteristics:



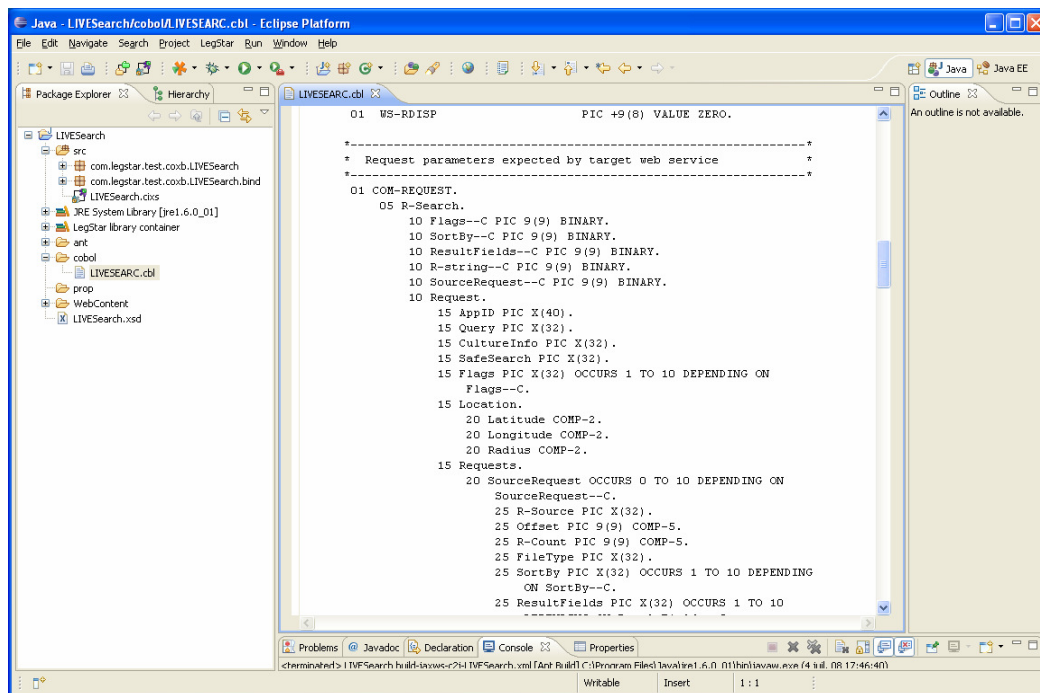
When you click on generate on the operations mapping editor, you can now select the mainframe proxy option:



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:



Clicking the finish button will create the artifacts for a Servlet proxy as well as a sample COBOL program. This sample is not complete but following the TODO comments, you should be able to figure out how to set input values or get results back:



You can deploy the Proxy Servlet to a J2EE server such as Tomcat or Jetty using the generated build.xml ant script.

V. Wrap up

LegStar comes at a time where SOA and Web Services are maturing rapidly thanks to their wide adoption.

Sun, Apache foundation and Eclipse have fed the market with several, high quality, free, SOA frameworks and standards that keep evolving with the support of very large communities.

At the same time, the nature of legacy integration is changing. J2EE and .Net applications, which were only peripheral to the legacy applications, are now becoming more and more core to the IT infrastructure. This has triggered new integration needs where legacy applications now act as clients to J2EE applications for instance.

Open-source and the changing role of legacy applications have modified the landscape for integration solutions. We believe LegStar has the right architecture to meet the challenges of the future:

- By leveraging open-source and standards, LegStar can focus on bridging legacy applications with SOA as well as the next generation of architectures that will inevitably follow.
- With a high level of modularity, a large number of use cases can benefit from LegStar features.
- By being open-source itself, LegStar can attract Mainframe developers with a Java inclination as well as Java developers curious about legacy applications.

We also hope LegStar can inspire a standardization process among integration vendors, COBOL to XML binding does not have to be proprietary anymore than Java to XML binding. JAXB has become a standard through the Java Community Process, hopefully a similar process can be organized in the legacy integration world.