

NEMSIS v3 Web Services Guide

Date

November 2, 2011

November 14, 2011 (FINAL)

Authors

Su Shaoyu - NEMSIS Lead Developer

N. Clay Mann – NEMSIS P.I.

Contributors

Grant Dittmer – Digital Innovations

Jerome Soller – Cognitech

Josh Legler - Utah Department of Health

EMSPIC Team – University of North Carolina-Chapel Hill

NEMSIS v3 Web Services Guide

Background

In NEMSIS V2, the dominant data submission mechanisms are ftp and web submission using Java Applet. Such procedures are manual and difficult to automate. The NEMSIS TAC has implemented a simple Web Services API for V2 data submission. Although used only by a few states, it has proved to be an efficient alternative. For software applications seeking V3 compliance, data submission via Web Services (WS) will be required. We hope the EMS software development community will take advantage of this mature technology and expand its usage to facilitate business workflow control and data exchange. If implemented properly, standardized NEMSIS WS API will provide the following advantages:

1. Fully automated electronic data exchange (including submission and retrieval) between the WS Consumer's system and the WS Provider's system. Human intervention could be minimized.
2. Better transaction management: The result of data submission / retrieval could be easily monitored without worrying about timeout. In NEMSIS systems, XML and business validation report could be available in real time, from the same communication channel.
3. Possibility to move to real-time / surveillance.
4. Some benefits are intrinsic characteristics of WS, such as vendor independence, scalability, and wide acceptance/adoption.

Purpose of NEMSIS v3 Web Services API

A unified interface to support data exchange between distributed NEMSIS v3 systems

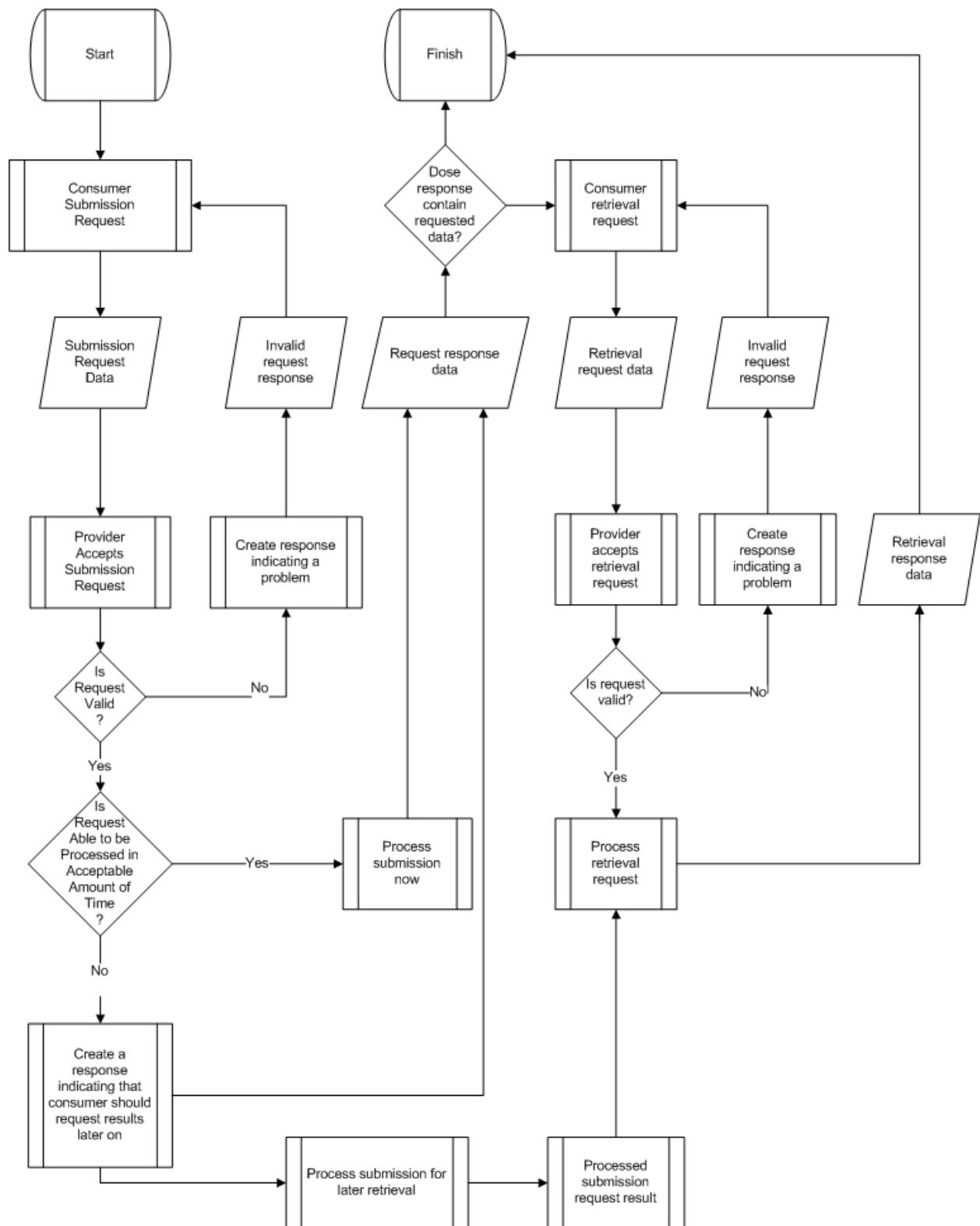
Major design goals

1. Flexibility: no constraints on how to implement, which programming language to use, or what platform to deploy.
2. Efficiency: instant response, quick turn-around.
3. Unity: minimize confusion over information exchange.

Disclaimer

If any discrepancy is found between this guide and the NEMSIS Version 3 Compliance Testing Guide, the compliance guide will take precedence.

Work Flow Diagram (Data Submission Request Only):



Main Web Service functions

1. Submit Data
 - a. Synchronous (required*)
 - b. Asynchronous (recommended**)
2. Retrieve Data Process Status (recommended**)
 - a. Asynchronous
3. Helper function – query data submission size limit (required*)
4. Search (recommended**)

*: “required” means all software vendors need to implement such interfaces.

**: “recommended” means that the interfaces are preferred, but not required.

Detail requirements and considerations

Protocol

1. SOAP Web Service is recommended
2. There are plenty of discussions regarding the pros and cons of SOAP vs. RESTful Web Services implementations. Basically, it is an “oranges vs. apples” argument. For the sake of unity, we propose the use of the SOAP protocol.
3. RESTful Web Services will continue to be supported for version 2 data submission to NEMSIS TAC.

Security

HTTPS is the required communication channel. WS messages might include sensitive information, such as login credentials and Protected Health Information (PHI). We must protect these types of data at the transport-level. Further discussion about security is included in a later section.

Status Codes

1. It is up to the vendor to decide whether to utilize the SOAP header for error reporting. At the NEMSIS TAC, we prefer to include most errors in the Response Data and use the appropriate Status Code as indicators.
2. NEMSIS WSDL defines a set of standard response codes. All WS response messages include a status code.
 - a. All error codes are negative integers.
 - b. All success codes are positive integers.
 - c. Integer “0” means that the requested action is not completed yet. The client should query the server later to retrieve final processing status.
 - d. States or vendors could design their own error codes, with integers smaller than -100. Similarly, custom success codes should use integers greater than 100. It is hard to

distinguish state custom codes and vendor custom codes. There are many cases in which state and vendor means the same thing.

- The vendors are not required to have all response codes predefined in the WSDL. Only a subset of them will be required for certification. They are all for basic data submission workflow:

Category	Code Value	Meaning
PrivilegeErrorCodes	-1	Invalid username and/or password
	-2	Permission denied to the client for the operation
	-3	Permission denied to the client for that organization
SubmitDataProcessCodes	-12	Failed import of a file, because of failing XML validation
	-13	Failed import of a file, because of [FATAL] level Schematron rule violation
	1	Successful import of a file
QueryLimitCodes	51	Successful operation of QueryLimit

- To improve data exchange, if a state/vendor uses codes from -100 to 100, it is required to adopt the definition from the NEMESIS WSDL. For example, it is bad to use “-1” as a status code for “Failed import of a file, because of the [FATAL] level Schematron rule violation” since it has been defined in the WSDL as “Invalid username and/or password”.

Important Common Objects

DataPayload

Payload object could be used to submit data, or return search results, or return reports. It is a union of three data types: string, XML element, or Base64 encoded binary. For example, if one wishes to submit a XML file, any option could be used:

PayloadOfString	<pre> <DataPayload> <PayloadOfString> <encoding>text/xml</encoding> <payload><![CDATA[<?xml version="1.0" encoding="UTF-8"?> <product> <name>pen</name> <price>\$1</price> </product>]]></payload> </PayloadOfString> </DataPayload> </pre>
PayloadOfBinary	<pre> <DataPayload> <PayloadOfBinary> <encoding>base64Binary</encoding> <compressed>>false</ compressed> <payload><![CDATA[67542893472937492]]></payload> </pre>

	</PayloadOfBinary> </DataPayload>
PayloadOfXmlElement	<DataPayload> <PayloadOfString> <encoding>xml</encoding> <payload> <product> <name>pen</name> <price>\$1</price> </product> </payload> </PayloadOfString> </DataPayload>

DataSchema and DataSchemaVersion

Four NEMESIS schemas are predefined. States/vendors could define custom schema used in data exchange. As of today, "NEMESIS EMS" and "NEMESIS Demographics" are required.

Combined with DataSchema, DataSchemaVersion helps to identify the exact schema used for the payload. For example, DataSchema="NEMESIS EMS" and DataSchemaVersion="2.2.1" is a valid combination. It points to current v2 schema. However, there is no version 2.5.6 for NEMESIS EMS data. So DataSchema="NEMESIS EMS" and DataSchemaVersion="2.5.6" is not a valid combination. This is particularly important for NEMESIS V3, since we allow each state to have state-specific schemas. We will develop a strategy and maintain a list of valid combinations: each of them should reference to one published standard (XSD).

SubmitDataReport

SubmitDataReport is for the report of data submission processing. It should include at least one XmlValidationErrorResponse element for the XML validation report. The implementation should follow these formats:

1. If the submission fails XML validation, SubmitDataReport should contain one XmlValidationErrorResponse. Since the submission is rejected at that step, SchematronReport should not be included.
2. If the submission passes XML validation, SubmitDataReport should contain
 - a. XmlValidationErrorResponse, with XmlValidationErrorResponse's TotalErrorCount set to zero.
 - b. SchematronReport, which contains either complete Schematron result file(s), or digested Schematron result file(s), or both of them. The decision is up to the state/vendor.
3. SubmitDataReport could include optional CustomReport element(s). It is designed to help the state/vendor handle their specific requirements.

XmlElementInfo

This is used to identify the offending XML element reported in XML validation or Schematron validation. Line/column numbers, or XPATH location, could be used to identify the position of an XML element. Usually, DOM parsers (validators) report XPATH information and SAX/StAX parsers (validators) report line/column numbers. Some special XML processors, like SAXON EE/PE version, can report both. The design is to allow line/column numbers and XPATH location to be reported together: but it is not required to do so. The design also allows position information to be “unknown”.

XmlValidationError

We expect two major types of XML validation errors. (For complete list of XML validation errors, check <http://svn.apache.org/viewvc/xerces/java/trunk/src/org/apache/xerces/impl/msg/>.) One could be pinpointed to a particular element: for example, if the element is defined as an integer but the value “ABC” is submitted. For this kind of error, we use XmlElementInfo to report the offending element. Another type of error is not focused on one element: for example, if a CSV file is submitted for XML validation. In this case, use XmlGeneralErrorList to include a list of error messages.

Main Functions

Communication in Web Services is defined by the request message and response message. For NEMSIS V3, we propose four common interfaces (functions): SubmitData, RetrieveStatus, QueryLimit and Search. Data structure for the request and response messages corresponding to these functions are defined.

Username, password, and organization are always required for any WS request. This information could be included in the request’s SOAP header, although we fail to find any advantage. For the proposed functions, values for element requestType are predefined. The state/vendor could develop other functions with a custom value for “requestType”. All functions also include an element of “additionalInfo” to allow for custom input.

NEMSIS WS response messages all include a status code (discussed above) and an echoing “requestType”, set to the same value as in the request. Except for the function of QueryLimit, they also include a unique identifier “requestHandle”: a system-assigned unique identifier at the server side for the transaction that takes place as a result of the request. In multiple query response situations, the “requestHandle” (or simply referred to as “handle”) is used as a common point of reference.

SubmitData

To submit data, the client needs to specify the data payload, data schema name, and schema version. As discussed above, the combination of schema name and version will decide the standard for the data submitted. After the data are submitted, it is subject to XML validation and Schematron rule checking. The submission could be rejected due to:

1. XML validation fails
2. National Schematron rule(s) violated

3. Critical state/vendor Schematron rule(s) violated
4. Other critical business rule(s) violated

The response for data submission could be synchronous or asynchronous: in the synchronous situation, an object of "SubmitDataReport" is included in the response, together with status code, handle, and echoing requestType. In the asynchronous situation, the server is not able to process the submitted data in time. Then "SubmitDataReport" is not included in the response. The client should use the assigned requestHandle in the response message to query the server later.

RetrieveStatus

RetrieveStatus function is used to retrieve results from the previous submission or search request. This is most important for an asynchronous response from the submission or search request. However, even with a synchronous response, the server might save the submission processing status or search result for a limited time period. In this case, if the client needs to retrieve the result later, it can use the returned requestHandle.

The response to RetrieveStatus could be:

1. If the process is still not finished (pending), the server should return the same requestHandle and status code of pending (0).
2. If the process is completed and status is available, the server should return the same requestHandle, proper status code, and a SubmitDataReport or SearchResult object, depending on the original request.
3. If the status is not available, either because the status has expired (e.g., the client wants to find the status for one search request from last year), or because the status has been deleted (to save space), or the requestHandle is not a valid identifier at all, the server should return with proper status code.

QueryLimit

Different web servers and WS implementations could apply a unique constraint on the size of the whole Web Service message. WS consumers can use this interface to query WS server's configuration for this limit.

The response to QueryLimit could be:

1. A positive integer to indicate the size limit on data payload, expressed in KB (1024 bytes).
2. A negative integer (-1) and an error status code.

Search

The proposed Search function is based on Utah's POLARIS project <http://health.utah.gov/ems/data/>). The POLARIS WSDL can be found at http://health.utah.gov/ems/web_services/PolarisWS.wsdl. = The most important tasks are to define data structures for search criteria and a returned result set.

There are several issues related to this model that we need to consider:

1. How to return a big dataset using WS if the search criteria are broad? For example, should a WS server really want to respond to a search which matches 1 million records? If so, should the server break the whole dataset into smaller chunks? Also, how long should the server keep the search result in an Asynchronous situation? It is not cheap to keep a big dataset around, in a database, or file system. To be frank, returning a big dataset will take too much time and resource use for the servers. Then, what about limiting the search result by rejecting a too broad search?
So far, due to the size of the dataset, it is not practical for NEMSIS TAC to implement this kind of dynamic and real-time search. Instead, we built a data warehouse and then present the data using Reporting Services and Cubes.
2. How useful is a Search function to all agencies/states? The previous issue could be resolved technically. However, from our discussion with various vendors and state managers, we don't think this feature could be required universally.
3. At the NEMSIS TAC, it is very possible we will not implement any Search functionality. We hope the community could help us clarify whether it is necessary to design this "Search" function. This is why we list the Search Function as "recommended".
4. Is it possible to define a common search result object? Using the full NEMSIS schema is one option. But it is overkill in many instances, since there are over 550 elements in the full NEMSIS version 3 schema. We put an element of "returnElementList" in the request message. Ideally, it contains a list of element names, delimited by semi-colon or coma. If the server finds any PCR events matching the search criteria, the returned PCR should only have those elements populated. Again, this is just an idea.
5. In POLARIS, the SearchParameters object includes three constraints to help further limit the search criteria. This is the explanation from POLARIS developers:
 - a. Complete: When a PCR has been marked complete by a user, and the PCR has passed all validation, it becomes complete. Otherwise, it is considered incomplete.
 - b. Modified: If a PCR was successfully marked complete, then later modified, the status is changed to modified.
 - c. Extended Search: In some cases, a patient will be cared for by more than one unit and by more than one agency. If a user/client has permission to do so, we allow them to see the PCRs filled out within 5 days of the incident for that same patient, even if they were filled out by a different agency.

Other vendors might want to put on similar search constraints. Is it necessary to create something similar to AdvancedSearch, namely "AdvancedConstraint", and have the structure modeled like this:

```
<xs:complexType name="AdvancedConstraint">
  <xs:sequence>
    <xs:element maxOccurs="1" minOccurs="1" name="ConstraintName" type="xs:string" />
    <xs:element maxOccurs="1" minOccurs="1" name="Operator" type="tns:Operator" />
    <xs:element maxOccurs="1" minOccurs="1" name="ConstraintValue" type="xs:string" />
    <xs:element maxOccurs="1" minOccurs="1" name="AndOr" type="tns:AndOr" />
  </xs:sequence>
</xs:complexType>
```

Use cases:

Case 1 – Submit Data Request, Synchronous scenario

Step 1. Agency “Elmo” wants to send a set of EMS records to state “Sesame Street”. A Submit Request is sent with “Request Type” = “SubmitData”. The SOAP message looks like this:

SOAP Message 1

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:ws="http://ws.nemsis.org/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:SubmitDataRequest>
      <ws:username>sutest</ws:username>
      <ws:password>openthedoor</ws:password>
      <ws:organization>sesame</ws:organization>

      <ws:requestType>SubmitData</ws:requestType>
      <ws:SubmitPayload>
        <!--You have a CHOICE of the next 3 items at this level-->
        <!--
          <ws:PayloadOfString>
            <ws:encoding>text/xml</ws:encoding>
            <ws:payload?</ws:payload>
          </ws:PayloadOfString>
        -->
        <ws:PayloadOfBinary>
          <ws:encoding>base64Binary</ws:encoding>
          <ws:compressed>true</ws:compressed>
          <ws:payload>cid:103713444083</ws:payload>
        </ws:PayloadOfBinary>
        <!--
          <ws:PayloadOfXmlElement>
            <ws:encoding>xml</ws:encoding>-->
            <!--You may enter ANY elements at this point-->
          <!--
            </ws:PayloadOfXmlElement>
          -->
        </ws:SubmitPayload>
        <ws:requestDataSchema>bbbb</ws:requestDataSchema>
        <ws:schemaVersion>3456</ws:schemaVersion>
        <ws:additionalInfo>fun</ws:additionalInfo>
      </ws:SubmitDataRequest>
    </soapenv:Body>
  </soapenv:Envelope>
```

Step 2. The State receives the message and successfully processes it within a reasonable amount of time. It replies with a “requestHandle”, a system-assigned unique identifier for the transaction of handling Elmo’s WS request. Then the response message looks like this:

SOAP Message 2

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <SubmitDataResponse xmlns="http://ws.nemsis.org/">
      <requestType>SubmitData</requestType>
      <requestHandle>12345678</requestHandle>
      <statusCode>1</statusCode>
      <reports>
        <XmlValidationErrorReport>
          <TotalErrorCount>0</TotalErrorCount>
        </XmlValidationErrorReport>
        <SchematronReport>
          <CompleteSchematronReport>
            <CompleteReport>
              <PayloadOfString>
                <payload>test string payload. should be XML file content.</payload>
              </PayloadOfString>
            </CompleteReport>
          </CompleteSchematronReport>
        </SchematronReport>
      </reports>
    </SubmitDataResponse>
  </S:Body>
</S:Envelope>
```

Case 2. Submit Data Request, Asynchronous scenario

Step 1. Same as step 1 in case 1, Agency “Elmo” sends XML to state “Sesame Street”.

Step 2. The State receives the xml file. Because the system is too busy handling other agencies’ requests, the response message looks like this:

SOAP Message 3

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <SubmitDataResponse xmlns="http://ws.nemsis.org/">
      <requestType>SubmitData</requestType>
      <requestHandle>12345678</requestHandle>
      <statusCode>0</statusCode>
    </SubmitDataResponse>
  </S:Body>
</S:Envelope>
```

Step 3. RetrieveStatus Request for previous submission

After 15 minutes, as a responsible agent, Elmo thinks it has given the state system enough time to process the data. Using “requestHandle”, Elmo can query the state system to get the result of its previous WS request. The SOAP message for this kind of “RetrieveStatus” request looks like this:

SOAP Message 4

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:ws="http://ws.nemsis.org/">
  <soapenv:Header/>
  <soapenv:Body>
    <ws:RetrieveStatusRequest>
      <ws:username>sutest</ws:username>
      <ws:password>openthedoor</ws:password>
      <ws:organization>sesame</ws:organization>
      <ws:requestType>RetrieveStatus</ws:requestType>
      <ws:requestHandle>12345678</ws:requestHandle>
      <!--Optional:-->
      <ws:originalRequestType>SubmitData</ws:originalRequestType>
      <ws:additionalInfo>fun</ws:additionalInfo>
    </ws:RetrieveStatusRequest>
  </soapenv:Body>
</soapenv:Envelope>
```

Step 4. Response to Step 3’s Request

Output 1: If the state has finished processing the submission request, then the response message looks like “SOAP Message 2”.

Output 2: If the state still hasn’t finished processing the submission request, the response message looks like “SOAP Message 3”. So Elmo needs to come back later to check the status (repeat step 3).

Note: if for any reason Elmo forgets the status of his submission, even after he has received notification of a successful submission, Elmo should be able to send a “RetrieveStatus” request to the state again. As a good bookkeeper, the state should be able to response back with “SOAP Message 2”. Certainly, if the requestHandle is actually for a submission 10 years ago, it is very possible that the state will tell Elmo the status has expired.

Discussions:

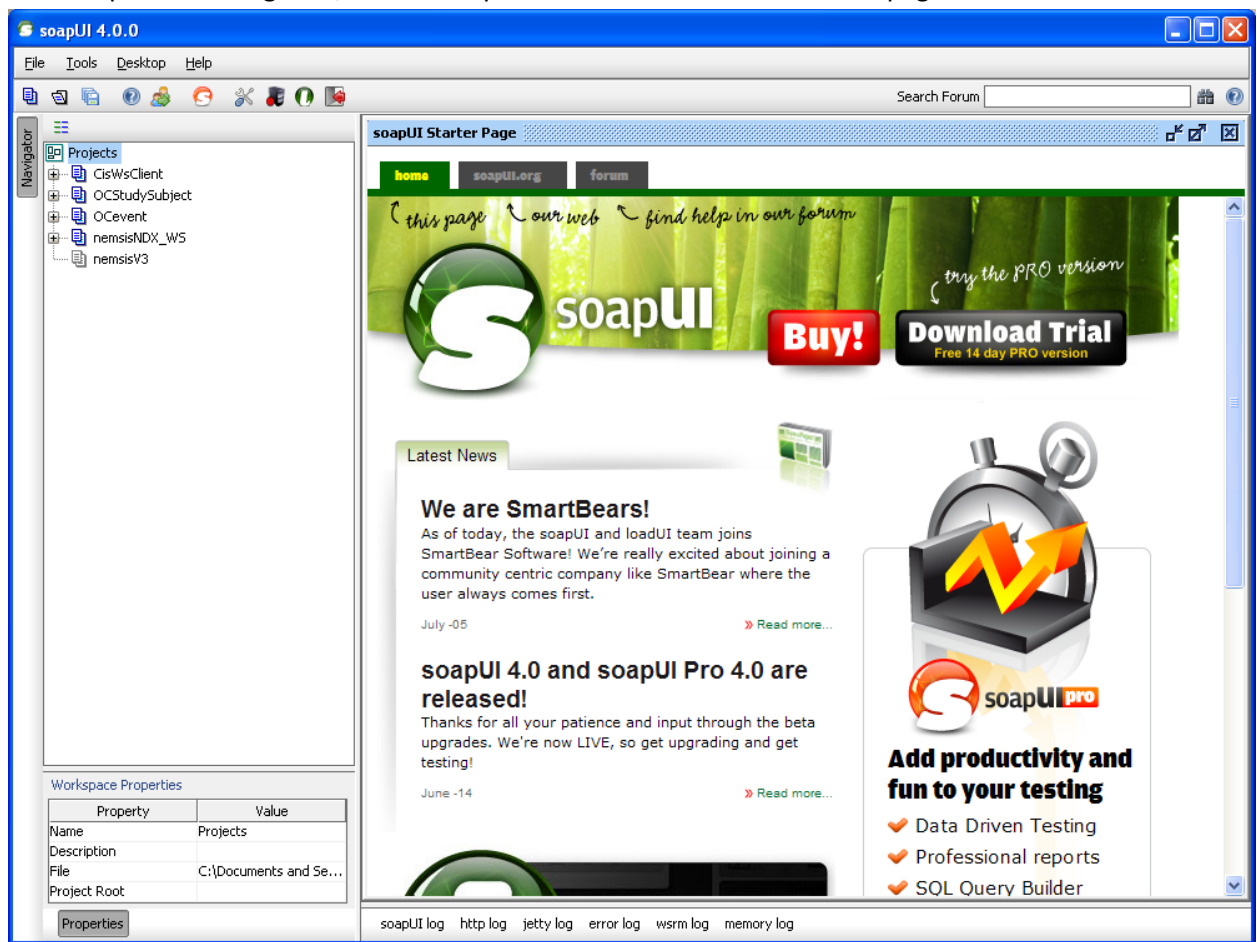
1. Authentication/Security Implementation
 - a. Some vendors might prefer to utilize the security element in the SOAP message's header. Web Service Security Specification, published by OASIS (<http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SOAPMessageSecurity.pdf>), provides a set of mechanisms to enforce message integrity and confidentiality. However, due to the vast differences among EMS vendors/deployments, the NEMSIS TAC will not specify a particular SOAP security header structure. It is very possible, and legitimate, that a vendor may choose to include the authentication information in the SOAP header. In such instances, both ends of the WS communication have to reach a mutual understanding about the underlining security mechanism (and ignore the username/password parameters in the proposed object nemsisV3WsRequest). For example, they can choose to use (a)symmetric keys to encrypt username/password.
 - b. It was stated in the draft NEMSIS WS Guide that "passwords should never be sent as clear-text". We should revise this statement to be "passwords should never be sent as clear-text over a non-secured channel". Again, we require that all WS communications use HTTPS since we need to protect not only the header section, but also the embedded data payload. Hashing, or password digestion, is not more secure than a password in clear text, if it is not sent on a secure channel or token not encrypted. (See <http://www.oasis-open.org/committees/download.php/16782/wss-v1.1-spec-os-UsernameTokenProfile.pdf>, line 159-162. On 10/19/2011, some researchers demonstrated at the ACM conference how to break W3C standard XML encryption.)
 - c. Password hashing might create unique challenges for implementation. For example, when using Microsoft Active Directory (AD) authentication, the WS recipient has to have a way to figure out the clear text password, then it can be sent to AD for authentication. OASIS documents have detailed discussions about other security concerns. (See references on page 27.)
2. Web Services is not a perfect solution for transmitting large amounts of data in one WS message. Most web servers and WS implementations apply a limit on the size of one Web Service message. Oversized WS messages could be rejected by a web server, even before reaching the WS receiving codes. Also, due to performance considerations, WS recipients might prefer small WS messages. Researchers at IBM have published a study on this issue. (See reference 4 on page 27) We strongly encourage software vendors to develop appropriate strategies that fit their own software/hardware environments.
3. Unfortunately, it is hard to predict the size of one NEMSIS EMS record, especially when medical image files might be embedded. This is why the proposed interface "QueryLimit" should return the limit on the size of the string, instead of how many EMS records are contained.

4. Many web servers and web clients (like most web browsers) have built-in HTTP compression capability to facilitate faster transmission speeds. But, in the case where the WS client/server doesn't have HTTP compression built-in, it is necessary to consider compressing the NEMSIS XML File. In most cases, NEMSIS XML files could be compressed easily. We often see the compression rates of 30:1. So a 30MB NEMSIS file could be compressed to 1MB. After Base64 encoding, which increases the size of binary data by 33%, the final string to transmit is about 1.4MB.

A Quick Guide for using soapUI

This is just a simple guide to use soapUI, one of the most popular open source applications for testing Web Services. I believe that most readers are already familiar with it. Feel free to skip this section. The following is for Windows users. Check <http://www.soapui.org/Getting-Started/getting-started.html> for additional information.

1. Download soapUI from <http://www.soapui.org/>. (I prefer to download the standalone version.)
2. Install soapUI to your system, if you download the installer. If you download the standalone version, simply unzip it to your hard drive; then go into its bin folder, and run “soapui.bat” to start soapUI.
3. Start soapUI from Programs, or run “soapui.bat”. You should see a starter page like this:

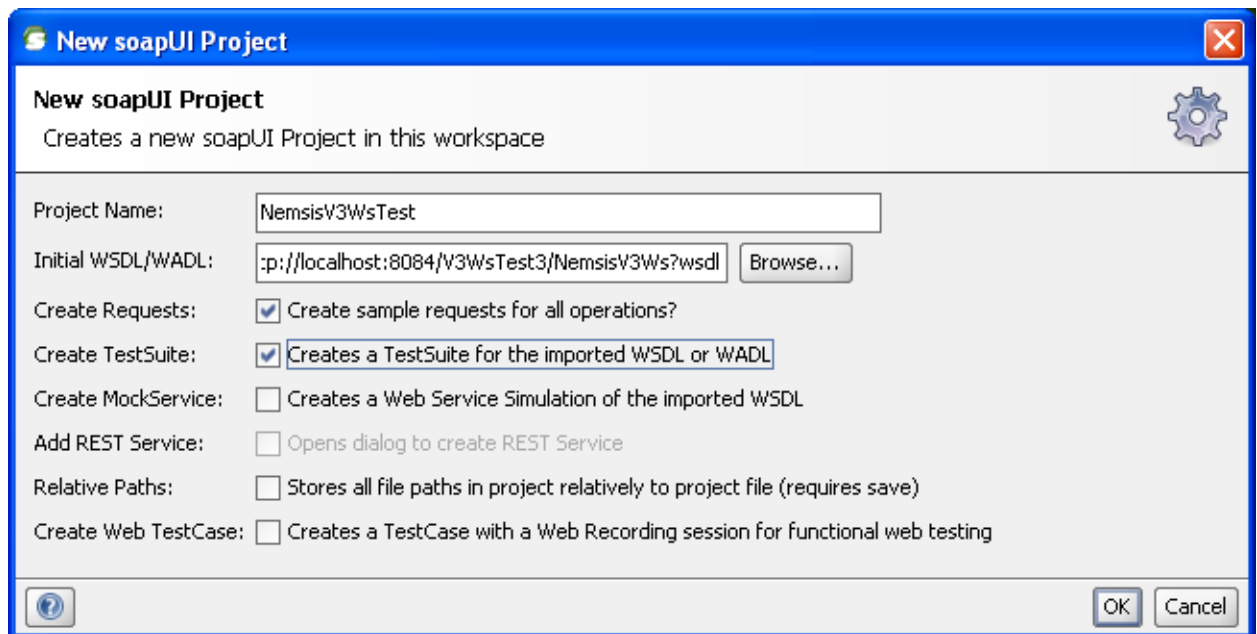


4. Click on File→"new soapUI Project" to start a new project.



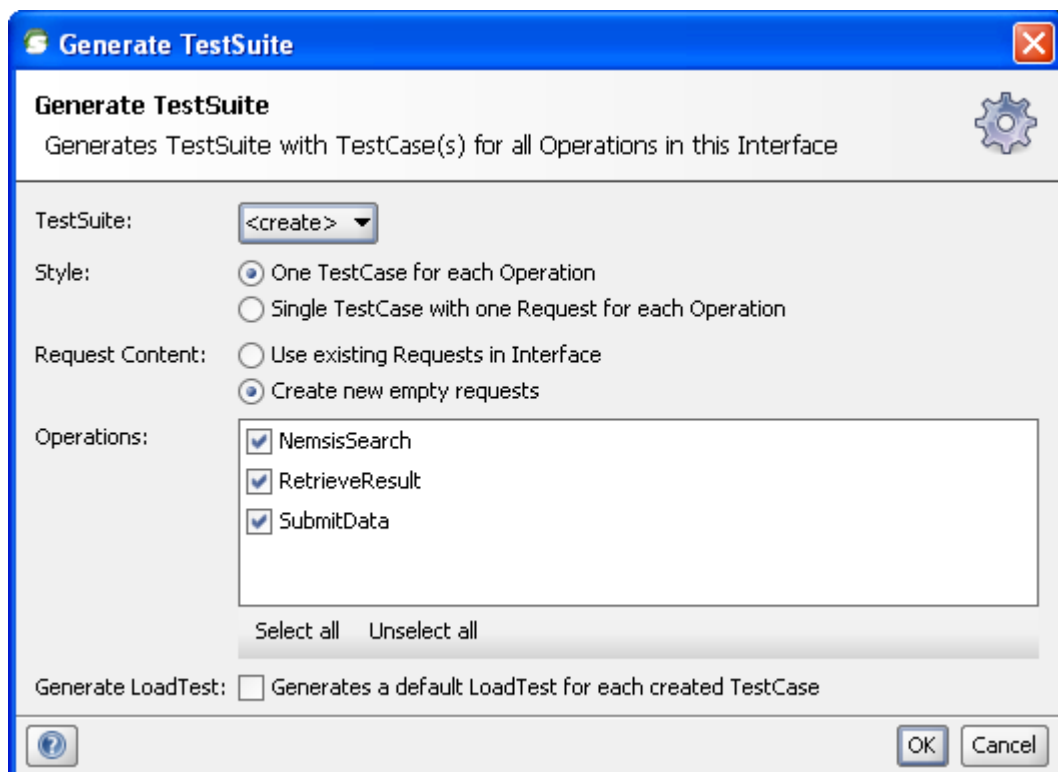
5. Name your project. Since I deployed my NEMESIS WS on my developing desktop, the WSDL could be accessed at <http://localhost:8084/V3WsTest3/NemesisV3Ws?wsdl>. Copy this to the text box next to "Initial WSDL/WADL". In many cases, you might save the published WSDL file to your hard drive – then you can use the Browse button to locate the WSDL file. It would be wise to investigate the WSDL file first: you have to make sure all referenced files are accessible. A funny thing about a WSDL file published by Spring Framework is that it usually doesn't include the full path in the SOAP address. For example, the line
- ```
<soap:address location="http://localhost:8084/V3WsTest3/NemesisV3Ws" />
```
- could simply be
- ```
<soap:address location=" /V3WsTest3/NemesisV3Ws" />
```
- In that case, you'd better save the WSDL and edit it. (You can also change the soapUI test property to force it point to the real server address.)

Remember to create TestSuite.



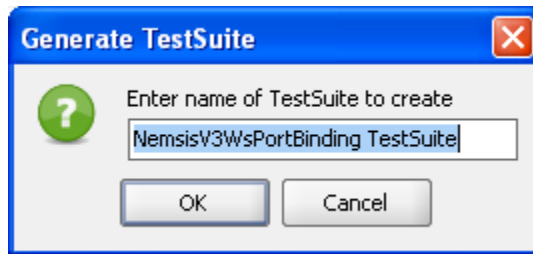
The 'New soapUI Project' dialog box is shown. It has a title bar with a question mark icon, the text 'New soapUI Project', and a close button. Below the title bar is a subtitle 'New soapUI Project' and a description 'Creates a new soapUI Project in this workspace' next to a gear icon. The main area contains several fields and checkboxes: 'Project Name:' with a text box containing 'NemsisV3WsTest'; 'Initial WSDL/WADL:' with a text box containing 'p://localhost:8084/V3WsTest3/NemsisV3Ws?wsdl' and a 'Browse...' button; 'Create Requests:' with a checked checkbox and label 'Create sample requests for all operations?'; 'Create TestSuite:' with a checked checkbox and label 'Creates a TestSuite for the imported WSDL or WADL'; 'Create MockService:' with an unchecked checkbox and label 'Creates a Web Service Simulation of the imported WSDL'; 'Add REST Service:' with an unchecked checkbox and label 'Opens dialog to create REST Service'; 'Relative Paths:' with an unchecked checkbox and label 'Stores all file paths in project relatively to project file (requires save)'; and 'Create Web TestCase:' with an unchecked checkbox and label 'Creates a TestCase with a Web Recording session for functional web testing'. At the bottom are a help icon, an 'OK' button, and a 'Cancel' button.

6. Now, soapUI will parse the WSDL and generate TestSuite for all functions defined in WSDL file.

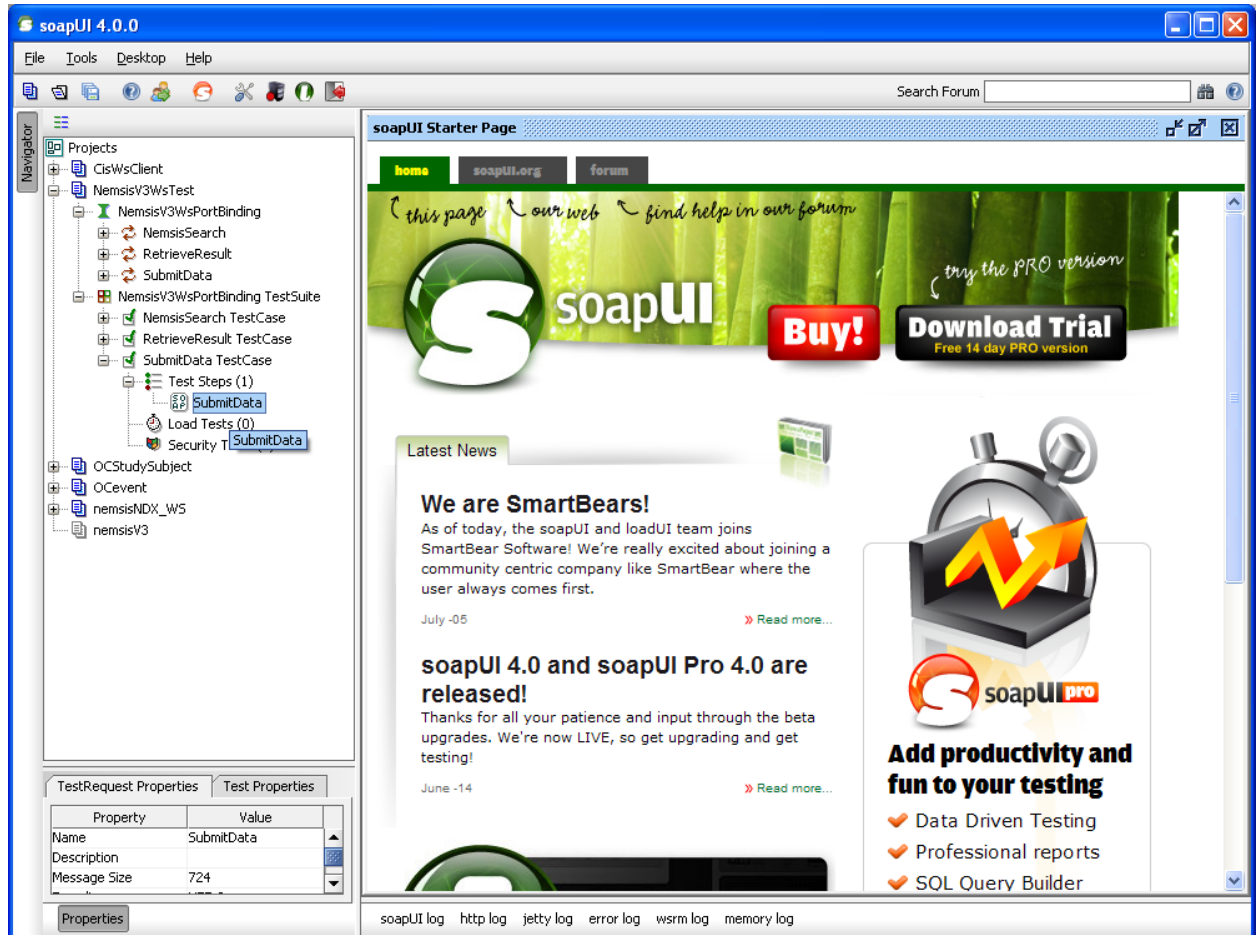


The 'Generate TestSuite' dialog box is shown. It has a title bar with a question mark icon, the text 'Generate TestSuite', and a close button. Below the title bar is a subtitle 'Generate TestSuite' and a description 'Generates TestSuite with TestCase(s) for all Operations in this Interface' next to a gear icon. The main area contains several fields and checkboxes: 'TestSuite:' with a dropdown menu showing '<create>'; 'Style:' with two radio buttons, 'One TestCase for each Operation' (selected) and 'Single TestCase with one Request for each Operation'; 'Request Content:' with two radio buttons, 'Use existing Requests in Interface' and 'Create new empty requests' (selected); 'Operations:' with a list box containing three items: 'NemsisSearch', 'RetrieveResult', and 'SubmitData', each with a checked checkbox; 'Select all' and 'Unselect all' buttons below the list box; and 'Generate LoadTest:' with an unchecked checkbox and label 'Generates a default LoadTest for each created TestCase'. At the bottom are a help icon, an 'OK' button, and a 'Cancel' button.

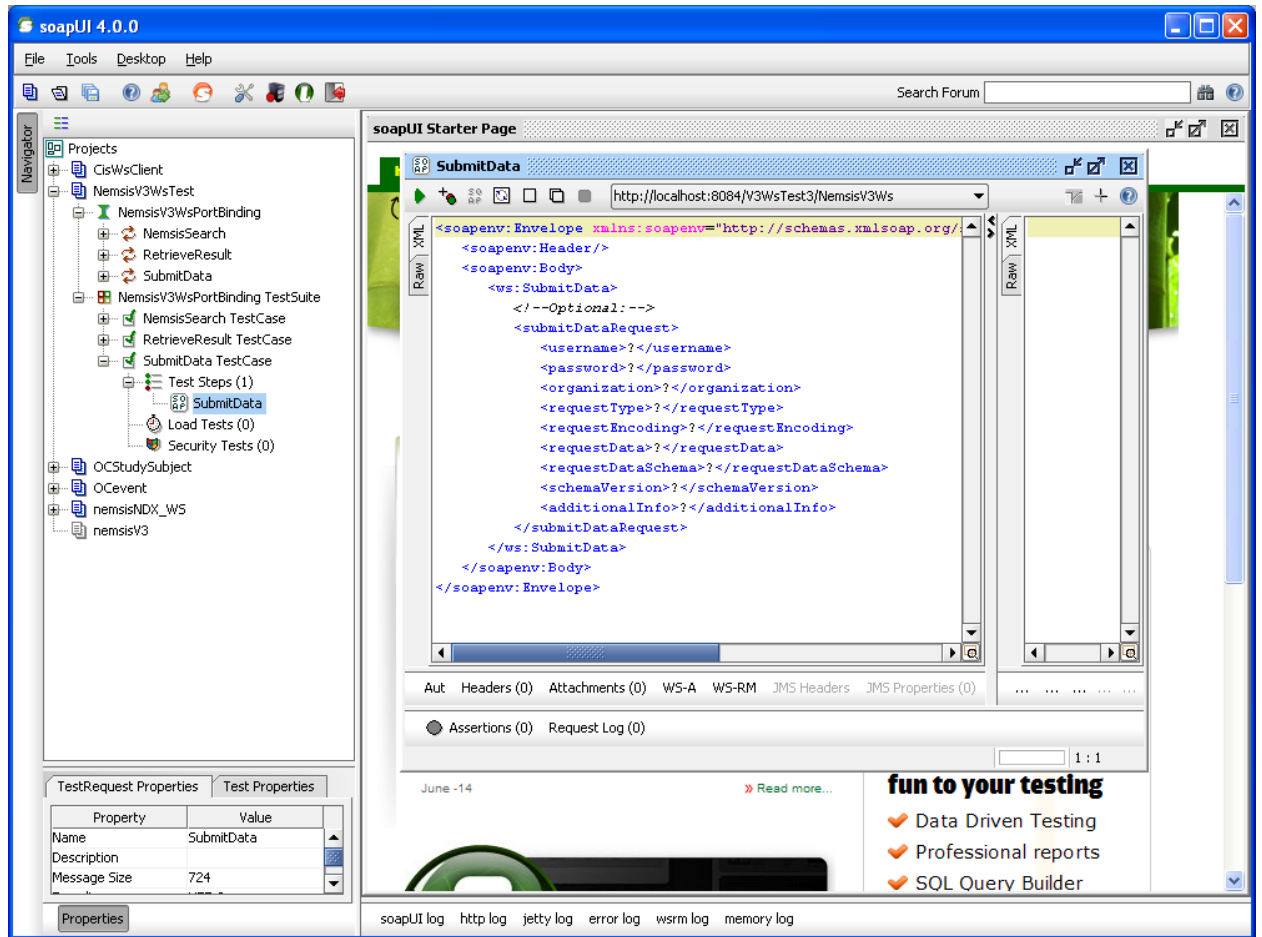
7. Give the TestSuite a name



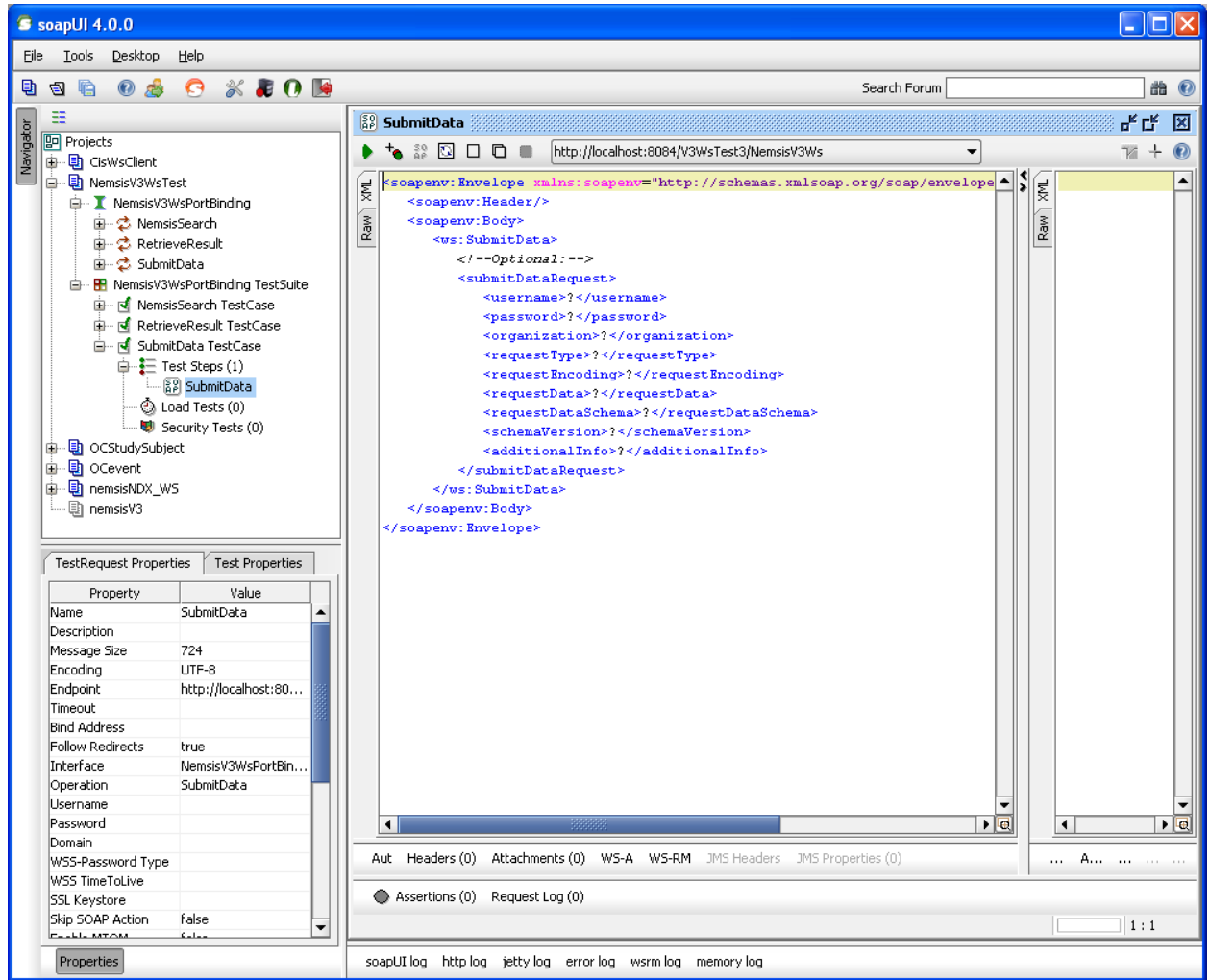
8. In project NemsisV3WsTest, go into “NemsisV3WsPortBinding TestSuite”, “SubmitData TestCase”, “Test Steps(1)”, “SubmitData”:



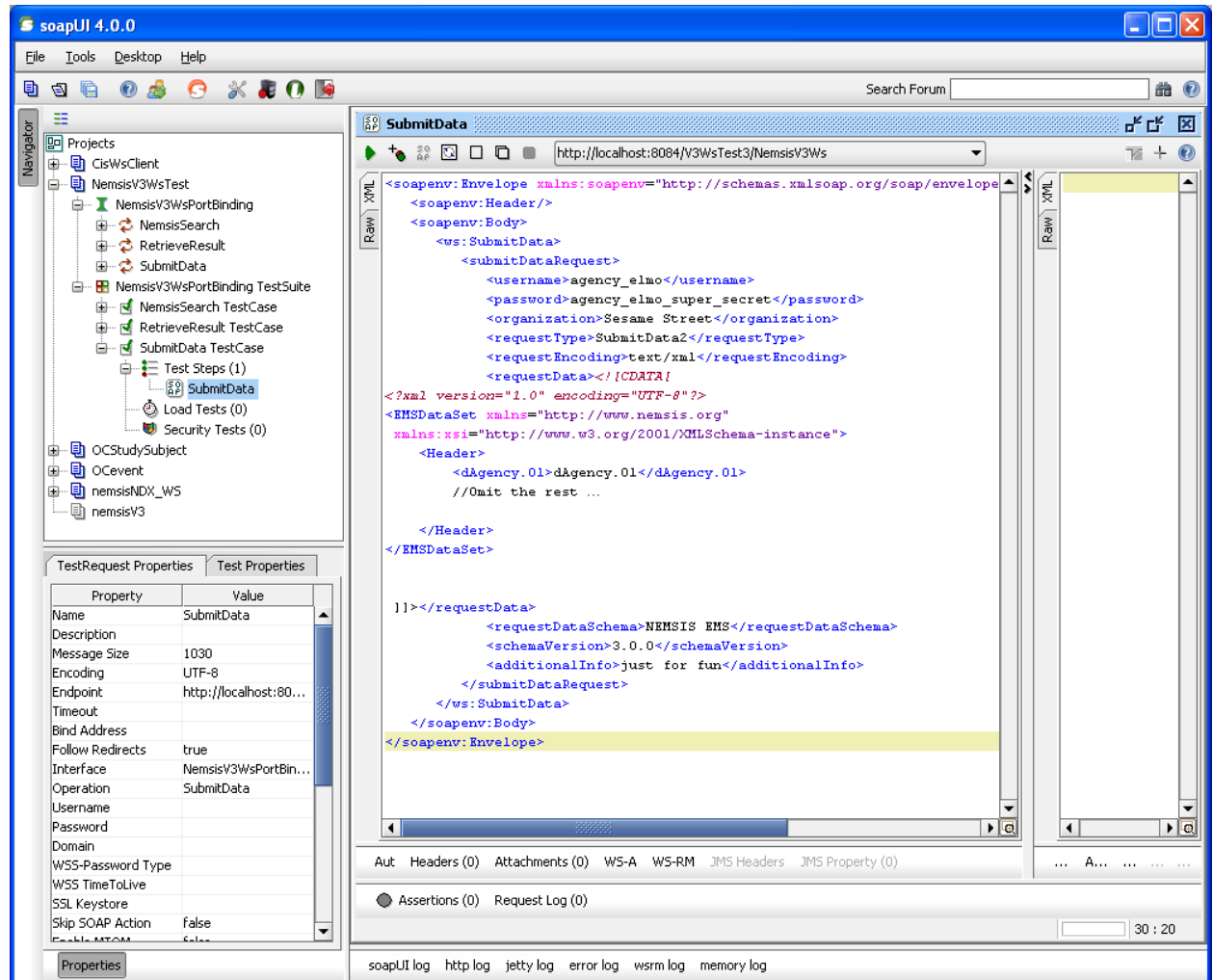
Double click on “SubmitData”,



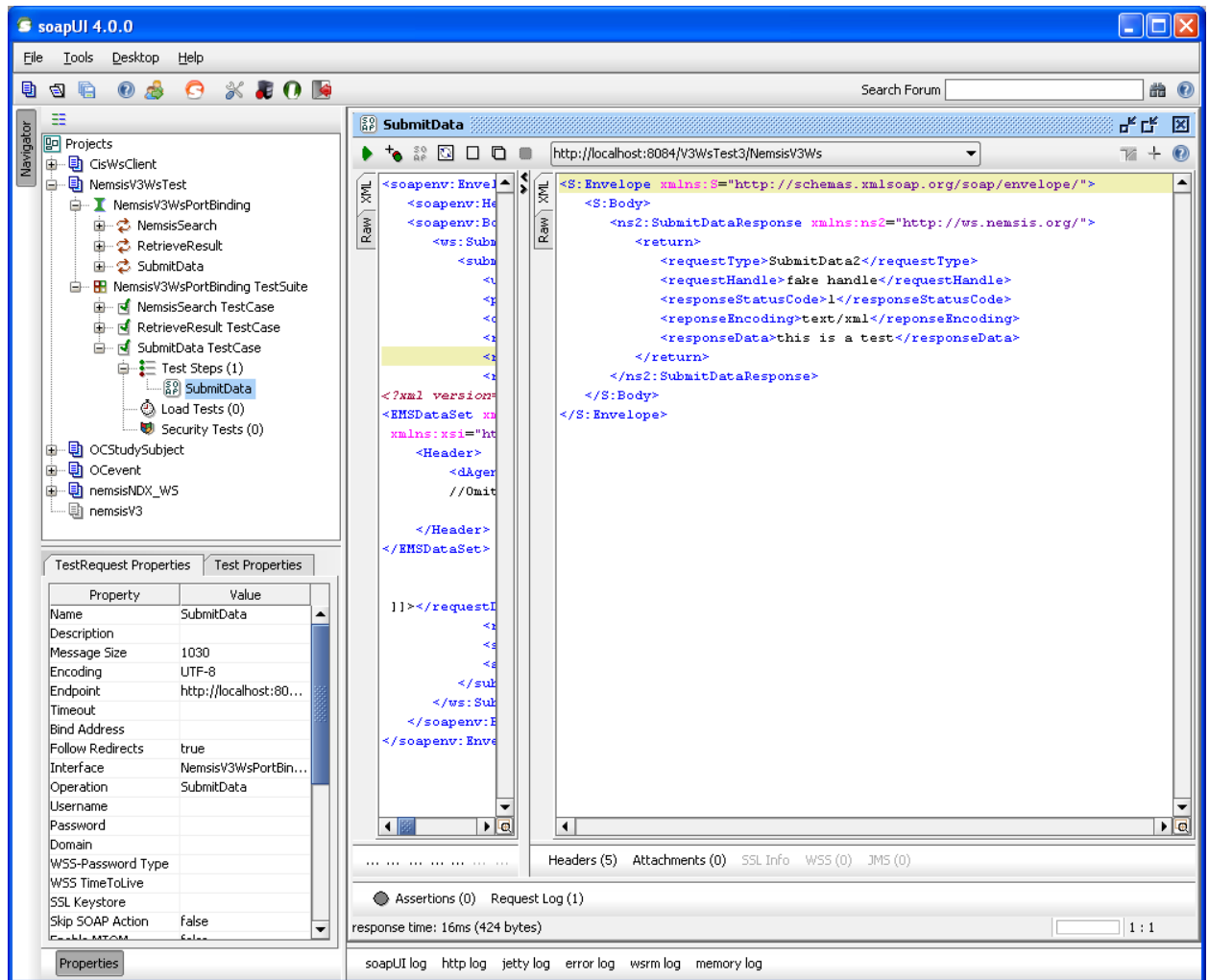
Now is a good time to close “Starter Page” and maximize the new “SubmitData” window.



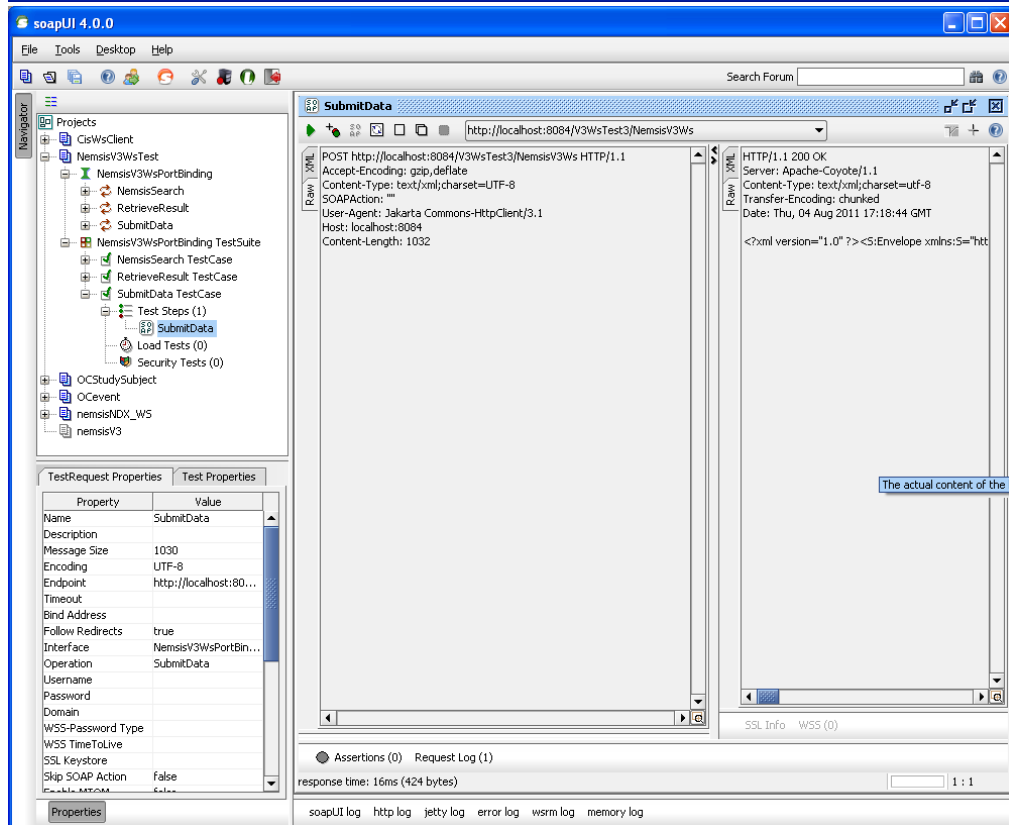
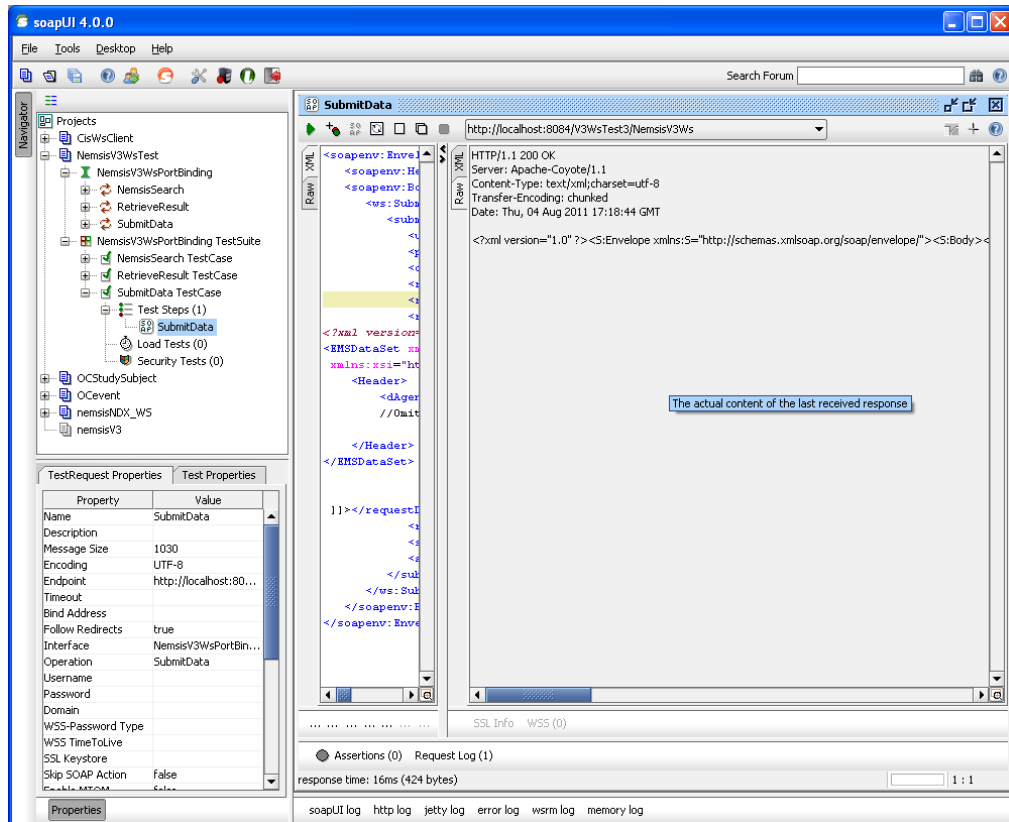
9. SoapUI has already created a template for you to fill in real data. You can copy and paste the content of “SOAP Message 1” in this file’s “User Cases” section.

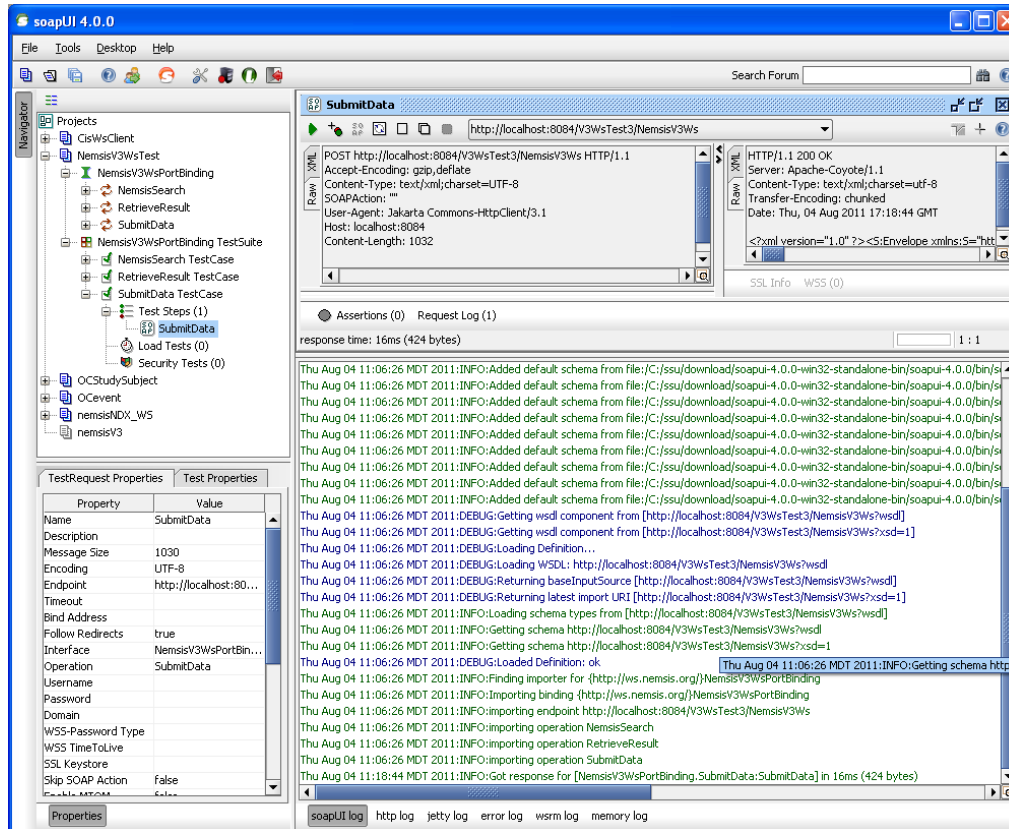


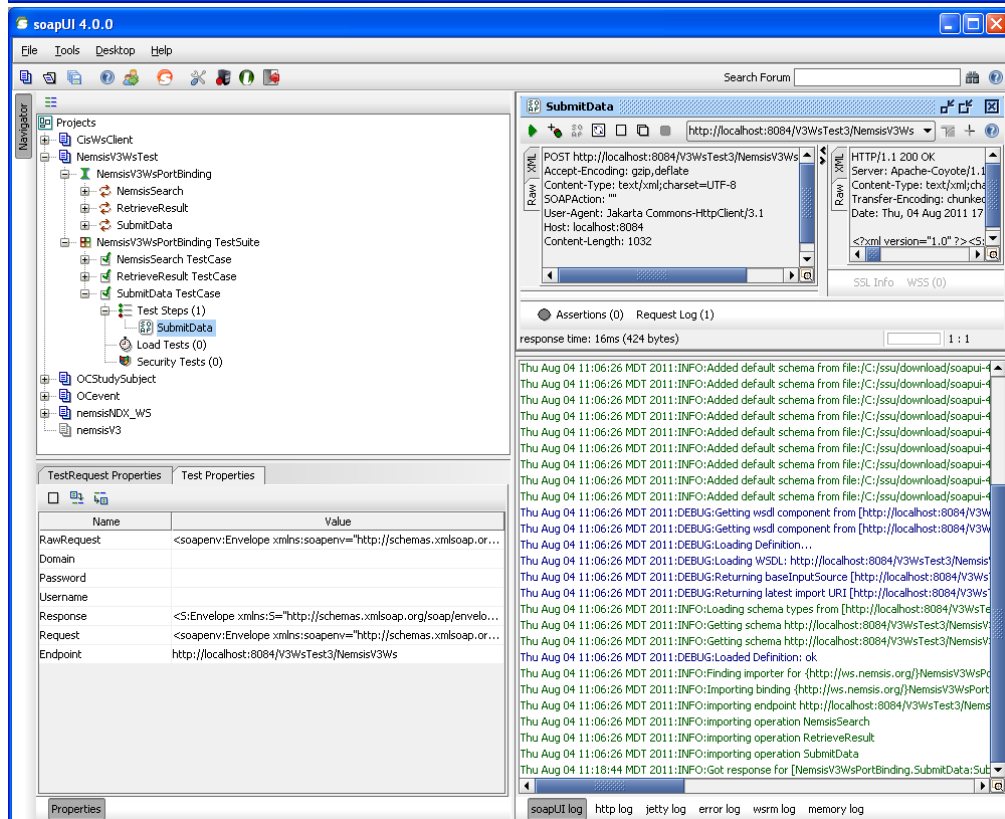
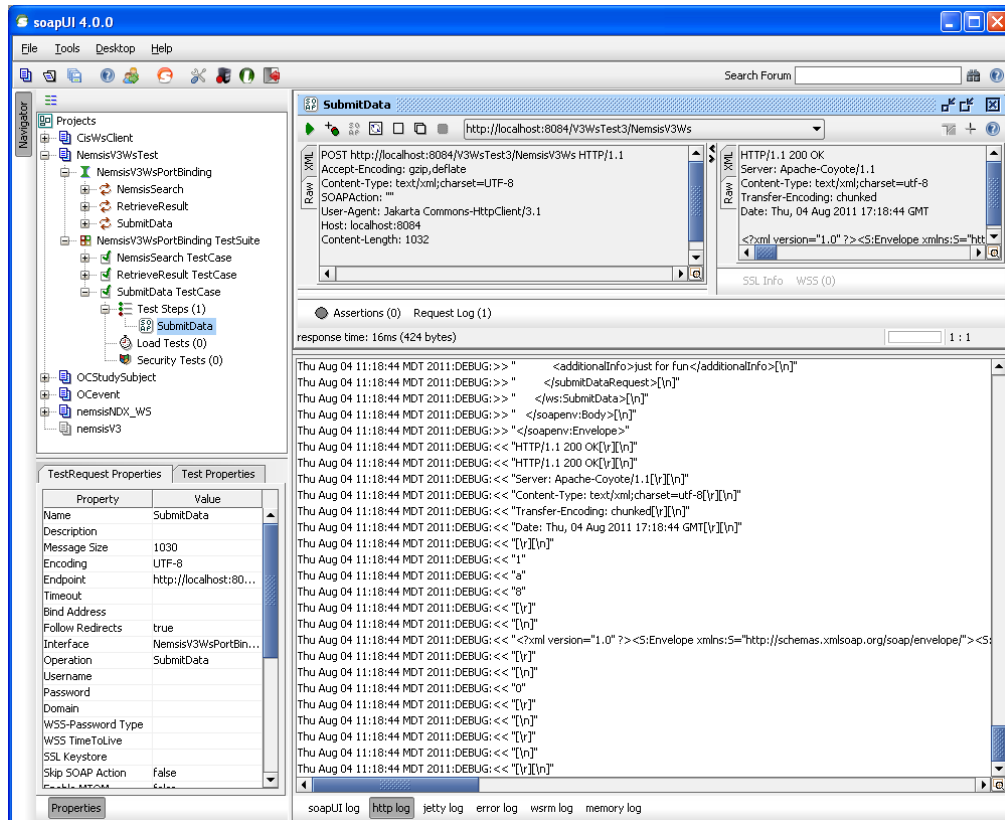
10. Magic moment: click on the green triangle button in left-upper corner of the “SubmitData” window. This will submit the SOAP message to the designated WS server (in Web Services terms, this is the “endpoint”. Make sure your WS server is up and running!). You can see the result in the right panel: since we are testing, my super-simple WS server returns a super-simple message.

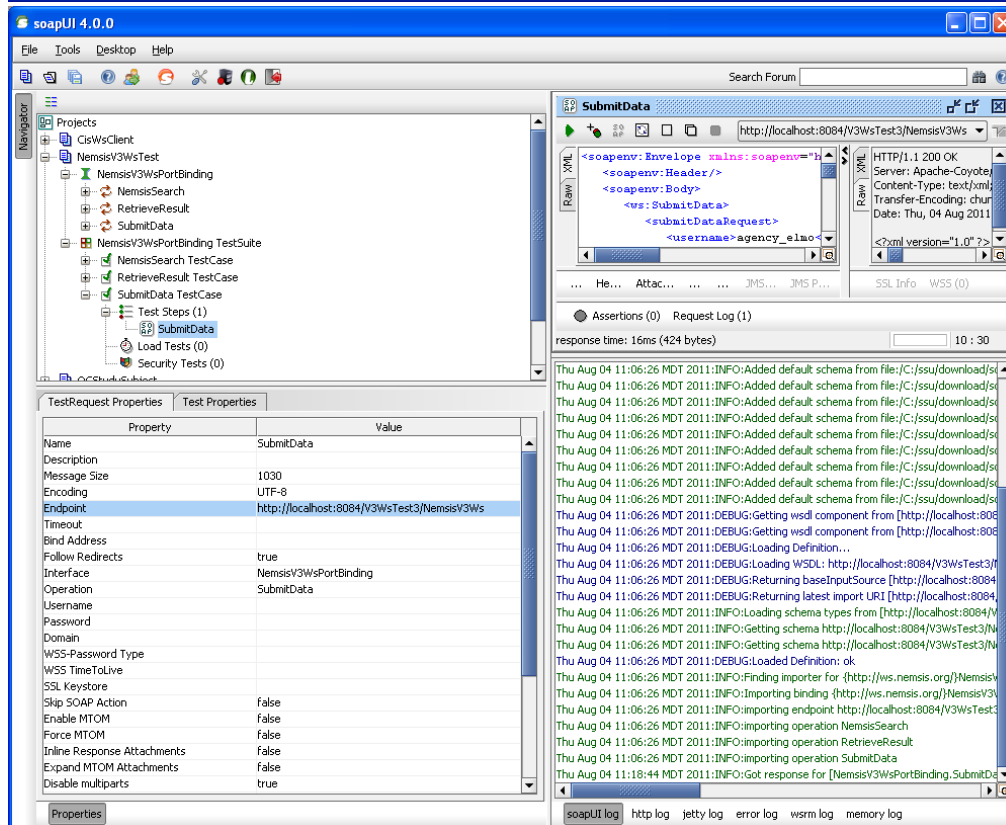


11. SoapUI provides a wide range of functionalities to help you debug a WS application. For example, you can check the raw request/response message, error log, http log. And you can change your endpoint, set up security, attach documents, etc. I attached some snapshots to provide further guidance. To explore the full power of soapUI, read its online document and the forum.









Reference:

1. WS-SecurityPolicy 1.2. OASIS Standard, 1 July 2007. <http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.pdf>
2. Web Services Security: SOAP Message Security 1.1. OASIS Standard Specification, 1 February 2006. <http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SOAPMessageSecurity.pdf>
3. Web Services Security: UsernameToken Profile 1.1. OASIS Standard Specification, 1 February 2006. <http://www.oasis-open.org/committees/download.php/16782/wss-v1.1-spec-os-UsernameTokenProfile.pdf>
4. SOAP Message Size Performance Considerations. <http://www.redbooks.ibm.com/abstracts/redp4344.html>