Integration with IBM SPSS Collaboration and Deployment Services - Scoring Service

A guide to accessing the scoring service

John C. Hunkins (jhunkins@us.ibm.com), Software Engineer, IBM
27 February 2015

Note: This article assumes some familiarity with IBM® SPSS® Collaboration and Deployment Services (C&DS). Please review the knowledge center documentation for clarification on any unfamiliar terms or concepts.

Introduction

IBM SPSS Collaboration and Deployment Services (C&DS) is a Java Enterprise Edition (JEE) based application which exposes a number of different public and private web services, including the Scoring Service. The purpose of the Scoring Service is to allow users to obtain a score from a predictive model in real time. A score represents a computed predictive value based on data provided to the model. In order to provide a score, the scoring service has to load a predictive model file from the C&DS content repository and provide it to a Score Provider which knows how to read the model file and execute the processing instructions that are contained within the model. There are a handful of different Scoring Providers available for installation in C&DS, such as:

- SmartScore
  - Handles Predictive Model Markup Language (PMML) models that are created by IBM® SPSS® Statistics
- Modeler
  - Handles IBM® SPSS® Modeler streams and scenario files.
  - Handles certain Modeler extensions such as Entity Analytics and Text Analytics
- Analytical Decision Management
  - Handles a special form of Modeler stream that is created as a part of an Analytical Decision Management project

In order to use a predictive model with the Scoring Service, the user must configure the model by creating a Scoring Configuration, which consists of a variable number of required/optional settings. There can be any number of Scoring Configurations associated with a particular model file. All Scoring Configurations require a user-specified unique name which is used as an identifier. This unique name will be used in various web service calls to target a specific Scoring Configuration. Once the model is configured, the Scoring Service validates that the configuration is valid by starting the configuration. If the Scoring Configuration is properly configured, it will be placed into a running state; otherwise it will be placed into an error or warning state depending on the severity of the issue.

Assuming that the Scoring Configuration is in a running state, the Scoring Configuration can accept score requests. A score request contains all of the required data in the appropriate format as defined by the Scoring Configuration. The customer/developer is responsible for creating a score request programmatically by invoking a scoring service application programming interface (API). Once the scoring service has the input data, it is made available to the Score Provider which in turn feeds the input data into the model and computes a score. The Score Provider then returns the computed score to the scoring service, which in turn delivers the score to the caller in the form of a score result.

Users interact with the Scoring Service indirectly using a tool called IBM® SPSS® Collaboration and Deployment Services Deployment Manager (DM). DM is a Graphical User Interface (GUI) which simplifies the administration tasks associated with managing Scoring Configurations, and makes web service calls to C&DS on behalf of the user. While it is possible for a developer to make web service calls to manage the Scoring Configuration lifecycle, it is not necessary or recommended due to the complexity involved with that API. Instead, the focus for developers is mainly on the portion of the Scoring Service API that deals with discovery of Scoring Configuration metadata (i.e. information about the Scoring Configuration) and scoring execution.

As will be demonstrated, the Scoring Service is developer centric because in order to get any value out of the feature, a developer must use some form of programmatic access. The Scoring Service provides access via Simple Object Access Protocol (SOAP) over Hypertext Transfer Protocol (HTTP), SOAP over Java Message Service (JMS) and Representational State Transfer (REST) (i.e. JavaScript Object Notation (JSON) over HTTP). There are other means of interaction with the scoring service (e.g. using the Scoring Service JavaServer Pages (JSP) tag library for user interface widgets, and Java Management Extensions (JMX) Mbeans for obtaining scoring performance metrics), but this article does not cover those topics.

The purpose of this article is to showcase the variety of techniques and technologies that can be used to access the scoring service programmatically. By providing a consistent set of examples in a variety of programming languages, a developer can choose the development path that suits their skills. Developers can also compare the examples to see how the same task is accomplished in using an unfamiliar technology.

Scoring Service API

The first area of focus is on the Scoring Service API that is available. Here is a high level overview of the API (omitting the parameters and return types for clarity):

- Scoring Configuration Lifecycle Management API calls
  - buildConfigurationDetails
    - This call is used to initially create a Scoring Configuration, where the results of this call contain defaults for many of the settings
  - updateConfigurationDetails
    - This call takes the configuration details provided and returns a potentially updated configuration.
    - This call is made as many times as necessary to update the settings
  - setConfigurationDetails
    - This call commits the configuration details as-is to the underlying persistence mechanism
  - getConfigurationDetails
    - Provides configuration details for a previously committed Scoring Configuration
  - removeConfiguration
    - Removes a previously committed Scoring Configuration
  - changeConfigurationRunningState
    - Changes the running state for a previously committed Scoring Configuration
    - It can be useful to suspend a Scoring Configuration so it no longer consumes server resources
- Scoring Configuration Metadata API calls
  - getConfigurations
    - Returns a list of the committed Scoring Configurations, their status, running state and cache size
  - getMetadata
    - Returns information about a particular Scoring Configuration, such as the inputs that are needed by the model and outputs that the model can generate
  - getMetricItems
    - Returns a list of performance metric identifiers and other data that is computed by the Scoring Service
  - getMetricValue
    - Returns a value for a particular metric item
- Scoring Execution API calls
  - getScore
    - Takes in the data needed by the scoring model and returns the score result that was computed by the model

- Miscellaneous API calls
    - getVersion
        - Returns the version number for the Scoring Service
    - getServiceDetails
        - Returns details about the Scoring Service itself

The article will focus on the most important calls for end users, which consist of getConfigurations, getMetadata, and getScore. The example code provided shows how to invoke each of these API calls. The article will only go into details about the getScore calls shown in the code example section, since that is the primary focus for end users. The other API calls can be explored independently.

In order to make an API call, the code must send/receive data (i.e. the payload) over some form of transport. The data sent is either a SOAP envelope, which is a particular format of Extensible Markup Language (XML) or JSON data. The transport used to send the data over is either HTTP or JMS. The scoring service is limited to handling the following combinations:

- SOAP over HTTP
- SOAP over JMS
- REST (i.e. JSON over HTTP)

A developer must create the payload to send to the scoring service and receive a response payload (and handle the resulting data as needed). As long as the incoming payload is in the correct format, the scoring service will respond appropriately.

Most programming languages have a means for handling XML and JSON data (or at least there is a third party library that exists to do so). For the examples provided in this article, a particular data binding technology is used to represent the data in object form, and convert to/from XML or JSON data as needed. The choices made for these examples are not necessarily the only options available.

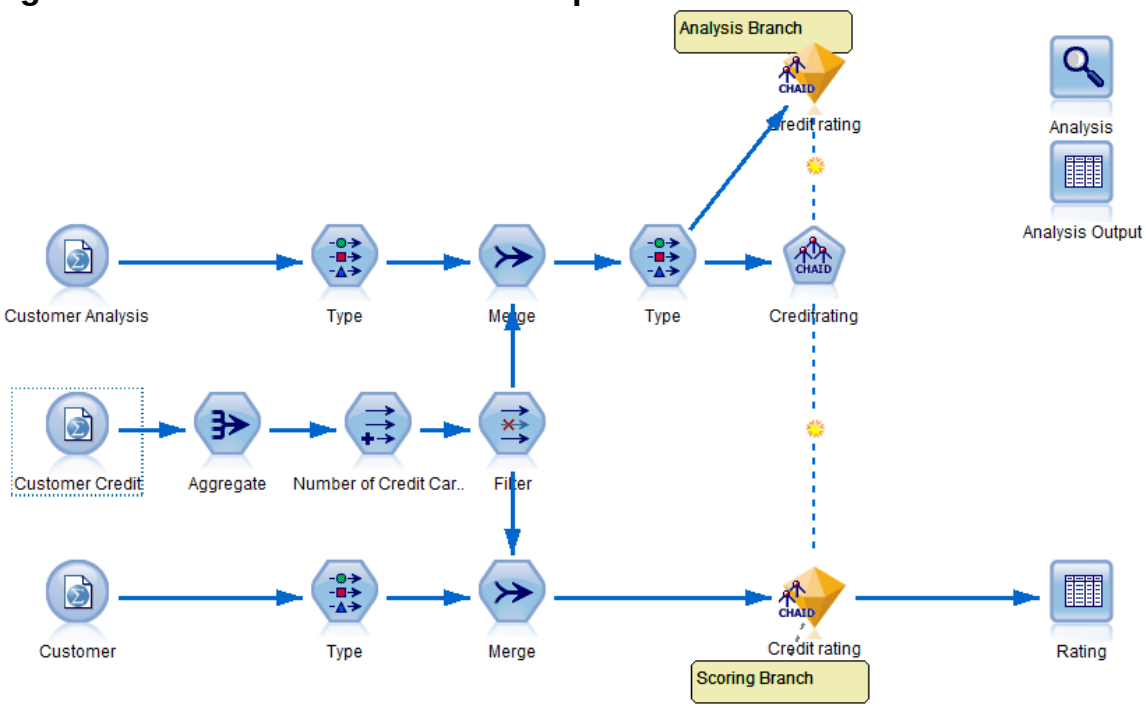
C&DS defines the Scoring Service API using Web Services Description Language (WSDL) and XML Schema. The WSDL may be viewed using a browser with the URL `http://<your_host_name>:<your_port>/scoring/services/Scoring.HttpV2?wsdl`. Developers are not required to be familiar with WSDL or XML Schema in order to work with the examples. The examples use one tool or another (depending on the programming language) to convert the WSDL/XML Schema into a form that is easy for a developer to use.

Since the scoring service API is defined in terms of WSDL and XML Schema, the JSON representation was designed to closely follow the XML format, which should make it easier for a developer to understand.

Example Model

The scoring model used in the code examples section is an IBM SPSS Modeler stream called "ExampleCredit1.str" that has two branches; one branch is used for analysis, and the other branch is used for real time scoring. Figure 1 shows a representation of the two branches.

Figure 1. The Modeler stream ExampleCredit1.str



The analysis branch is used to train the model with historical data, and the scoring branch is used with "live" (i.e. real time) data, where the past model results will ideally predict the credit worthiness of the current customer (*Note*: This model is just used for illustrative purposes, and does not represent actual credit worthiness.) To make it easier to create a scoring configuration using this model, the "Analysis Output" node was disconnected from the analysis branch so only the scoring branch will be visible in the Scoring Configuration wizard.

The scoring branch in this model contains two input nodes, one input node called "Customer" that represents customer demographic data, such as age, income level, education, number of car loans and a unique ID and another called "Customer Credit" which represents a separate source of information about a customer's credit card limit. A given customer may have zero or more credit cards, each with their own credit limit. The data from these two tables are pre-processed and then joined on the ID value using a merge node and then fed into the "model nugget" which provides the predictive capabilities.

If the Modeler client is used to execute the scoring branch, it would use inputs from a SAV file for the Customer and Customer Credit nodes, and display the calculated results

(a credit score rating) inside a table within the Modeler client. When using the scoring service, those inputs would instead be provided as a part of the web service request and the credit score rating would be returned as a web service response.

Payload Examples

As already stated, in most cases developers don't need to deal directly with XML or JSON, but it helps to see examples of what it looks like nonetheless. The example model in Example 1 has a SOAP envelope that contains a getConfigurations web service request. SOAP envelopes can contain headers which provide message related data that might be needed, and is separated from the body of the envelope. In this case there is a Web Service Security (WSSE) header that defines security related data (i.e. username and password in this case). It also contains an optional language header to indicate language preferences. All of the web service API calls that are covered in this article require the use of a security header. Note that the body of the SOAP envelope contains a single element: `<rem:getConfigurations/>`. The scoring service uses document/literal wrapped style so the name of the call is included. The call does not take any parameters, so the body of the wrapper element is empty.

Example 1. getConfigurations web service SOAP request

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:rem="http://xml.spss.com/scoring-v2/remote">
 <soapenv:Header>
  <wsse:Security soapenv:actor=http://schemas.xmlsoap.org/soap/actor/next
soapenv:mustUnderstand="0" xmlns:wsse=
"http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
  <wsse:UsernameToken xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-utility-1.0.xsd">
   <wsse:Username>Native//admin</wsse:Username>
   <wsse:Password wsse:Type=
"http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0
#PasswordText">[{AES}KrY+KLlOYo4O6545tgGsYQ==]</wsse:Password>
   <wsse:Nonce>lGpCC+uiWwYbWThVlv2lhw==</wsse:Nonce>
   <wsu:Created>2013-07-14T22:49:03Z</wsu:Created>
   </wsse:UsernameToken>
  </wsse:Security>
  <ns1:client-accept-language soapenv:actor=http://schemas.xmlsoap.org/soap/actor/next
soapenv:mustUnderstand="0" xmlns:ns1="http://xml.spss.com/ws/headers">
en-US;q=1.0, en;q=0.8</ns1:client-accept-language>
 </soapenv:Header>
 <soapenv:Body>
  <rem:getConfigurations/>
 </soapenv:Body>
</soapenv:Envelope>
```

In Example 2, there is a SOAP envelope that contains a getConfigurations web service response. Note that the response does not contain a SOAP envelope header, and just has a SOAP body. In this case, the return parameter is a wrapper element called `getConfigurationsResponse`. As shown, the contents of the response consists of a reference to the Scoring Configuration, whose ID is "Example Credit 1", which is "active" (i.e. the configuration has not been suspended) and has a cache size set to "1". It also conveys which model is referenced in the configuration, and its status (i.e. is it successfully running or not).

Example 2. getConfigurations web service SOAP response

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
 <soapenv:Body>
  <c:getConfigurationsResponse xmlns:a="http://xml.spss.com/pev/types"
xmlns:b="http://xml.spss.com/scoring/exception"
xmlns:c="http://xml.spss.com/scoring-v2/remote"
xmlns:d="http://xml.spss.com/scoring/remote"
xmlns:e="http://xml.spss.com/data"
xmlns:f="http://xml.spss.com/scoring"
xmlns="http://xml.spss.com/scoring-v2">
   <configurationReference cfgSerial="7f00000183efa35900000137b90a66f98141"
state="ACTIVE" cacheSize="1" id="Example Credit 1">
    <modelReference label="LATEST" resourcePath="/Scoring Examples/ExampleCredit1.str"
 id="7f00000183efa35900000137b90a66f98128"/>
    <configurationStatus statusCode="INFORMATION" message="Started"/>
   </configurationReference>
  </c:getConfigurationsResponse>
 </soapenv:Body>
</soapenv:Envelope>
```

In order to reduce the complexity of the remaining XML examples, only the body contents of the SOAP envelope will be shown. In Example 3, the `getMetadata` wrapper element represents the name of the web service call. This call takes a single parameter, the identifier for a Scoring Configuration.

Example 3. getMetadata web service SOAP request (body contents only)

```
<rem:getMetadata>
 <scor:id>Example Credit 1</scor:id>
</rem:getMetadata>
```

In Example 4, the response from the getMetadata call is shown (again only the SOAP body contents are shown). The XML data consists of a `metadataResult` element that can contain zero or more `metadataInputTable` elements, and each `metadataInputTable` can contain zero or more `metadataInputField` elements. The `metadataResult` can also contain zero or more `metadataOutputField` elements.

The purpose for the metadata is to describe the structure and datatypes of the data going into and out of the model. This call is used to understand what values a model requires during a getScore call, if a given value is required or not, and what outputs are configured to be returned from the model.

Example 4. getMetadata web service SOAP response (body contents only)

```
<c:getMetadataResponse
xmlns:a="http://xml.spss.com/pev/types"
xmlns:b="http://xml.spss.com/scoring/exception"
xmlns:c="http://xml.spss.com/scoring-v2/remote"
xmlns:d="http://xml.spss.com/scoring/remote"
xmlns:e="http://xml.spss.com/data"
xmlns:f="http://xml.spss.com/scoring"
```

```
xmlns="http://xml.spss.com/scoring-v2">
 <metadataResult>
  <metadataInputTable id="id84RPU94HPUM" name="Customer">
   <metadataInputField isRequired="true" name="ID" type="double" description="ID"/>
   <metadataInputField isRequired="true" name="Age" type="double" description="Age"/>
   <metadataInputField isRequired="true" name="Income level" type="string"
description="Income level"/>
   <metadataInputField isRequired="true" name="Education" type="string"
description="Education"/>
   <metadataInputField isRequired="true" name="Car loans" type="string"
description="Car loans"/>
  </metadataInputTable>
  <metadataInputTable id="id3NYQ3ZBWWX9" name="Customer Credit">
   <metadataInputField isRequired="true" name="ID" type="double" description="ID"/>
  </metadataInputTable>
  <metadataOutputField isReturned="true" name="ID" type="double" description="ID"/>
  <metadataOutputField isReturned="true" name="Number of Credit Cards" type="string"
description="Number of Credit Cards"/>
  <metadataOutputField isReturned="true" name="Age" type="double" description="Age"/>
  <metadataOutputField isReturned="true" name="Income level" type="string"
description="Income level"/>
  <metadataOutputField isReturned="true" name="Education" type="string"
description="Education"/>
  <metadataOutputField isReturned="true" name="Car loans" type="string"
description="Car loans"/>
  <metadataOutputField isReturned="true" name="$R-Credit rating" type="string"
description="$R-Credit rating"/>
  <metadataOutputField isReturned="true" name="$RC-Credit rating" type="double"
description="$RC-Credit rating"/>
 </metadataResult>
</c:getMetadataResponse>
```

Note in Example 4 that there are two input tables “Customer” and “Customer Credit” as described in the previous section. The Scoring Service expects that inputs are provided in “table format”. The code examples in this article use the following data as inputs as seen in Table 1 (a single row of data) and Table 2 (six rows of data). The data in Table 2 requires further explanation. As shown in the Modeler stream, the Customer Credit node will take the customer ID, as well as a value that represents the credit card limit. When this model is used by the scoring service, the score provider determined that the credit limit value was not used within model, so it was not listed as a required input in the resulting metadata call. Therefore, only the ID is required for the Customer Credit table, which is used to compute an aggregate value for the total number of credit cards for a given ID.

Table 1. Customer input table used in the examples

Age	Income level	Education	Car loans	ID
36	HIGH	College	2 or less	1

Table 2. Customer Credit input table used in the examples

ID
1
1
1
1
1
1

Finally, these tables of data are delivered as XML data in a getScore web service call. In Example 5, there is a score request (SOAP body contents only), starting with the `getScore` wrapper element. The `getScore` wrapper element contains a `ScoreRequest` element with its id set to the configuration name, which ensures that the scoring configuration called "Example Credit 1" receives the score request. A ScoreRequest element can contain zero or more `RequestInputTable`s. A table can contain zero or more `RequestInputRows`, which can ultimately have zero or more `Inputs` (i.e. a column and its associated value). It should be noted that it is possible to eliminate the name attribute for an input and rely on the "input ordering" that is defined in the Scoring Configuration. This allows for slightly optimized performance during XML parsing when speed of the scoring service is essential. This behavior is entirely optional and is not used in the example for purposes of clarity. Also note that each value attribute was given an explicit entry. To express an "empty string", the value could be provided as two quotes without any data in between the quotes (e.g. ""). If desired, it is also possible to provide a "null" value by omitting the value attribute altogether.

Example 5. getScore web service SOAP request (body contents only)

```
<rem:getScore>
 <scor:scoreRequest id="Example Credit 1">
  <scor:requestInputTable name="Customer">
    <scor:requestInputRow>
     <scor:input name="Age" value="36"/>
     <scor:input name="Income level" value="HIGH"/>
     <scor:input name="Education" value="College"/>
     <scor:input name="Car loans" value="2 or less"/>
     <scor:input name="ID" value="1"/>
   </scor:requestInputRow>
  </scor:requestInputTable>
   <scor:requestInputTable name="Customer Credit">
    <scor:requestInputRow>
     <scor:input name="ID" value="1"/>
    </scor:requestInputRow>
    <scor:requestInputRow>
     <scor:input name="ID" value="1"/>
    </scor:requestInputRow>
    <scor:requestInputRow>
     <scor:input name="ID" value="1"/>
    </scor:requestInputRow>
    <scor:requestInputRow>
     <scor:input name="ID" value="1"/>
    </scor:requestInputRow>
    <scor:requestInputRow>
     <scor:input name="ID" value="1"/>
    </scor:requestInputRow>
    <scor:requestInputRow>
     <scor:input name="ID" value="1"/>
    </scor:requestInputRow>
   </scor:requestInputTable>
 </scor:scoreRequest>
</rem:getScore>
```

In Example 6, an example of the SOAP body contents for a score result is provided. The scoring service only provides a single “table” of output data, so there is no need for a table element. The output is optimized to only show the “output columns” once and then zero or more rows after that. The ordering of the column names matches the output values for each output row. Note that this example shows a “null” value for the columns named “$R-Credit rating” and “RC-Credit rating”.

Example 6. getScore web service SOAP response (body contents only)

```
<c:getScoreResponse
xmlns:a="http://xml.spss.com/pev/types"
xmlns:b="http://xml.spss.com/scoring/exception"
xmlns:c="http://xml.spss.com/scoring-v2/remote"
xmlns:d="http://xml.spss.com/scoring/remote"
xmlns:e="http://xml.spss.com/data"
xmlns:f="http://xml.spss.com/scoring"
xmlns="http://xml.spss.com/scoring-v2">
 <scoreResult id="37a260a0-97e7-4999-8990-01d856571129">
  <columnNames>
   <name>ID</name>
   <name>Number of Credit Cards</name>
   <name>Age</name>
   <name>Income level</name>
   <name>Education</name>
   <name>Car loans</name>
   <name>$R-Credit rating</name>
   <name>$RC-Credit rating</name>
  </columnNames>
  <rowValues>
   <value value="1.0"/>
   <value value="5 or more"/>
   <value value="36.0"/>
   <value value="HIGH"/>
   <value value="College"/>
   <value value="2 or less"/>
   <value/>
   <value/>
  </rowValues>
 </scoreResult>
</c:getScoreResponse>
```

Code Examples

In order to illustrate the scoring service, a variety of examples are provided, each following the same overall structure, with some minor variation depending on the example. All of the examples are command line applications with the exception of the HTML example which requires a browser to execute. The command line applications are executed with a menu system and dumps out the results as text. The menu system looks like this:

```
1) Run Get Configurations Demo
2) Run Get Metadata Demo
3) Run Get Score Demo
Enter your choice (1-3):
```

The HTML example has buttons that represent the same menu options as the command line applications, and the results are displayed as text in the browser window.
The examples are designed with a common structure in order to make them consistent.

Each example has some form of:

- setup
  - any work that might be needed to setup the client
- shutdown
  - any work that might be needed to tear down the client
- execute
  - handles the "menu" where the user gets to choose the demo to run
- createScoreRequest
  - the work needed to create a score request (i.e. the outgoing payload in a getScore call)
- getMetadata / printMetadata
  - executes a getMetadata web service call and prints the results
- getConfigurations / printConfigurations
  - executes a getConfigurations web service call and prints the results
- getScore / printScore
  - executes a getScore web service call and prints the results

Preparation Tasks for Example Execution

Before executing the examples, add the Modeler file called ExampleCredit1.str to the C&DS content repository. Once the file is in the content repository, right click on the model and select "Configure Scoring…". If the "Configure Scoring…" option is not enabled, this indicates the Modeler Score Provider is not installed in C&DS.

After selecting "Configure Scoring…", the scoring wizard will launch and will display a panel to enter the name of the scoring configuration as "Example Credit 1" without the quotes. All of the examples are hard coded to use this configuration name. Failing to get the Scoring Configuration name right will cause the examples to fail to function properly. The defaults will be used for this Scoring Configuration. Click the "Finish" button, which should result in a configuration whose status is "Started". If the configuration does not have the "Started" status, the examples will fail to execute as expected. In order to see the status of your Scoring Configuration, make sure the Scoring view is visible in Deployment Manger by choosing `View->Show View->Scoring` and then select the appropriate server definition in the Server pop-up menu.

Each of the examples will require some modification in order to run them. For example, in the simplest case, it will be necessary to indicate the C&DS server host and port as well as the C&DS credentials for authentication. Other examples may require more modification beyond this, and these will be called out explicitly in the article. Be sure to refer to the ReadMe.txt files found in the examples for specific instructions.

Note that these examples do not provide details regarding error checking and reporting, language specific issues (e.g. Unicode handling), or deeper topics such as transport/protocol security. These details go beyond the scope of this article.

Java Examples

All of the Java examples are included in a single Rational Application Developer (RAD) Integrated Development Environment (IDE) project, which can be imported into an existing workspace. It is also possible to use the Eclipse IDE. The RAD/Eclipse IDE is not required, but it makes it easier to develop and run the examples. The Java examples have been tested with (and require) the following libraries, as shown in Table 3.

Table 3. Required Java libraries

commons-codec.jar	1.3	http://commons.apache.org/
commons-httpclient-3.0.1.jar	3.0.1	http://commons.apache.org/
commons-lang-2.4.jar	2.4	http://commons.apache.org/
commons-logging-api.jar	1.0.4	http://commons.apache.org/
commons-logging.jar	1.0.4	http://commons.apache.org/
jackson-all-1.9.7.jar	1.9.7	http://jackson.codehaus.org/1.9.7/jackson-all-1.9.7.jar
JSON4J.jar	1.0.1	Available in C&DS EAR/lib
com.ibm.ws.ejb.thinclient_8.0.0.jar	WebSphere version 8	Available in <WebSphere install>/runtimes
com.ibm.ws.orb_8.0.0.jar	WebSphere version 8	Available in <WebSphere install>/runtimes
com.ibm.ws.sib.client.thin.jms_8.0.0.jar	WebSphere version 8	Available in <WebSphere install>/runtimes
Java Software Development Kit	1.6	Sun or IBM JDK

Newer versions of these libraries will likely work, but have not been tested. These libraries should be placed into the `C:\ScoringClientExamples\Java\Example Project\lib` directory.

Be sure to read `C:\ScoringClientExamples\Java\Example Project\ReadMe.txt` for details about the example packaging and class descriptions. Note that the Java examples have common code in the class `com.ibm.spss.example.ExampleBase`, and each example extends from that class.

Java (SOAP over HTTP)

The Java SOAP over HTTP example can be found in `com.ibm.spss.example.soap.SimpleJAXWSScoringExample`. This class contains the `main` method, and can be used to execute the example.

The Java SDK provides a SOAP web service implementation called Java API for XML Web Services (JAX-WS). This technology allows developers to send and receive SOAP envelopes over HTTP. JAX-WS leverages another Java technology called Java Architecture for XML Binding (JAXB) to assist with converting Java classes to/from XML.

The Java SDK comes with the `wsimport` tool which allows developers to automatically generate Java web service client classes from WSDL. The `wsimport` tool minimally requires a single WSDL_URI parameter to tell it where to find the WSDL like this:

```
"c:\Program Files\IBM\Java60\bin\wsimport.exe"
http://localhost:7001/scoring/services/Scoring.HttpV2?wsdl
```

The default output location for the wsimport is the current directory, but this is configurable by adding additional parameters. Run the wsimport tool without parameters to see the options the tool supports. By default the tool will generate code into Java packages using the target namespace of the WSDL and XSD files.

The package name is calculated by using the reverse internet domain name followed by the path component of the URL (e.g. targetNamespace="http://xml.spss.com/scoring/wsdl" would become com.spss.xml.scoring.wsdl). The default package definition can be customized.

Note that the WSDL/XSD files represent an XML contract that the client and server must adhere to, so it is not possible change the namespaces used in the WSDL/XSD. However, the default package naming convention can be overridden and a custom package structure can be used via jaxws:bindings elements for WSDL files and jaxb:schemaBindings elements for the XSD files. Be aware that C&DS provides its own package overrides, and that these overrides only apply to JAX-WS/JAXB code. Any other web service client technology will ignore these overrides.

One requirement for JAX-WS is that the WSDL must be available to the JAX-WS client/server at runtime. By default JAX-WS will obtain the WSDL from the same location that was specified by the WSDL_URI parameter provided to wsimport. It is also possible to have JAX-WS load the WSDL locally, which can be more efficient. Typically, to load the WSDL locally, the generated code and the WSDL/XSD files would be included in a JAR with the WSDL/XSD files located under the `\META-INF\wsdl` directory.

The generated code creates a web services class called `com.spss.scoring.ws.jaxws.ScoringServices` which is used to invoke the web service endpoint. The object can be instantiated using the default constructor, but this uses the defaults entered when the code was generated. Instead, by using the other constructor which takes a java.net.URL and javax.xml.namespace.QName, the WSDL can be specified either locally or via the server.

For example, to get the WSDL from a server, use:

```
com.spss.scoring.ws.jaxws.ScoringServices service =
  new com.spss.scoring.ws.jaxws.ScoringServices(
    new
URL("http://localhost:7001/scoring/services/Scoring.HttpV2?wsdl"),
    new QName("http://xml.spss.com/scoring/wsdl", "ScoringServices"));
```

If the WSDL/XSD is stored locally, use this option instead:

```
com.spss.scoring.ws.jaxws.ScoringServices service =
  new com.spss.scoring.ws.jaxws.ScoringServices(
    DemoClass.class.getResource("/META-INF/wsdl/scoring.wsdl"),
    new QName("http://xml.spss.com/scoring/wsdl", "ScoringServices"));
```

In the local WSDL case, the URL is specified as a class resource lookup, which will work as long as the client code is packaged in a JAR. The classloader mechanism will search for the WSDL at the root of the JAR under the path `/META-INF/wsdl/scoring.wsdl`. The example provided at `com.ibm.spss.example.soap.SimpleJAXWSScoringExample.setup()` uses the simple constructor because the generated client code was manually modified to use the local WSDL.

The `setup()` method also enables the security mechanism required by C&DS. This is accomplished by using the `ScoringServices` object, which allows developers to supply an object that implements the JAX-WS interface called `javax.xml.ws.handler.soap.SOAPHandler`. JAX-WS allows code to be inserted into the request/response cycle so that developers can read and even manipulate the SOAP message on the client and the server. As Example 7 shows, the client and server can intervene in the request/response cycle anywhere an X is shown. Logical handlers only have access to the body of the SOAP message, but SOAP handlers have access to the entire SOAP envelope.

Example 7. Diagram showing request/response flow where handlers can intervene (signified by X)

```
                  logical     SOAP             SOAP      logical
                  handler    handler          handler    handler
|------|request  ->   X    ->    X    |===| ->    X    ->    X   -> request |-------|
|Client|                                                                   |service|
|------|response <-   X    <-    X    |===| <-    X    <-    X   <- response|-------|
```

From the perspective of the Java examples provided in this article, the C&DS security headers need to be applied in the client outbound request using a SOAP handler. The most important part of the `SOAPHandler` interface is the `public boolean handleMessage(SOAPMessageContext context)` method. Inside this method the `SOAPMessageContext` object can be used to discover if a message is incoming or outgoing and apply the SOAP header as shown in Example 8.

Example 8. An example SOAPHandler.handleMessage(SOAPMessageContext context) implementation

```
// Apply this handler to only outbound traffic
if((Boolean)context.get(SOAPMessageContext.MESSAGE_OUTBOUND_PROPERTY))
{
 // get the message
 SOAPMessage message = context.getMessage();
 try
 {
  // get the message header
  SOAPEnvelope envelope = message.getSOAPPart().getEnvelope();
  SOAPHeader header = envelope.getHeader();
  if (header == null)
  {
   header = envelope.addHeader();
  }

  // add the UsernameToken header
  header.addChildElement(createUsernameTokenSecurityHeader());
  // assuming the language was provided, apply the custom language
header
  if(i_acceptLanguage != null)
  {
   header.addChildElement(createLanguageHeader());
  }
 }
 catch (Exception e)
 {
  e.printStackTrace();
 }
}
// allow any other handler to execute
return true;
```

Refer to `com.ibm.spss.example.soap.SecurityHandler` for the full example. **IMPORTANT:** Be sure to apply a handler prior to obtaining the client web service proxy. If this is not done, the handler chain will not be invoked. See `com.ibm.spss.example.soap.SimpleJAXWSScoringExample.setup()` for an example of how to do this in the proper order.

Next, the URL for the C&DS server should be provided, but this has to be done on the web service client proxy object. All that needs to be done here is use the ScoringServices object to obtain the web service client proxy, get the request context object (which is essentially a `Map)`, and place the URL as a String into the map as shown in Example 9. When running the example, be sure to include the correct server host and port.

Example 9. Set the C&DS URL on the client web service proxy

```
// set the URL for the server
ScoringV2 httpV2 = service.getHttpV2();
((BindingProvider)httpV2).getRequestContext()
  .put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, URL);
```

In order to create a score request, the generated code created by wsimport is used. The generated code consists of simple Java objects that are "assembled" and provided the necessary request inputs. The score request creation can be seen in Example 10, and the full example can be found in `com.ibm.spss.example.ExampleBase`.

Example 10. Create a ScoreRequest object and fill it with input data

```
// Create the score request object
ScoreRequest scoreRequest = new ScoreRequest();
// First put the configuration ID into it:
scoreRequest.setId(configID);

/** Request Input Table for "Customer" */

// Create storage for inputs, rows
RequestInputRow requestInputRows1 = new RequestInputRow();

// Create a new request input table with a given name and the rows it will contain
RequestInputTable requestInputTable1 = new RequestInputTable();
requestInputTable1.setName("Customer");

// Add the table to our list of input tables
scoreRequest.getRequestInputTable().add(requestInputTable1);

// Add the request inputs as a new row
requestInputTable1.getRequestInputRow().add(requestInputRows1);

// Add each input to the request
requestInputRows1.getInput().add(createInput("Age", "36"));
requestInputRows1.getInput().add(createInput("Income level", "HIGH"));
requestInputRows1.getInput().add(createInput("Education", "College"));
requestInputRows1.getInput().add(createInput("Car loans", "2 or less"));
requestInputRows1.getInput().add(createInput("ID", "1"));

/** Request Input Table for "Customer Credit" */

// Create storage for inputs, rows
RequestInputRow requestInputRows2 = new RequestInputRow();

// Create a new request input table with a given name and the rows it will contain
RequestInputTable requestInputTable2 = new RequestInputTable();
requestInputTable2.setName("Customer Credit");

// Add the table to our list of input tables
scoreRequest.getRequestInputTable().add(requestInputTable2);

// Add the request inputs as a new row (repeat the same row of inputs multiple times)
requestInputTable2.getRequestInputRow().add(requestInputRows2);
requestInputTable2.getRequestInputRow().add(requestInputRows2);
requestInputTable2.getRequestInputRow().add(requestInputRows2);
requestInputTable2.getRequestInputRow().add(requestInputRows2);
requestInputTable2.getRequestInputRow().add(requestInputRows2);
requestInputTable2.getRequestInputRow().add(requestInputRows2);

// Add each input to the request
requestInputRows2.getInput().add(createInput("ID", "1"));
```

One thing to note about JAXB objects is the use of Java collections to represent portions of the XML that allow zero or more elements. For example, a `ScoreRequest` object contains zero or more `RequestInputTable` elements, so the JAXB object that represents a `ScoreRequest` contains a Java collection of type `List<RequestInputTable>`. In order to add `RequestInputTable` objects to the `ScoreRequest`, call `getRequestInputTable()`, and then add a `RequestInputTable` to the returned collection.

Once the `ScoreRequest` object is populated, it is a simple matter to use the client web service proxy (i.e. a proxy object that implements the `ScoringV2` interface) to invoke the scoring service as seen in Example 11.

Example 11. Invoke the scoring service getScore call

```
ScoringV2 scoring = // the client web service proxy that was populated earlier
ScoreResult getScoreResponse = scoring.getScore(scoreRequest);
```

Java (REST – JSON over HTTP using JAXB)

This example uses the JAXB classes that were generated for the JAX-WS example, and uses the Jackson library to convert back and forth between JAXB objects and JSON. Using a library like Jackson makes JSON parsing easy because the data ends up inside JAXB objects which are much easier to understand and manipulate, because the Java objects directly correspond to XML schema elements. Other JSON libraries like JSON4J are more difficult to use because the data structures are a hierarchy of JSON Array and JSON Object types. The next example illustrates the use of JSON4J.

The REST JAXB example can be found in `com.ibm.spss.example.rest.RestJaxbExample`, which contains the main method.

Since the HTTP calls are being made from our Java client, the Apache Commons HttpClient library is used to simplify direct HTTP programming. The setup method `com.ibm.spss.example.rest.RestJaxbExample.setup()` simply creates an HttpClient object, allows it to send authorization headers preemptively to avoid the normal overhead associated with HTTP authorization, and sets the C&DS credentials. Details are shown in Example 12 below.

Example 12. Setting up HttpClient in the RestJaxbExample.setup() method

```
// if the HTTP Client object has not been initialized
if(httpClient == null)
{
 httpClient = new HttpClient();
 // allow the HTTP client to preemptively send authentication headers
 httpClient.getParams().setAuthenticationPreemptive(true);

 // add the security credentials
 Credentials credentials = new UsernamePasswordCredentials("admin", "my_password");
 httpClient.getState().setCredentials(
  new AuthScope(HOST,PORT,AuthScope.ANY_REALM), credentials);
}
```

Following the REST convention, a REST web service consists of resources which correspond to a base URI combined with a relative URI. For a given resource URI, the developer invokes an HTTP call using GET, POST, PUT, or DELETE. For example, a list of all scoring configurations can be obtained via an HTTP GET by combining the base URI `http://{server}:{port}/scoring/rest` with a relative URI `configuration` which results in `http://{server}:{port}/scoring/rest/configuration`. C&DS comes with detailed documentation specifying the behavior of the REST web service API, so it won't be covered in detail here. See `<C&DS Installation>\Server\documentation\en\web_services\Scoring_Service_REST_Developers_Guide.pdf` for more information.

Whenever a web service call is made in this example it has to be done using the `com.ibm.spss.example.rest.RestJaxbExample.execute(String uri, org.apache.commons.httpclient.methods.RequestEntity requestEntity)` method. This method makes sure the URI provided is properly URL encoded, sets either GET or POST depending on the existence of the requestEntity parameter (i.e. the outgoing payload), adds necessary HTTP request headers, executes the call and returns the HTTP body content as a string. This can be seen in Example 13. When executing this code example, be sure to update the HOST and PORT variables to match the server.

Example 13. Executing an HTTP request and returning the response string

```
private String executeRequest(String uri,
 org.apache.commons.httpclient.methods.RequestEntity requestEntity)
 throws EncoderException, HttpException, IOException
{
 // encode the URI to be sure we don't use illegal characters in the URL
 URLCodec codec = new URLCodec();
 String encodedURI = codec.encode(uri);

 HttpMethod method = null;
 if(requestEntity == null)
 {
  method = new GetMethod(baseURL + encodedURI);
 }
 else
 {
  method = new PostMethod(baseURL + encodedURI);
  ((PostMethod)method).setRequestEntity(requestEntity);
 }

 // set language and JSON content type
 method.addRequestHeader("Accept-Language", "en_US");
 method.addRequestHeader("Content-Type", "application/json; charset=utf-8");

 httpClient.executeMethod(method);

 String responseBodyAsString = method.getResponseBodyAsString();

 method.releaseConnection();

 return responseBodyAsString;
}
```

Since this example shares the code to create a ScoreRequest object (see Example 10), it will not be repeated here. In order to use the JAXB ScoreRequest object Jackson is used to convert the object into JSON data in the form of a Java `String` using the `org.codehaus.jackson.map.ObjectMapper` object. The JSON string is sent to C&DS using an HTTP POST with the URL `http://{server}:{port}/scoring/rest/configuration/{configuration id}/score.` The HTTP response contains JSON data as a `String` and is converted from JSON back into JAXB objects, using a Jackson `ObjectMapper` that has been specially initialized using a `JaxbAnnotationIntrospector.` Once the data is converted to a JAXB `ScoreRequest` object, it can be printed out. This entire process can be seen in Example 14.

Example 14. Invoke the configuration/{configuration id}/score resource

```
protected void getScore() throws Exception
{
 // build the Score Request using JAXB classes
 ScoreRequest scoreRequest = createScoreRequest(configId);

 // convert the JAXB score request into JSON
 String jsonScoreReqStr = new ObjectMapper().writeValueAsString(scoreRequest);

 // execute request
 String jsonStr = executeRequest("configuration/" + configId + "/score",
  new StringRequestEntity(jsonScoreReqStr));

 // parse and display response
 ScoreResult scoreResult = unmarshalJSON(jsonStr, ScoreResult.class);
 printScoreResponse(scoreResult);
}
…
private <E> E unmarshalJSON(String jsonStr, Class<E> jaxbClass) throws
JsonParseException, JsonMappingException, IOException
{
 ObjectMapper mapper = new ObjectMapper();
 AnnotationIntrospector introspector = new JaxbAnnotationIntrospector();
 mapper.setDeserializationConfig(
  mapper.getDeserializationConfig().withAnnotationIntrospector(introspector));
 E result = mapper.readValue(jsonStr, jaxbClass);
 return result;
}
```

Java (REST – JSON over HTTP using JSON4J)

This next example has a main method located in `com.ibm.spss.example.rest.RestJsonExample`. This example is *very* similar to the previous example but instead of using JAXB objects, there is a hierarchy of "generic" `com.ibm.json.java.JSONObject` and `com.ibm.json.java.JSONArray` objects to represent the JSON data. A JSON object represents key/value pairs, and JSON array represents a sequence of values. JSON can also contain Strings, Numbers and Boolean values.

Example 15 represents a *partial* graph of objects that are expected in a score request. Note that the scoring service can accept "context" data, but this is outside the scope of the article, so only the request input table portion of the graph is shown. The Example shows a JSON Object key/value pair with the value's data type followed by a key in parentheses.

Example 15. A hierarchy of JSONObject and JSONArray objects that represent a ScoreRequest

```
JsonObject
|---> String ("id")
|---> JsonArray ("requestInputTable")
      |---> JsonObject
            |---> String ("name”)
            |---> JsonArray ("requestInputRow")
                  |---> JsonObject
                        |---> JsonArray ("input")
                              |---> JsonObject
                                    |---> String ("name")
                                    |---> String ("value")
```

The code shown in Example 16 provides an example of how to create a score request using JSONObject and JSONArray objects.

Example 16. Create a JSON score request and fill it with input data

```
JSONObject jsScoreRequest = new JSONObject();

//the id
jsScoreRequest.put("id", "Example Credit 1");

//get the list of InputTables
JSONArray jsRequestInputTables = new JSONArray();
jsScoreRequest.put("requestInputTable", jsRequestInputTables);

JSONObject jsRequestInputTable = new JSONObject();
jsRequestInputTables.add(jsRequestInputTable);

jsRequestInputTable.put("name", "Customer");

//get the list of Input Rows
JSONArray jsRequestInputRows = new JSONArray();
jsRequestInputTable.put("requestInputRow", jsRequestInputRows);

//get the list of Inputs
JSONObject jsRequestInputRow = new JSONObject();
JSONArray jsInputs = new JSONArray();
jsRequestInputRow.put("input", jsInputs);
jsRequestInputRows.add(jsRequestInputRow);

JSONObject jsInput = new JSONObject();
jsInput.put("name", "Age");
jsInput.put("value", "36");
jsInputs.add(jsInput);

JSONObject jsInput2 = new JSONObject();
jsInput2.put("name", "Income level");
jsInput2.put("value", "HIGH");
jsInputs.add(jsInput2);
```

```
JSONObject jsInput3 = new JSONObject();
jsInput3.put("name", "Education");
jsInput3.put("value", "College");
jsInputs.add(jsInput3);

JSONObject jsInput4 = new JSONObject();
jsInput4.put("name", "Car loans");
jsInput4.put("value", "2 or less");
jsInputs.add(jsInput4);

JSONObject jsInput5 = new JSONObject();
jsInput5.put("name", "ID");
jsInput5.put("value", "1");
jsInputs.add(jsInput5);


JSONObject jsRequestInputTable2 = new JSONObject();
jsRequestInputTables.add(jsRequestInputTable2);

jsRequestInputTable2.put("name", "Customer Credit");

//get the list of Input Rows
JSONArray jsRequestInputRows2 = new JSONArray();
jsRequestInputTable2.put("requestInputRow", jsRequestInputRows2);

//get the list of Inputs

JSONObject jsInput6 = new JSONObject();
jsInput6.put("name", "ID");
jsInput6.put("value", "1");

JSONObject jsRequestInputRow2 = new JSONObject();
jsRequestInputRows2.add(jsRequestInputRow2);
JSONArray jsInputs2 = new JSONArray();
jsRequestInputRow2.put("input", jsInputs2);
jsInputs2.add(jsInput6);

JSONObject jsRequestInputRow3 = new JSONObject();
jsRequestInputRows2.add(jsRequestInputRow3);
JSONArray jsInputs3 = new JSONArray();
jsRequestInputRow3.put("input", jsInputs3);
jsInputs3.add(jsInput6);

JSONObject jsRequestInputRow4 = new JSONObject();
jsRequestInputRows2.add(jsRequestInputRow4);
JSONArray jsInputs4 = new JSONArray();
jsRequestInputRow4.put("input", jsInputs4);
jsInputs4.add(jsInput6);

JSONObject jsRequestInputRow5 = new JSONObject();
jsRequestInputRows2.add(jsRequestInputRow5);
JSONArray jsInputs5 = new JSONArray();
jsRequestInputRow5.put("input", jsInputs5);
jsInputs5.add(jsInput6);

JSONObject jsRequestInputRow6 = new JSONObject();
jsRequestInputRows2.add(jsRequestInputRow6);
```

```
JSONArray jsInputs6 = new JSONArray();
jsRequestInputRow6.put("input", jsInputs6);
jsInputs6.add(jsInput6);
```

The work to setup the HTTP client and execute requests is exactly the same as the previous example, so simply refer to Example 12 and Example 13 for those examples. As shown in Example 17, the work of executing a score request is very similar to Example 14; the only difference is the use of `JSONObject`s, rather than JAXB objects. First, the `JSONObject` is converted to a `String`, and sent to C&DS using a HTTP POST using the URL `http://{server}:{port}/scoring/rest/configuration/{configuration id}/score`. The HTTP response contains JSON data as a `String` and is converted from a JSON `String` into a `JSONObject` using the JSON4J parser and then printed.

Example 17. Invoke the configuration/{configuration id}/score resource

```
protected void getScore() throws Exception
{
 // build the Score Request
 JSONObject jsonScoreRequest =
ScoreRequestJsonTransformer.toJsonObject();

 String jsonScoreReqStr = jsonScoreRequest.toString();

 // execute request
 String jsonStr = executeRequest("configuration/" + configId + "/score",
  new StringRequestEntity(jsonScoreReqStr));

 // parse and display response
 JSONObject jsonScoreResult = JSONObject.parse(jsonStr);
 printScoreResponse(jsonScoreResult);
}
```

Java (SOAP over JMS)

The SOAP over JMS example has its main method located in `com.ibm.spss.example.jms.SimpleJMSScoringExample`. Please refer to that class for the full example.

SOAP over JMS differs from SOAP over HTTP in that the transport being used for communication is JMS instead of HTTP. In the SOAP over HTTP example, the JAX-WS framework understands how to process SOAP envelopes and handle HTTP communication automatically, which makes the client usage extremely simple. The mechanism described here requires more effort on the part of the developer because of the need to open a connection to a server that hosts a particular JMS queue, interact with multiple queues, and handle outgoing and incoming JMS messages, as well as process the SOAP envelopes contained inside the JMS messages. All of these activities must be handled in the code.

It should be noted that the concept for using SOAP over JMS has existed for some time, but the formal W3C recommendation was not finalized until February 16, 2012. The SOAP over JMS implementation provided by C&DS for the scoring service predates any formal standards, and application server support for such features was proprietary to each vendor. The decision was made to provide a unique C&DS implementation which is essentially non-standard, but still fully functional.

To setup this example, the Java Naming and Directory Interface (JNDI) is used in order to lookup the resources required for JMS communication. The process for accessing a JMS engine depends entirely on the environment that is being used. This example assumes that an external client will connect to IBM WebSphere 8, which requires runtime libraries shown in Table 3. If the example is used in a different server environment, refer to the documentation for that server for the appropriate library and connection values. The WebSphere example initializes the `javax.naming.InitialContext` using provider URL `iiop://localhost:30604` and `com.ibm.websphere.naming.WsnInitialContextFactory` for the initial context factory. When running this example, be sure to change the URL to correspond to the value that your WebSphere server is using. The `InitialContext` object is used to discover a JMS queue found at JNDI name `queue/PASWScoring`, and a queue connection factory found at JNDI name `ConnectionFactory`. Both of these resources are bound to JNDI by C&DS for use by C&DS itself as well as external clients.

Using these resources, the developer can create a queue connection. This queue connection is used to create two different queue sessions, a sender session and a listener session. The sender session is used to send a JMS message to the scoring queue (i.e. an outgoing JMS message), and a listener session is used to create a temporary reply-to queue for messages coming back to the client (i.e. an incoming JMS message). A JMS message consumer is also created so that the client can consume messages coming in from the JMS reply-to queue. The entire setup code can be seen in Example 18.

Example 18. Setup the JMS connection, queues and sessions

```
protected void setup() throws Exception
{
 // JMS Step 1 - Initialize JMS
 // Create a hash table of settings required to access JMS
 // Note that these settings are application server specific
 Hashtable<String, String> hashTable = new Hashtable<String, String>();

 // Some servers require a value for java.naming.factory.url.pkgs
 // hashTable.put(javax.naming.InitialContext.URL_PKG_PREFIXES, "");
 hashTable.put(javax.naming.InitialContext.PROVIDER_URL, PROVIDER_URL);
 hashTable.put(javax.naming.InitialContext.INITIAL_CONTEXT_FACTORY,
  INITIAL_CONTEXT_FACTORY);

 // Create an InitialContext object so you can discover
 // various C&DS objects such as connection factories and queues.
 javax.naming.InitialContext context = new javax.naming.InitialContext(hashTable);

 // attempt to get the connection factory and queue
 scoringQueue = (javax.jms.Queue)context.lookup("queue/PASWScoring");
 javax.jms.QueueConnectionFactory factory =
  (javax.jms.QueueConnectionFactory)context.lookup("ConnectionFactory");

 // create a connection to the queue and start it
 queueConnection = factory.createQueueConnection();
 queueConnection.start();

 // JMS Step 2 - Open a temporary reply queue.
 listenerSession = queueConnection.createQueueSession(false,
  javax.jms.Session.AUTO_ACKNOWLEDGE);
 temporaryQueue = listenerSession.createTemporaryQueue();

 // JMS Step 3 - Listen to the temporary reply queue for messages returned by
the scoring service
 consumer = listenerSession.createConsumer(temporaryQueue);

 // JMS Step 4 - Create a QueueSender from a QueueSession
 // with this session you can send as many messages as you like...
 senderSession = queueConnection.createQueueSession(false,
  javax.jms.Session.AUTO_ACKNOWLEDGE);
 sender = senderSession.createSender(scoringQueue);
}
```

When it comes time to actually send a JMS message and receive a response JMS message, the sender session is used to create a `javax.jms.BytesMessage`, which will contain the data that will be sent to the server (i.e. an outgoing SOAP message). In order for C&DS to know where to send a response, the message is updated with a reply-to queue which points to the temporary queue. Once the outgoing message is fully prepared, it is sent using the queue sender. Immediately after the message is sent, the message consumer that is listening to the reply-to queue waits for an inbound message (i.e. it blocks until the message is received). The bytes from the reply message are read and converted back into a Java String, which represents the reply SOAP message.

While this example provides an implementation that runs synchronously (i.e. a message is sent and waits for a reply) it is possible to create an asynchronous implementation (i.e. a message is sent and the method returns immediately, and at some later point a reply is received and processed independently). When asynchronous behavior is needed, JMS provides a mechanism for matching up messages. Every JMS message can contain a unique identifier found in the JMS header called `JMSMessageID`, and also has an

optionally set `JMSCorrelationID` JMS header, which can be used to indicate a relationship between a request message and a response message. The example calls these values out but does nothing with them because the example is using synchronous behavior. Example 19 shows the full approach.

Example 19. Send/receive a JMS message

```
private String sendJMSMessage(byte[] bs) throws Exception
{
 String replyMessage = null;

 javax.jms.BytesMessage messageToSend = senderSession.createBytesMessage();

 // JMS Step 5 - Set the reply-to queue on the JMS message.
 messageToSend.setJMSReplyTo(temporaryQueue);

 // JMS Step 6 - Send the message to a JMS queue
 // populate the message with the soap envelope and send it
 messageToSend.writeBytes(bs);
 sender.send(messageToSend);

 // JMS Step 7 - receive the reply message
 // NOTE: This method blocks until a message is received
 javax.jms.Message replyJMSMessage = consumer.receive();


 // the message format should be a bytes message
 if (replyJMSMessage != null && replyJMSMessage instanceof javax.jms.BytesMessage)
 {
  javax.jms.BytesMessage bytesMessage = (javax.jms.BytesMessage) replyJMSMessage;
  byte[] bytes = new byte[(int) bytesMessage.getBodyLength()];
  bytesMessage.readBytes(bytes);

  // the reply message
  replyMessage = new String(bytes, "UTF-8");

  // The JMS correlation ID can be used to match a sent message with a response message.
  // In this case, we are running synchronously, so this value is not important, but if
  // asynchronous behavior is needed, this is required to match the outgoing and incoming
  // messages
  String jmsCorrelationID = replyJMSMessage.getJMSCorrelationID();
 }

 // *AFTER* the message is sent, get the message ID
 // You would keep the message ID around somewhere so you can match it to a reply later.
 // This is only necessary for asynchronous access (for this example it is not required)
 String messageID = messageToSend.getJMSMessageID();
 return replyMessage;
}
```

The previous examples explain how to setup, send and receive a message via JMS, but have not discussed how to create and handle the SOAP envelopes. As in the previous examples, the JAX-WS generated classes are used to simplify this process. Other methods of creating SOAP envelopes can be used as long as they comply with the XML schema. The code in Example 10 will be reused to create a JAXB `ScoreRequest` object, but in this case the JAX-WS wrapper class GetScore is also used, and the ScoreRequest object is inserted inside it, as shown in Example 20. In the SOAP over HTTP example, the JAX-WS framework normally processes the SOAP envelope and as a result handles the wrapper classes as well. In the SOAP over JMS example that functionality has to be coded separately, starting with the wrapper classes.

Example 20. Invoke the scoring getScore call using SOAP over JMS

```
protected void getScore() throws Exception
{

 // Build a score request object
 // The data represents a score request.
 ScoreRequest scoreRequest = createScoreRequest(configId);

 GetScore getScoreWrapper = new GetScore();
 getScoreWrapper.setScoreRequest(scoreRequest);

 // execute request
 GetScoreResponse response = (GetScoreResponse)executeRequest(getScoreWrapper);

 // extract the data from the response wrapper
 ScoreResult getScoreResponse = response.getScoreResult();

 // print out the results
 printScoreResponse(getScoreResponse);

}
```

Now that the JAXB wrapper class `GetScore` has been created, it needs to be inserted into a `SOAPMessage` object. To do this, the Java SOAP message and JAXB API are needed. The Java SOAP message API contains a `MessageFactory` which allows for the creation of SOAP messages. Using the JAXB API it is possible to use a JAXB Marshaller to write the JAXB wrapper object directly into the SOAP message body. The result is a fully complete `SOAPMessage` object. Note that it is not necessary to provide the SOAP security headers as normally required in SOAP over HTTP.

Customers are expected to properly secure their JMS environment using the security mechanisms provided by the server, which will prevent unauthorized access to the scoring JMS queue. Prior to sending, the `SOAPMessage` is converted to a byte array and sent using the mechanism already discussed in Example 19. At this point, the score response is still in the form of a Java String, which represents the SOAP message. The process of unmarshalling the data from a String to a JAXB object begins with the Java SOAP `MessageFactory` object and creating an empty response message, and obtaining the `SOAPPart` from the message. In other words, the XML portion of the SOAP message is obtained, as opposed to the other optional parts of a SOAP message such as a SOAP attachment (*Note:* the scoring service does not use SOAP attachments). The contents of the SOAPPart are set to the response message, and then the changes are "saved" to the response `SOAPMessage`. Once this is complete, the `SOAPMessage` now contains the response data and the data can be unmarshalled into a JAXB `ScoreResult` object, which can be used to print out the values as all of the other Java examples do. Example 21 provides the complete approach.

Example 21. Execute and process the GetScore and GetScoreResponse wrapper objects

```
private Object executeRequest(Object wrapper) throws Exception
{
 // create a message factory to create SOAP messages
 MessageFactory factory = MessageFactory.newInstance(SOAPConstants.SOAP_1_1_PROTOCOL);

 // this represents the outgoing request message
 SOAPMessage messageRequest = factory.createMessage();

 // marshal the JAXB wrapper object directly into the SOAP message body
 Marshaller marshaller = V2_JAXB_CONTEXT.createMarshaller();
 marshaller.setProperty(Marshaller.JAXB_ENCODING, "UTF-8");
 marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.FALSE);
 marshaller.marshal(wrapper, messageRequest.getSOAPBody());

 // convert the SOAP message into bytes which can be sent via JMS message
 ByteArrayOutputStream baos = new ByteArrayOutputStream();
 // by writing out the message, the contents of the SOAP message are automatically
 // saved
 messageRequest.writeTo(baos);

 // send the message and get a response
 String response = sendJMSMessage(baos.toByteArray());

 // create a string reader for the response message
 StringReader sr = new StringReader(response);

 // create a temporary soap message object to receive the raw response string
 SOAPMessage messageResponse = factory.createMessage();

 // place the raw SOAP message string into the SOAPMessage object
 SOAPPart requestSoapPart = messageResponse.getSOAPPart();
 requestSoapPart.setContent(new StreamSource(sr));

 // the "changes" must be saved after which we obtain the SOAP body
 messageResponse.saveChanges();
 Node firstChild = messageResponse.getSOAPBody().getFirstChild();

 // create a unmarshaller instance so we can convert the SOAP body into JAXB objects
 Unmarshaller unmarshaller = V2_JAXB_CONTEXT.createUnmarshaller();
 return unmarshaller.unmarshal(firstChild);
}
```

C++ (SOAP over HTTP)

Unlike Java, C++ does not offer any built-in libraries to handle SOAP over HTTP. Therefore the C++ SOAP over HTTP example makes use of third party libraries to simplify the task of creating a web services client.

The libraries mentioned in this article are not included in the example archive and must be obtained from their respective projects. Other software is necessary for building the example. The following is a list of requirements for the project:

- gSOAP
  - http://sourceforge.net/projects/gsoap2/files/gSOAP/gsoap_2.8.8.zip/download
  - This provides the web service client functionality
- OpenSSL
  - http://www.openssl.org/source/openssl-1.0.1c.tar.gz
  - This is a required library needed by gSOAP.
- ActivePerl:
  - http://downloads.activestate.com/ActivePerl/releases/
  - This is required to build OpenSSL. Obtain the latest version.
- Visual Studio
  - This is required to build the libraries
  - The example provided in this article includes a Visual Studio project that contains the example code, and default directories where generated code, libraries and include files should be placed

While this example is not technically limited to Windows only, it only covers Windows usage of tools and libraries. Be sure to read the project documentation for gSOAP and OpenSSL for details on usage for other platforms.

To get started, obtain gSOAP and OpenSSL, and make sure ActivePerl and Visual Studio are properly installed. First, the OpenSSL library should be compiled by executing the commands in Example 22. This uses ActivePerl to configure the OpenSSL project and create the files needed to compile using Visual Studio. Note that the --prefix portion of the command tells the build where the installed output will be located.

Example 22. Using ActivePerl to prepare OpenSSL for compiling via command line

```
C:\temp\temp_openssl\openssl-1.0.1c>C:\Perl64\bin\perl.exe Configure VC-WIN32
no-asm --prefix=C:/temp/temp_openssl/install

C:\temp\temp_openssl\openssl-1.0.1c>ms\do_ms
```

Next open a Visual Studio Developer Command Prompt in order to get the proper Visual Studio environment variables, and execute the commands shown in Example 23. If all goes well, this will produce OpenSSL output in the install folder specified in the command line.

Example 23. Using Microsoft Visual Studio to compile OpenSSL

```
C:\temp\temp_openssl\openssl-1.0.1c>nmake -f ms\ntdll.mak
C:\temp\temp_openssl\openssl-1.0.1c>nmake -f ms\ntdll.mak install
```

From the "install" folder, copy the contents of the include folder (e.g. C:\temp\temp_openssl\install\include\openssl) into the example project location (e.g. `c:\ScoringClientExamples\C++\SOAP\gSoapCPPScoringExample\gSoapCPPScoringExample\OpenSSL\inc\openssl`). Copy the libeay32.lib and ssleay32.lib from the "install" lib directory (e.g. C:\temp\temp_openssl\install\lib) to the example project location (e.g. `c:\ScoringClientExamples\C++\SOAP\gSoapCPPScoringExample\gSoapCPPScoringExample\OpenSSL\lib`).

The gSOAP project comes with pre-built executables wsdl2h.exe and soapcpp2.exe, both of which are used in a two-step process to create generated code. The wsdl2h executable is a WSDL importer and data binding tool, and the soapcpp2 executable creates stub and skeleton code. Details regarding these two executables can be found in the project documentation. The generated code output from gSOAP along with select gSOAP project code will be placed into the example project location once these steps are complete.

First, modify the `typemap.dat` file included with the gSOAP archive, which is located in `gsoap-2.8\gsoap`. Add the namespace prefix "spss" which is defined as "http://xml.spss.com/ws/headers". Type bindings can be provided to bind XML schema types to C/C++ types as well. Example 24 shows the lines that should be placed into the typemap.dat file.

Example 24. Add these lines to typemap.dat gSOAP file

```
spss = "http://xml.spss.com/ws/headers"
spss__client_accept_language = | char* | char*
```

Next, run the "WSDL to Header" executable. This will create the gSOAP header file that contains information needed to create generated source code. Example 25 shows the command that was executed for this example, followed by a description of each option.

Example 25. wsdl2h.exe command and option description

```
wsdl2h.exe -o scoring.h -t "c:\downloads\gsoap-2.8\gsoap\typemap.dat"
http://localhost:8080/scoring/services/Scoring.HttpV2/WEB-INF/wsdl/scoring.wsdl

-o specifies the output file
-t specifies the typemap.dat file to use
The last argument is the URL to the WSDL file.
```

Note that the scoring.h output file created by wsdl2h.exe is only a temporary file that is fed into soapcpp2.exe, and should not be used directly in a project. Before running soapcpp2.exe, modifications will be necessary for the scoring.h file before it can be used (see Example 26). Add a line to import the header that contains the code necessary to

include the WS-Security SOAP headers found in “wsse.h”, and also define a struct that will contain the client-accept-language SOAP header as well as the WS-Security SOAP headers. The WS-Security SOAP header is handled by a gSOAP plugin, but the client-accept-language SOAP header is a proprietary SPSS SOAP header. This struct is used in the example to define the SOAP header values (see the SoapExample constructor for how this is done).

Example 26. Modify the generated scoring.h file

```
// the following line goes with the other import statements
#import "wsse.h".
// this struct appears just before the end of the scoring.h file
struct SOAP_ENV__Header {
 char *spss__client_accept_language;
 mustUnderstand struct _wsse__Security *wsse__Security;
};
```

Once the scoring.h file has been modified, it can be used to create generated code using soapcpp2.exe. There are a number of options to choose for this command. Example 27 shows the command that was executed for this example (followed by a description of each option).

Example 27. Executing soapcpp2.exe

```
soapcpp2.exe -1 -I"C:\Downloads\gsoap-2.8\gsoap\import" -C
 -d"C:\Downloads\temp\test" -j -s -x scoring.h

-1 specifies the use of SOAP 1.1 namespaces and encodings
-I specify the path for #import
-C generates client-side code only
-d Saves sources in directory specified by < path >
-j Generate C++ service proxies and objects that can share a soap struct
-s Generates deserialization code with strict XML validation checks
-x Do not generate sample XML message files
The last argument specifies the gSOAP header file
```

After the code has been generated, it must be modified to correct errors. The spss namespace prefix is defined to be http://tempuri.org/spss.xsd, which is incorrect. The namespace should be http://xml.spss.com/ws/headers. This should be corrected by doing a find/replace using a text editor. In this example, ScoringV2HttpBinding.nsmap and soapScoringV2HttpBindingProxy.cpp were modified.

Once the code has been properly modified, it can be placed into the example location. Make a copy of the generated code from the location specified in the –d option and place it into `c:\ScoringClientExamples\C++\SOAP\gSoapCPPScoringExample\gSoapCPPScoringExample\generated`. Also find the following files in the gSoap distribution (some are in `gsoap-2.8\gsoap` and others are in `gsoap-2.8\gsoap\plugin`) and place them into

`c:\ScoringClientExamples\C++\SOAP\gSoapCPPScoringExample\gSoapCPPScoringExample\gSoap`:

- dom.cpp
- mecevp.c
- mecevp.h
- smdevp.c
- smdevp.h
- soap.nsmap
- stdsoap2.cpp
- stdsoap2.h
- threads.c
- threads.h
- wsseapi.cpp
- wsseapi.h

Now that setup is complete, focus on the main C++ code, which is located at `C:\ScoringClientExamples\C++\SOAP\gSoapCPPScoringExample\gSoapCPPScoringExample\gSoapCPPScoringExample.cpp`. The `SoapExample` class is defined directly in this file (a separate header file was not used to keep the example self contained). The setup code for the example takes place in the `SoapExample` constructor. First, the gSoap implementation provides a WS-Security plug-in that allows setting the username and password for requests. Call the gSoap `soap_wsse_add_UsernameTokenText` function to do so. Next use the generated scoring proxy `ScoringV2HttpBindingProxy` to set required values for the web service endpoint URL and client accept header that were defined earlier in the code generation steps. An example of this code can be seen in Example 28. When running this example, be sure to use the correct host and port for your C&DS server.

Example 28. C++ SOAP example setup

```
// set security and language SOAP headers. The security header is
// required, but the language header is optional.

// Add security information to SOAP request
soap_wsse_add_UsernameTokenText(scoring.soap, "Id", "admin", "my_password");

// add the language header
scoring.soap->header->spss__client_accept_language = "en-US;q=1.0, en;q=0.8";
// set the URL for the server
scoring.soap_endpoint = URL.c_str();
```

In order to create a score request, use the generated code created by gSoap. The generated code consists of simple gSOAP defined objects that are “assembled” and provided the necessary request inputs. The score request creation can be seen in Example 29. Note that the names of generated classes are prefixed with a name space value that was defined during the code generation step. For example, the “getScore” wrapper object was defined

as _ns5__getScore, where the namespace prefix “ns5” was associated with namespace “http://xml.spss.com/scoring-v2/remote”. This makes reading the class names more difficult for the developer, but ensures that the class names will not conflict with similar class names that may be defined elsewhere.

Example 29. C++ gSOAP score request

```
_ns5__getScore request;
_ns5__getScoreResponse response;

// build the score request and add it to the getScore wrapper object
_ns8__scoreRequest scoreRequest;
request.ns8__scoreRequest = &scoreRequest;
// First put the configuration ID into it:
scoreRequest.id = configId;

/** Request Input Table for "Customer" */

// Create storage for inputs, rows
_ns8__requestInputRow requestInputRows1;

// Create a new request input table with a given name and the rows it will contain
_ns8__requestInputTable requestInputTable1;
std::string tableName1("Customer");
requestInputTable1.name = &tableName1;

// Add the table to our list of input tables
scoreRequest.ns8__requestInputTable.push_back(&requestInputTable1);

// Add the request inputs as a new row
requestInputTable1.ns8__requestInputRow.push_back(&requestInputRows1);

// Add each input to the request
_ns8__input age;
std::string ageStr("Age");
age.name = &ageStr;
std::string ageValue("36");
age.value = &ageValue;
requestInputRows1.ns8__input.push_back(&age);

_ns8__input incomeLevel;
std::string incomeStr("Income level");
incomeLevel.name = &incomeStr;
std::string incomeValue("HIGH");
incomeLevel.value = &incomeValue;
requestInputRows1.ns8__input.push_back(&incomeLevel);

_ns8__input education;
std::string educationStr("Education");
education.name = &educationStr;
std::string educationValue("College");
education.value = &educationValue;
requestInputRows1.ns8__input.push_back(&education);

_ns8__input carLoan;
std::string carLoanStr("Car loans");
carLoan.name = &carLoanStr;
std::string carLoanValue("2 or less");
carLoan.value = &carLoanValue;
requestInputRows1.ns8__input.push_back(&carLoan);

_ns8__input id;
std::string idStr("ID");
id.name = &idStr;
std::string idValue("1");
id.value = &idValue;
requestInputRows1.ns8__input.push_back(&id);
```

```
/** Request Input Table for "Customer Credit" */

// Create storage for inputs, rows
_ns8__requestInputRow requestInputRows2;

// Create a new request input table with a given name and the rows it will contain
_ns8__requestInputTable requestInputTable2;
std::string tableName2("Customer Credit");
requestInputTable2.name = &tableName2;

// Add the table to our list of input tables
scoreRequest.ns8__requestInputTable.push_back(&requestInputTable2);

// Add the request inputs as a new row (repeat the same row of inputs multiple times)
requestInputTable2.ns8__requestInputRow.push_back(&requestInputRows2);
requestInputTable2.ns8__requestInputRow.push_back(&requestInputRows2);
requestInputTable2.ns8__requestInputRow.push_back(&requestInputRows2);
requestInputTable2.ns8__requestInputRow.push_back(&requestInputRows2);
requestInputTable2.ns8__requestInputRow.push_back(&requestInputRows2);
requestInputTable2.ns8__requestInputRow.push_back(&requestInputRows2);

// Add each input to the request
_ns8__input id2;
std::string id2Str("ID");
id2.name = &id2Str;
std::string id2Value("1");
id2.value = &id2Value;
requestInputRows2.ns8__input.push_back(&id2);
```

Once the score request has been formed, use the scoring proxy object to submit the score request. If the result code comes back with SOAP_OK, print out the score response. This is shown in Example 30.

Example 30. Execute a score request using C++ gSOAP

```
// make a "get score" call
int rc = scoring.getScore(&request,&response);

if(rc == SOAP_OK)
{
 // print out the results
 printScoreResponse(response);
}
else
{
 std::cout << "Error! (rc: " << rc << ")" << std::endl;
 scoring.soap_stream_fault(std::cout);
}
```

C++ (REST – JSON over HTTP)

As in the previous example, C++ does not offer any built in libraries to handle JSON over HTTP. Therefore the C++ JSON over HTTP example makes use of third party libraries to simplify the task of creating a web services client.

The libraries mentioned in this article are not included in the example archive and must be obtained from their respective projects. Other software is necessary for building the example. The following is a list of requirements for the example project:

- JsonCPP
  - http://sourceforge.net/projects/jsoncpp/files/jsoncpp/0.5.0/jsoncpp-src-0.5.0.tar.gz/download
  - This is the JSON parser
- cURL
  - http://curl.haxx.se/download/curl-7.25.0.zip
  - This handles the HTTP communication
- Visual Studio
  - This is required to build the libraries
  - The example provided in this article includes a Visual Studio project that contains the example code, and default directories where libraries and include files should be placed

While this example is not technically limited to Windows, only Windows usage of tools and libraries is discussed. Be sure to read the project documentation for JsonCPP and cURL for details on usage.

To get started, obtain JsonCPP and cURL, and make sure Visual Studio is properly installed. First, the JsonCPP library should be compiled by opening the Visual Studio project that is found in `jsoncpp-src-0.5.0\makefiles\vs71`. Allow Visual Studio to upgrade the project to the current version, if applicable. Once that is complete, select lib_json in the solution explorer, right click lib_json and choose build. The output from the build is located in `jsoncpp-src-0.5.0\build\vs71\debug\lib_json`. Copy the `json_vc71_libmtd.lib` library into the example project under `c:\ScoringClientExamples\C++\REST\SimpleCPPScoringExample\SimpleCPPScoringExample\jsoncpp\lib`. Copy the include files found in `jsoncpp-src-0.5.0\include\json` to `c:\ScoringClientExamples\C++\REST\SimpleCPPScoringExample\SimpleCPPScoringExample\jsoncpp\include`.

Next, compile curl by opening the Visual Studio file `curl-7.25.0\vc6curl.dsw`, and allowing Visual Studio to upgrade the project to the current version. Once that is complete, select libcurl in the solution explorer, right click libcurl and choose build. The output from the build is located in `curl-7.25.0\lib\DLL-Debug`. Copy the `libcurld.dll` and `libcurld_imp.lib` to

C:\ScoringClientExamples\C++\REST\SimpleCPPScoringExample\SimpleCPPScoringExample\libcurl\lib. Also copy the include files (all *.h files) found in curl-7.25.0\include\curl to C:\ScoringClientExamples\C++\REST\SimpleCPPScoringExample\SimpleCPPScoringExample\libcurl\include\curl.

The C++ (REST – JSON over HTTP) example is similar in concept to the Java REST examples (which can be found here and here), where a C++ library (cURL) is used to communicate via HTTP and another C++ library (JsonCPP) is used to parse JSON data. The code for this example can be found in C:\ScoringClientExamples\C++\REST\SimpleCPPScoringExample\SimpleCPPScoringExample\SimpleCPPScoringExample.cpp. The RestExample class is defined directly in this file (a separate header file was not used to keep the example self contained). The setup code for the example takes place in the RestExample constructor, and teardown in the destructor. The example is mainly concerned with initializing and then eventually cleaning up the cURL library. See Example 31 for the example.

Example 31. Setting up and shutting down cURL

```
// Do any setup work in this constructor
RestExample::RestExample()
{
 // We need to initialize curl... each call to this must have a matching call
 // to curl_global_cleanup()
 curl_global_init(CURL_GLOBAL_ALL);
}

// Do any shutdown work in this destructor
RestExample::~RestExample()
{
 /* we're done with libcurl, so clean it up */
 curl_global_cleanup();
}
```

In order to communicate with the C&DS server, an execute method is called each time a request is sent, which opens a HTTP connection using the cURL library (see Example 32). Assuming the cURL library is initialized, the URL that is passed into the execute method is escaped, and a structure is created to capture the response data that comes back from the server. The structure is passed into a call back function that will copy the data supplied from the cURL library, into the structure. This callback function is called any number of times by cURL as data is acquired. In this way the cURL library provides the data to the application and the application controls the lifecycle of the memory allocated. The structure and callback function can be seen in Example 33.

Before instructing cURL to send a request, a number of different “options” need to be set using curl_easy_setopt, such as the URL to contact, HTTP authorization headers, user agent, content type, language, and request method (e.g. POST or GET). Once all of the options are set, the request can be sent using curl_easy_perform, and then finally cleanup cURL and the memory that was allocated.

Example 32. C++ REST request execution example

```
void RestExample::executeRequest(const std::string uri,
std::string &jsonResponseStr, const std::string &jsonRequestString)
{
 /* init the curl session */
 CURL *curl_handle = curl_easy_init();
 if(curl_handle)
 {
  // encode the URI to be sure we don't use illegal characters in the URL
  char *escaped = curl_easy_escape(curl_handle,uri.c_str(),uri.length());
  std::string encodedURI;
  encodedURI += escaped;
  curl_free(escaped);

  struct MemoryStruct responseData;
  responseData.memory = (char *)malloc(1);  /* will be grown as needed by realloc */
  responseData.size = 0;    /* no data at this point */

  // create and specify URL
  std::string url = baseURL + encodedURI;
  curl_easy_setopt(curl_handle, CURLOPT_URL, url.c_str());

  // send all data to this function
  // we pass our 'responseData' struct to the callback function
  curl_easy_setopt(curl_handle, CURLOPT_WRITEFUNCTION, ResponseCallback);
  curl_easy_setopt(curl_handle, CURLOPT_WRITEDATA, (void *)&responseData);

  // Set Authorization
  curl_easy_setopt(curl_handle, CURLOPT_USERNAME, "admin");
  curl_easy_setopt(curl_handle, CURLOPT_PASSWORD, " my_password");

  // libcurl can be more verbose. Uncomment this line if desired.
  //curl_easy_setopt(curl_handle, CURLOPT_VERBOSE, 1);

  // some servers don't like requests that are made without a user-agent
  // field, so we provide one
  curl_easy_setopt(curl_handle, CURLOPT_USERAGENT, "libcurl-agent/1.0");

  // Set other header values
  struct curl_slist *pRequestHeaderList = NULL;
  pRequestHeaderList = curl_slist_append(pRequestHeaderList,
   "Content-Language: en_US");
  pRequestHeaderList = curl_slist_append(pRequestHeaderList,
   "Content-Type: application/json; charset=utf-8");
  curl_easy_setopt(curl_handle, CURLOPT_HTTPHEADER, pRequestHeaderList);

  // if we have a non-empty string, use a HTTP post
  if(jsonRequestString != "")
  {
   curl_easy_setopt(curl_handle, CURLOPT_HTTPPOST, 1);
   curl_easy_setopt(curl_handle, CURLOPT_POSTFIELDS, jsonRequestString.c_str());
   curl_easy_setopt(curl_handle, CURLOPT_POSTFIELDSIZE, jsonRequestString.size());
  }
  // otherwise use a HTTP GET
  else
  {
   curl_easy_setopt(curl_handle, CURLOPT_HTTPGET, 1);
  }

  // send the request
  CURLcode retval = curl_easy_perform(curl_handle);
  //printf("retval: %d\n", retval);

  long httpReturnCode = 0;
  curl_easy_getinfo(curl_handle, CURLINFO_HTTP_CODE, &httpReturnCode);
  //printf("HTTP return code: %d\n", httpReturnCode);

  /* cleanup curl stuff */
  curl_slist_free_all(pRequestHeaderList);
```

```
  curl_easy_cleanup(curl_handle);

  /*
  * Now, our responseData.memory points to a memory block that is responseData.size
  * bytes big and contains the server response.
  */
  // printf("\n");
  // printf("Body bytes retrieved: %lu\n", (long)responseData.size);
  if (responseData.memory)
  {
   //printf("%s\n", responseData.memory);
   // copy the data into the response string
   jsonResponseStr += responseData.memory;
   // free the memory that we allocated
   free(responseData.memory);
  }
 }
}
```

Example 33. C++ struct and callback method for cURL communication

```
// This structure is used to store the response data from a REST call.
// It is passed into libcurl as user data
struct MemoryStruct {
  char *memory;
  size_t size;
};

// This callback function is called whenever libcurl has data that needs to be saved.
// This could be called any number of times with however much data is available.
//  contents = pointer to the data
//  size     = size of the dat
//  nmemb    = number of bytes
//  userp    = user data (should be a MemoryStruct)
const size_t ResponseCallback(void *contents, size_t size, size_t nmemb, void *userp)
{
 // determin full size of the data to be saved
 size_t realsize = size * nmemb;
 struct MemoryStruct *mem = (struct MemoryStruct *)userp;

 // resize the user defined memory storage
 mem->memory = (char *)realloc(mem->memory, mem->size + realsize + 1);
 if (mem->memory == NULL) {
  /* out of memory! */
  std::cout << "not enough memory (realloc returned NULL)" << std::endl;
  exit(EXIT_FAILURE);
 }

 // copy the contents into the user defined memory
 memcpy(&(mem->memory[mem->size]), contents, realsize);
 mem->size += realsize;
 mem->memory[mem->size] = 0;

 return realsize;
}
```

The code just described focused on the HTTP communications, but before the getScore function can call the execute function, it must build the score request (see the getScore function in Example 34).

Example 34. The C++ rest getScore call

```
void RestExample::getScore()
{
 // build the Score Request
 Json::Value jsonScoreRequest(Json::objectValue);
 buildScoreRequest(jsonScoreRequest);

 Json::FastWriter writer;
 std::string jsonScoreReqStr = writer.write(jsonScoreRequest);

 // execute request
 std::string jsonScoreResponseStr;
 executeRequest(std::string("configuration/") + configId + std::string("/score"),
jsonScoreResponseStr, jsonScoreReqStr);

 // parse and display response
 Json::Value jsonScoreResult;
 jsonParse(jsonScoreResponseStr, jsonScoreResult);
 printScoreResponse(jsonScoreResult);
}

void RestExample::jsonParse(const std::string jsonStr, Json::Value &v)
{
 Json::Reader reader;
 bool parsingSuccessful = reader.parse( jsonStr, v );
 if ( !parsingSuccessful )
 {
  // report to the user the failure and their locations in the document.
  std::cout  << "Failed to parse\n"
             << reader.getFormatedErrorMessages();
  exit(EXIT_FAILURE);
 }
}
```

In order to build the score request JSON data, the JsonCPP library is used. The code for this is shown in Example 35. A score request can be created using a hierarchy of Json::Value objects. Once a Json::Value object is added into the hierarchy, a copy of the object is made and stored internally. In other words, the objects are not stored by reference. This behavior has a side effect, if a Json::Value object is created manually and added into the hierarchy, any changes made to the original manually created object won't have any effect on the object that is stored in the hierarchy.

Because of the copy behavior inside the Json::Value object, it is best to let the Json library create the objects, and refer to the objects it creates. Fortunately, the Json::Value API makes this a simple task. By making use of the [] operator, refer to objects by name (if the Json::Value represents an "object") or by index (if the Json::Value represents an "array"). The array indices are referred to using an unsigned int literal (e.g. 0u). If a Json::object does not exist (by either key or by array index), a Json::Value object is created automatically.

Example 35. Creating the score request for the C++ REST example

```
// set the configuration ID
scoreRequest["id"] = "Example Credit 1";

// Add a "Customer" input table with a single row that contains 5 inputs
scoreRequest["requestInputTable"][0u]["name"] = "Customer";
scoreRequest["requestInputTable"][0u]["requestInputRow"][0u]["input"][0u]["name"] =
 "Age";
scoreRequest["requestInputTable"][0u]["requestInputRow"][0u]["input"][0u]["value"] =
 "36";
scoreRequest["requestInputTable"][0u]["requestInputRow"][0u]["input"][1u]["name"] =
 "Income level";
scoreRequest["requestInputTable"][0u]["requestInputRow"][0u]["input"][1u]["value"] =
 "HIGH";
scoreRequest["requestInputTable"][0u]["requestInputRow"][0u]["input"][2u]["name"] =
 "Education";
scoreRequest["requestInputTable"][0u]["requestInputRow"][0u]["input"][2u]["value"] =
 "College";
scoreRequest["requestInputTable"][0u]["requestInputRow"][0u]["input"][3u]["name"] =
 "Car loans";
scoreRequest["requestInputTable"][0u]["requestInputRow"][0u]["input"][3u]["value"] =
 "2 or less";
scoreRequest["requestInputTable"][0u]["requestInputRow"][0u]["input"][4u]["name"] =
 "ID";
scoreRequest["requestInputTable"][0u]["requestInputRow"][0u]["input"][4u]["value"] =
 "1";

// Add a "Customer Credit" input table with 6 rows, each of which contains a single input
scoreRequest["requestInputTable"][1u]["name"] = "Customer Credit";

scoreRequest["requestInputTable"][1u]["requestInputRow"][0u]["input"][0u]["name"] = "ID";
scoreRequest["requestInputTable"][1u]["requestInputRow"][0u]["input"][0u]["value"] = "1";
scoreRequest["requestInputTable"][1u]["requestInputRow"][1u]["input"][0u]["name"] = "ID";
scoreRequest["requestInputTable"][1u]["requestInputRow"][1u]["input"][0u]["value"] = "1";
scoreRequest["requestInputTable"][1u]["requestInputRow"][2u]["input"][0u]["name"] = "ID";
scoreRequest["requestInputTable"][1u]["requestInputRow"][2u]["input"][0u]["value"] = "1";
scoreRequest["requestInputTable"][1u]["requestInputRow"][3u]["input"][0u]["name"] = "ID";
scoreRequest["requestInputTable"][1u]["requestInputRow"][3u]["input"][0u]["value"] = "1";
scoreRequest["requestInputTable"][1u]["requestInputRow"][4u]["input"][0u]["name"] = "ID";
scoreRequest["requestInputTable"][1u]["requestInputRow"][4u]["input"][0u]["value"] = "1";
scoreRequest["requestInputTable"][1u]["requestInputRow"][5u]["input"][0u]["name"] = "ID";
scoreRequest["requestInputTable"][1u]["requestInputRow"][5u]["input"][0u]["value"] = "1";
```

Now that the request JSON data has been created, return back to Example 34 where the JSON data that was built is serialized to a string and sent with the `executeRequest` function. Once a response has been returned, the raw string is converted back into a Json::Value object, and printed.

.NET (SOAP over HTTP)

When using .NET, users can take advantage of the Windows Communication Foundation (WCF) framework to create a web service client. This makes the process of creating a SOAP client relatively simple, but it does have some drawbacks covered later.

The framework will generate code based off of the scoring WSDL/XSD files, allowing users to interact with a simple to use proxy object that represents the scoring service endpoint. The proxy endpoint class is passed a score request object that contains the request inputs. This process is very similar to the other SOAP over HTTP examples seen in this article.

In modern versions of Microsoft Visual Studio, a web service client can be created using the tools built into the IDE. To do so, right click the references folder in the solution explorer and select "Add Service Reference". A panel prompts for a web service URL where the WSDL can be found (for example, `http://<your_host_name>:<your_port>/scoring/services/Scoring.HttpV2?wsdl`), and the .NET namespace where the generated code will placed.

When the "Go" button is pressed in the dialog, it will discover the WSDL and display the services provided by this web service, and then click "OK" to create the service reference. By default, the service reference that is created hides all of the files that are generated from the WSDL. To see the files that are generated, select the project, and choose Project->Show All Files from the menu.

There are limitations with the WCF generated code due to the way the .NET framework interprets the WSDL. When using C&DS 5, the scoring service WSDL would be interpreted in such a way that would result in collections that would not allow the WCF generated code to work as-is. The code would generate multidimensional arrays when it should have generated a single dimension array. For C&DS 6, changes were made to the WSDL that works around WCF limitations. The C&DS 5 workarounds are documented in the Scoring_Service_Developers_Guide.pdf. These workarounds required developers to open the `<ServiceReference>\Reference.svcmap\Reference.cs` file and make manual changes. The example provided in this article was originally generated using C&DS 5, and the Reference.cs code was modified. If targeting C&DS 6, update the service reference by pointing to the C&DS 6 WSDL. This can be accomplished by right clicking the service reference and choosing "Configure Service Reference" and updating the service address, which will re-generate the code automatically.

Another limitation with WCF framework is that the default behavior for WCF clients is to disallow the use of WS-Security UsernameToken elements over unsecured HTTP connections. The WCF framework also cannot understand the SOAP fault format that is returned from the C&DS web services. Using HTTPS is beyond the scope of this article, so a DLL called IBM.SPSS.WCF.Utilities.dll is included which allows the endpoint behaviors required in the example. As shown in Example 36, the two new behaviors are added to the client endpoint, one which applies the UsernameToken SOAP header and the

other that intercepts SOAP faults and formats the fault so WCF interprets the fault correctly.

Example 36. Initializing the WCF scoring client

```
private static ScoringV2Client _client = null;
/// <summary>
/// Returns an instance of the ScoringClient with the requisite behaviors added to it.
/// </summary>
public static ScoringV2Client Client
{
 get
 {
  if (_client == null)
  {
   _client = new ScoringV2Client();
   // Add the endpoint behaviors that will ultimately add the Usernametoken
   // security header to the SOAP message with a username/password of
   // "username" and "password", and allow the Axis formatted SOAP
   // faults to be re-formatted as valid WCF SOAP faults.
   _client.Endpoint.Behaviors.Add(
      new ApplyClientInspectorsBehavior(
       new HeaderInjectionMessageInspector(
     new UsernameTokenSecurityHeader("admin", "my_password")
       ),
    new SOAPFaultFormatMessageInspector()
    )
   );
  }
  return _client;
 }
}
```

The URL endpoint will likely need to change to a different host and port. Making this change with the Visual Studio user interface will automatically regenerate the client code, which can be problematic if there were changes made to the generated client code. In order to make the change without generating the code again, simply edit the app.config file for the project manually. Look for the endpoint address as shown in Example 37, and change the value as needed.

Example 37. Changing the endpoint URL

```
<system.serviceModel>
 <diagnostics>
  <messageLogging logEntireMessage="true" logMalformedMessages="true"
   logMessagesAtServiceLevel="true" logMessagesAtTransportLevel="true" />
 </diagnostics>

 <client>
  <endpoint address="http://localhost:8080/scoring/services/Scoring.HttpV2"
   binding="basicHttpBinding" bindingConfiguration=""
   contract="IBM.SPSS.Scoring.ScoringV2" name=""/>
 </client>
</system.serviceModel>
```

Once the client proxy is setup, web service request inputs and score request objects can be created, and the getScore call can be made. This is easily accomplished using the code shown in Example 38.

Example 38. Execute a getScore call using .NET WCF

```
public static void GetScore(string configurationId)
{
 // Create the scoreRequest1 object and set the scoring configuration ID
 // we want to score against.
 scoreRequest1 request = new scoreRequest1();
 request.id = configurationId;

 // Create the data row for the first input table. It will have 5 columns.
 input1[] customerRow = new input1[]
  { new input1(), new input1(), new input1(), new input1(), new input1() };

 // set the values for the 5 inputs of the Customer table
 customerRow[0].name = "Age";
 customerRow[0].value = "36";

 customerRow[1].name = "Income level";
 customerRow[1].value = "HIGH";

 customerRow[2].name = "Education";
 customerRow[2].value = "College";

 customerRow[3].name = "Car loans";
 customerRow[3].value = "2 or less";

 customerRow[4].name = "ID";
 customerRow[4].value = "1";

 // Create the Customer table, and give it just a single row which is the
 // previously created customerRow.
 requestInputTable customerTable = new requestInputTable();
 customerTable.name = "Customer";
 customerTable.requestInputRow = new requestInputRow[] { new requestInputRow() };
 customerTable.requestInputRow[0].input = customerRow;

 // Create the Customer Credits input row, which will have a single column
 input1[] customerCreditRow = new input1[] { new input1()};
 customerCreditRow[0].name = "ID";
 customerCreditRow[0].value = "1";

 // Create the Customer Credit table, and give it 6 rows, each being the
 // previously created customerCreditRow
 requestInputTable customerCreditTable = new requestInputTable();
 customerCreditTable.name = "Customer Credit";
 customerCreditTable.requestInputRow = new requestInputRow[] {
  new requestInputRow(), new requestInputRow(), new requestInputRow(),
  new requestInputRow(), new requestInputRow(), new requestInputRow() };
 customerCreditTable.requestInputRow[0].input = customerCreditRow;
 customerCreditTable.requestInputRow[1].input = customerCreditRow;
 customerCreditTable.requestInputRow[2].input = customerCreditRow;
 customerCreditTable.requestInputRow[3].input = customerCreditRow;
 customerCreditTable.requestInputRow[4].input = customerCreditRow;
 customerCreditTable.requestInputRow[5].input = customerCreditRow;

 // Now give both tables to the score request
 request.requestInputTable = new requestInputTable[] {
  customerTable, customerCreditTable};
// Make the scoring call, and then print the result
 scoreResult1 result = Client.getScore(request);
 PrintScoreResult(result);
}
```

HTML (REST – JSON over HTTP)

Out of all the REST examples provided, this is the easiest to use by far because it uses native HTTP communication and JSON parsing support. The example consists of a very basic HTML page which relies on JavaScript to make the score requests, and places the results into the HTML page dynamically. The example can be found in the `C:\ScoringClientExamples\HTML\REST\` directory, and the file names are `SimpleScoringExample.js` and `SimpleScoringExample.html`.

To use the example, place these files into the same application server that runs C&DS. The example defines the base URL for the C&DS server as `baseURL = '/scoring/rest/';` and picks up the host and port by default. If the URL is changed or the files are hosted in a different server, be aware that cross site scripting security issues may prevent the example from working.

In order to send an HTTP request and get a response, the example uses the `setupRequest` function which is shown in Example 39. This function makes use of the built in browser XMLHttpRequest object. Depending on the browser in use, it may be difficult to get access to this object, which will cause the example to fail. The example was tested using IE 8 and Firefox 3.5. Note that for IE 8, the meta tag `<meta http-equiv="X-UA-Compatible" content="IE=8" />` was added to ensure that the `XMLHttpRequest` object is exposed in the page. When the request is sent, it uses either GET or POST and tries to open the given URL with the user credentials provided using a synchronous connection (i.e. the function call will block until the response is returned). Note that in the example the `Accept-Language`, `Content-Type` and `If-Modified-Since` HTTP headers are set. Setting the `If-Modified-Since` header is very important to ensure that the browser does not return a cached copy of the result.

Example 39. Setting up the XMLHttpRequest object

```
function setupRequest(method, url)
{
 // Put more code here in case you are concerned about browsers that do
 // not provide XMLHttpRequest object directly
 if (window.XMLHttpRequest)
 {
  xmlhttp = new XMLHttpRequest();
 }

 if (window.ActiveXObject)
 {
  xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
 }
 xmlhttp.open(method, url, false, 'admin', 'my_password');
 // set language and JSON content type
 xmlhttp.setRequestHeader('Accept-Language', 'en_US');
 xmlhttp.setRequestHeader('Content-Type', 'application/json; charset=utf-8');
 // skip browser caching
 xmlhttp.setRequestHeader('If-Modified-Since', new Date(0));

 return xmlhttp;
}
```

Before a call can be made, the score request inputs must be setup. In this case the process involves creating a JavaScript object which contains the appropriate hierarchy of JavaScript objects and arrays. Example 40 shows the details of this approach.

Example 40. Create a JavaScript object hierarchy for a score request

```
function createScoreRequest(configID)
{
 // Build a JavaScript object that will be converted into JSON stringdata.
 // The JSON data represents a score request.
 // First put the configuration ID into it:
 var jsonScoreRequest = {'id': configID};

 // create storage for all of the request input tables and add it to the request
 var requestInputTables = [];
 jsonScoreRequest['requestInputTable'] = requestInputTables;

 /** Request Input Table for "Customer" **/
 // Create storage for inputs, rows
 var requestInputs1 = [];
 var requestInputRows1 = [];

 // Create a new request input table with a given name and the rows it will contain
 var requestInputTable1 = {'name': 'Customer'};
 requestInputTable1['requestInputRow'] = requestInputRows1;

 // Add the table to our list of input tables
 requestInputTables.push(requestInputTable1);

 // Add the request inputs as a new row
 requestInputRows1.push({'input': requestInputs1});

 // Add each input to the request
 requestInputs1.push({'name': 'Age', 'value': '36'});
 requestInputs1.push({'name': 'Income level', 'value': 'HIGH'});
 requestInputs1.push({'name': 'Education', 'value': 'College'});
 requestInputs1.push({'name': 'Car loans', 'value': '2 or less'});
 requestInputs1.push({'name': 'ID', 'value': '1'});

 /** Request Input Table for "Customer Credit" **/
 // Create storage for inputs, rows
 var requestInputs2 = [];
 var requestInputRows2 = [];

 // Create a new request input table with a given name and the rows it will contain
 var requestInputTable2 = {'name': 'Customer Credit'};
 requestInputTable2['requestInputRow'] = requestInputRows2;

 // Add the table to our list of input tables
 requestInputTables.push(requestInputTable2);

 // Add the request inputs as a new row (repeat the same row of inputs multiple times)
 requestInputRows2.push({'input': requestInputs2});
 requestInputRows2.push({'input': requestInputs2});
 requestInputRows2.push({'input': requestInputs2});
 requestInputRows2.push({'input': requestInputs2});
 requestInputRows2.push({'input': requestInputs2});
 requestInputRows2.push({'input': requestInputs2});

 // Add each input to the request
 requestInputs2.push({'name': 'ID', 'value': '1'});

 return jsonScoreRequest;
}
```

Once the JavaScript object is created, it is converted into a string and sent using the XMLHttpRequest object, and the resulting JSON string is converted back to a JavaScript

object using the JavaScript `eval()` function, and then printed out. Example 41 shows the details of this approach.

Example 41. Making the getScore call using JavaScript

```
function getScore()
{
        // reset the output area in the HTML document
        resetOutput();

 // This is the configuration ID for the configuration we want to score against
 var configID = 'Example Credit 1';

 var jsonScoreRequest = createScoreRequest(configID);

 // make a "get score" call
 var url = baseURL + 'configuration/' + configID + '/score';
 xmlhttp = setupRequest("POST",url);
 // Issue request
 xmlhttp.send(JSON.stringify(jsonScoreRequest));

 getScoreResponse = eval( "(" + xmlhttp.responseText + ")" );

 // print out the results
 printScoreResponse(getScoreResponse);
}
```

Python (REST – JSON over HTTP)

This example uses the Python scripting language to invoke the scoring REST API. Python can be downloaded from http://www.python.org/. The example is located in `C:\ScoringClientExamples\Python\REST\SimpleScoringExample.py`. The example was created using Python 2.7, and relies only on Python standard library modules. When running the example, make sure to change the host and port for the `baseURL`.

In order to handle HTTP communication, the Python example relies on the urllib2 module. As shown in Example 42, the urllib2 module can be configured to use HTTP basic authentication for a given URL, and set HTTP headers as needed.

Example 42. Setting up a urllib2.Request object in Python

```
def setupRequest(url):
    # setup basic HTTP authentication
    pwdMgr = urllib2.HTTPPasswordMgrWithDefaultRealm()
    pwdMgr.add_password(None, url, 'admin', 'my_password')

    handler = urllib2.HTTPBasicAuthHandler(pwdMgr)

    opener = urllib2.build_opener(handler)

    urllib2.install_opener(opener)

    # create the request object
    req = urllib2.Request(url)
    # set language and JSON content type
    req.add_header('Accept-Language', 'en_US')
    req.add_header('Content-Type', 'application/json; charset=utf-8')
    return req
```

Just as with the other REST examples, the score request inputs must be setup. By comparing this code with the JavaScript code found in the previous example, the similarities are apparent. In Python, dictionary and array objects are used to build the score request inputs. Example 43 shows the details of this approach.

Example 43. Create a Python dictionary/array object hierarchy for a score request

```
def createScoreRequest(configID):
    # First put the configuration ID into it:
    jsonScoreRequest = {'id': configID}

    # create storage for all of the request input tables and add it to the request
    requestInputTables = []
    jsonScoreRequest['requestInputTable'] = requestInputTables

    ''' Request Input Table for "Customer" '''
    # Create storage for inputs, rows
    requestInputs1 = []
    requestInputRows1 = []

    # Create a new request input table with a given name and the rows it will contain
    requestInputTable1 = {'name': 'Customer'}
    requestInputTable1['requestInputRow'] = requestInputRows1

    # Add the table to our list of input tables
    requestInputTables.append(requestInputTable1)

    # Add the request inputs as a new row
    requestInputRows1.append({'input': requestInputs1})

    # Add each input to the request
    requestInputs1.append({'name': 'Age', 'value': '36'})
    requestInputs1.append({'name': 'Income level', 'value': 'HIGH'})
    requestInputs1.append({'name': 'Education', 'value': 'College'})
    requestInputs1.append({'name': 'Car loans', 'value': '2 or less'})
    requestInputs1.append({'name': 'ID', 'value': '1'})

    ''' Request Input Table for "Customer Credit" '''

    # Create storage for inputs, rows
    requestInputs2 = []
    requestInputRows2 = []

    # Create a new request input table with a given name and the rows it will contain
    requestInputTable2 = {'name': 'Customer Credit'}
    requestInputTable2['requestInputRow'] = requestInputRows2

    # Add the table to our list of input tables
    requestInputTables.append(requestInputTable2)

    # Add the request inputs as a new row (repeat the same row of inputs multiple times)
    requestInputRows2.append({'input': requestInputs2})
    requestInputRows2.append({'input': requestInputs2})
    requestInputRows2.append({'input': requestInputs2})
    requestInputRows2.append({'input': requestInputs2})
    requestInputRows2.append({'input': requestInputs2})
    requestInputRows2.append({'input': requestInputs2})

    # Add each input to the request
    requestInputs2.append({'name': 'ID', 'value': '1'})

    return jsonScoreRequest;
```

Once the Python dictionary object is created, the `json` module can be used to convert the Python dictionary object into a string, which is provided to the `urllib2` module via the `urlopen` function along with the `urllib2.Request` object. The return value from the `urlopen` function is a file-like object that represents the response data. `StringIO` is used to convert the file-like object into a string, which the `json` module uses to convert the string into a Python object, which is then printed out. This entire process can be seen in Example 44.

Example 44. Making the getScore call using Python

```
def getScore():
    # This is the configuration ID for the configuration we want to score against
    configID = 'Example Credit 1'

    # make sure the configuration name is URL encoded
    configID = urllib.quote(configID)

    # Build a python dictionary object that will be converted into JSON data.
    # The JSON data represents a score request.
    jsonScoreRequest = createScoreRequest(configID)

    # make a "get score" call
    url = baseURL + 'configuration/' + configID + '/score'
    req = setupRequest(url)
    data = urllib2.urlopen(req,json.dumps(jsonScoreRequest)).read()
    getScoreResponse = json.load(StringIO(data))

    # print out the results
    printScoreResponse(getScoreResponse)
```

Conclusion

The purpose of this article is to convey more details about the scoring service API and the ease with which it can be executed. As a developer, it is simplest to start to learn the scoring API by using the language with the greatest familiarity, and then expand out into other languages and technologies if desired. Some people find that SOAP web services are difficult to understand, but the complexity of SOAP web services is often hidden by using existing technologies which conceal the SOAP messages and only expose the web service client proxies, which are much easier to use.

About the author

John Hunkins is a software engineer working on the IBM SPSS Collaboration and Deployment Services product. He is currently a member of the scoring service component team.

Trademarks