Knowarc_D0.0-0_00

8/9/2008

ARC1 Python Web Services Developer's Guide

Tamás Kazinczy[1]

[1] kazy@niif.hu

Contents

List of Figures

1 Preface

1.1 Purpose of this document

This document has two main objectives. On one hand it guides you through the creation of a new Python-based Web Service in ARC1[1]. On the other hand it describes the steps needed to test the newly-created ARC1 Service.

The document has the following structure:

- In the first section we provide a brief description about the intended audience and the service to be implemented.
- In the next section Service Design will be discussed.
- In section Implementing the Service we provide an overview of the Python implementation.
- In section Putting GreetingService to work we describe the steps needed to make the Python service work.
- In section Testing the Service we introduce a tool to test the service with and the steps of testing.

1.2 Intended audience

ARC1 Python Web Services Developer's Guide is intended for developers, who:

- have some experience with Python;
- want to create Python-based services in ARC1;
- are familiar with XML[2], XSD[3], SOAP[4] and WSDL[5].

1.3 About the service to be implemented

The service (GreetingService) to be implemented is quite simple. Users send their names to the service in a message and, as an answer, they get a personal greeting that tells them the local time on the server. So if the name of the user is Anonymous, the answer would be something like: 'Hello, Anonymous! Local time is: [Mon Aug 11 14:17:53 2008]'.

[1]See `http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/README` and `http://www.knowarc.eu/`
[2]See `http://www.w3.org/XML`
[3]See `http://www.w3.org/XML/Schema`
[4]See `http://www.w3.org/TR/soap/`
[5]See `http://www.w3.org/TR/wsdl` and `http://www.w3.org/TR/wsdl20`

2 Service Design

For the sake of clarity and the simplicity of testing, we use a top-down approach in the service design. First, we create an XML Schema Definition (XSD), then use this XSD to construct the WSDL of the service.

2.1 Designing the XML Schema Definition (XSD)

The GreetingService is quite simple: it has a single input and a single output. As an input, it takes a full personal name; as an output, it sends a greeting message in the following form: Hello, XY! Local time is: [SERVER-DATE-AND-TIME].

In designing the service first, we provide a targetNamespace for the schema. Let it be, for example `http://example.com/GreetingService/types`. We also define a namespace prefix `tns` for this namespace so that we can refer to the types defined in this schema.

```
<schema
  targetNamespace="http://example.com/GreetingService/types"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://example.com/GreetingService/types"
  elementFormDefault="qualified">
```

Illustration 1: Definition of target namespace

Next, we create simple types for the input (name) and output (greeting).

```
<simpleType name="name">
  <restriction base="xsd:string"/>
</simpleType>
<simpleType name="greeting">
  <restriction base="xsd:string"/>
</simpleType>
```

Illustration 2: Simple types for input (name) and output (greeting)

GreetingService is of a request-response type, so we create types for the request and response as well.

```
<complexType name="GreetingServiceRequestType">
  <sequence>
    <element name="yourName" type="tns:name" minOccurs="1" maxOccurs="1"/>
  </sequence>
</complexType>
```

Illustration 3: Complex type to represent request

```
<complexType name="GreetingServiceResponseType">
  <sequence>
    <element name="yourGreeting" type="tns:greeting" minOccurs="1" maxOccurs="1"/>
  </sequence>
</complexType>
```

Illustration 4: Complex type to represent response

Here we have used the previously created simple types (referred to as `tns:name` and `tns:greeting`) to define the elements `yourName` and `yourGreeting` of complex types that will represent the request and response.

All we need to do now is to create the request and response elements which are of complex types mentioned before.

```
<element name="GreetingServiceRequest" type="tns:GreetingServiceRequestType"/>
<element name="GreetingServiceResponse" type="tns:GreetingServiceResponseType"/>
```

Illustration 5: Request and response element

The complete XSD can be found in Appendix A.1 of this guide.

2.2 Designing the Service WSDL

Construction of the service WSDL is relatively straightforward.

We use those elements defined in our XML schema so we have to provide a namespace prefix for the namespace `http://example.com/GreetingService/types` to be able to refer to them. We also define the target namespace (`http://example.com/GreetingService`) as well as a prefix for our service definition.

```
<definitions
  targetNamespace="http://example.com/GreetingService"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:gs="http://example.com/GreetingService"
  xmlns:gst="http://example.com/GreetingService/types">
```

Illustration 6: Namespaces

We also need to import our XML schema in the types section. It is worth noting that (according to the WS-I Basic Profile Version 1.1): “To import XML Schema Definitions, a DESCRIPTION MUST use the XML Schema `import` statement.”

```
<wsdl:types>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:import namespace="http://example.com/GreetingService/types"
                schemaLocation="GreetingService.xsd"/>
  </xsd:schema>
</wsdl:types>
```

Illustration 7: Types

Then, we create the messages to be used by the service. These messages will use the request and response elements that we created earlier in the schema.

```
<wsdl:message name="GreetingServiceRequestMessage">
  <wsdl:part name="payload" element="gst:GreetingServiceRequest"/>
</wsdl:message>
<wsdl:message name="GreetingServiceResponseMessage">
  <wsdl:part name="payload" element="gst:GreetingServiceResponse"/>
</wsdl:message>
```

Illustration 8: Definition of WSDL messages

The next step is to create a portType. Here we define the operation (greet) along with its input and output messages.

```
<wsdl:portType name="GreetingServicePT">
  <wsdl:operation name="greet">
    <wsdl:input name="GreetingServiceInput"
                message="gs:GreetingServiceRequestMessage"/>
    <wsdl:output name="GreetingServiceOutput"
                 message="gs:GreetingServiceResponseMessage"/>
  </wsdl:operation>
</wsdl:portType>
```

Illustration 9: Definition of WSDL portType

Then we define the binding where we use the portType created previously. The key point here is that we use `document/literal` style SOAP HTTP binding.

```
<wsdl:binding name="GreetingServiceBinding" type="gs:GreetingServicePT">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="greet">
    <soap:operation soapAction="greet"/>
    <wsdl:input name="GreetingServiceInput">
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output name="GreetingServiceOutput">
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
```

Illustration 10: Definition of WSDL binding

As the last step of the design, we define the service with a port that uses this binding. For the sake of simplicity, we should provide an endpoint address where our service will listen.

```
<wsdl:service name="GreetingService">
  <wsdl:port name="GreetingServicePort" binding="gs:GreetingServiceBinding">
    <soap:address location="http://arctest.ki.iif.hu:50000/GreetingService"/>
  </wsdl:port>
</wsdl:service>
```

Illustration 11: Definition of WSDL service

The complete WSDL can be found in Appendix A.2 of this guide.

3 Implementing the Service

3.1 Before jumping in

Though configuration is discussed later in this document, some points need prior explanation.

For demonstration purposes, fixed parts of the greeting of GreetingService are stored in ARC HED's[1] XML configuration file. In the implementation we will get these parts from the XML right after the instantiation of the Python class (in the `__init__()` method).

3.2 Overview of the Python implementation

As it was pointed out before, we get fixed parts of the greeting in the `__init__()` method. This also includes resolving the namespace prefix of the service namespace `http://example.com/GreetingService`.

In ARC1, Python-based services use a wrapper to access the core service classes and their methods. This means that upon starting ARC HED, the wrapper loads the Python based services as modules. The nature of this wrapper indicates that every such (Python based) service is required to provide a process() method. In this method, we get the payload from the incoming SOAP message, get the name of the consumer, get the local time, then construct the greeting message, assemble the outgoing SOAP message and finally return.

The source code including comments can be found in Appendix A.3 of this guide.

[1]HED: Hosting Environment Daemon

4 Putting GreetingService to work

4.1 Creating `__init__.py`

Let us assume that our Python implementation (greetingservice.py) resides in a directory called GreetingService[1]. Without `__init__.py` in it, this is only a simple directory; it is `__init__.py` what makes GreetingService a Python package, so we need to have one, even if it is an empty file.

4.2 Creating `Makefile.am`

We need `Makefile.am` to generate the `Makefile`. It is rather simple in our example, consisting of only three lines:

```
pythondir= $(PYTHON_SITE_PACKAGES)/GreetingService
python_DATA = __init__.py greetingservice.py
EXTRA_DISTS = $(python_DATA)
```

Illustration 12: Contents of `Makefile.am`

The first line tells where GreetingService will be installed. Here `$(PYTHON_SITE_PACKAGES)` refers to something like `/usr/local/lib/python2.5/site-packages/`.

Second line tells what the Python sources are (here you see the Python source files separated by spaces) and the third one that we will use these files in our service.

4.3 Generating the `Makefile`

Open a terminal, go to ARC1 home directory (`$ARC1_HOME`), then edit `./configure.ac` (you may need root privileges to do this):

1. search for the text `AC_CONFIG_FILES`;
2. put an extra path there for the `GreetingService Makefile` (this is required to be the relative path from `$ARC1_HOME`, so if the `GreetingService` directory was in `$ARC1_HOME/src/services` then it should be `src/services/GreetingService/Makefile`).

After saving configure.ac run '`./autogen.sh`' and '`./configure`' in this order as in case of ordinary ARC1 configuration (doing this needs root privileges).

[1] This directory is required to be under ARC1 home directory ($\$ARC1\_HOME$)

4.4 Configuring the ARC HED

Edit ARC1's XML configuration file used when starting ARC HED.

We define the `http://example.com/GreetingService` namespace in the `<ArcConfig>` node (see below), as we also use this configuration file to store fixed parts of the greeting of `GreetingService`.

```
<ArcConfig
  xmlns="http://www.nordugrid.org/schemas/ArcConfig/2007"
  xmlns:tcp="http://www.nordugrid.org/schemas/ArcMCCTCP/2007"
  xmlns:arex="http://www.nordugrid.org/schemas/a-rex/Config"
  xmlns:gsrv="http://example.com/GreetingService">
```

Illustration 13: Definition of service namespace in ArcConfig node

We have to create a new `<Service>` node in `<Chain>` to represent our service.

```
<Service name="pythonservice" id="greetingservice">
  <ClassName>GreetingService.greetingservice.GreetingClass</ClassName>
  <gsrv:part1>Hello, </gsrv:part1>
  <gsrv:part2>! Local time is: [</gsrv:part2>
  <gsrv:part3>]</gsrv:part3>
</Service>
```

Illustration 14: Definition of service in ARC HED

As mentioned before, Python-based services are handled through a wrapper. This wrapper is called `pythonservice`, that is where the name attribute of `<Service>` node comes from.

ClassName consists of three parts, separated by periods. The first part refers to the package (`GreetingService` here), the second part refers to the Python module (`greetingservice` here – so the Python source file is `greetingservice.py`) while the third one is the name of the Python Class (`GreetingClass`).

The next three nodes (`<gsrv:part1>`, `<gsrv:part2>` and `<gsrv:part3>`) are the fixed parts of the greeting.

All we have to do now is to check the `<Component>` nodes (so we have TCP, HTTP, SOAP and POST present in the HTTP component), then create a new node for `GreetingService` in `<Plexer>`.

```
<Plexer name="plexer.service" id="plexer">
  <next id="a-rex">^/arex$</next>
  <next id="count">^/count$</next>
  <next id="greetingservice">^/GreetingService$</next>
</Plexer>
```

Illustration 15: Configuration of Plexer

The new node tells that the Service, whose ID is `greetingservice`, will be made accessible at the relative URL `/GreetingService`.

4.5 Installation

Installation of the service is also straightforward. Open a terminal, go to the directory of the service (for example `$ARC1_HOME/src/services/GreetingService`) and execute the command: '`make install`' (doing this may need root privileges). After the execution of the command ARC HED needs to be restarted.

5 Testing the Service

The top-down design approach helps us to test `GreetingService`. There are several Open Source tools that can help to test the service without having to write a client for it.

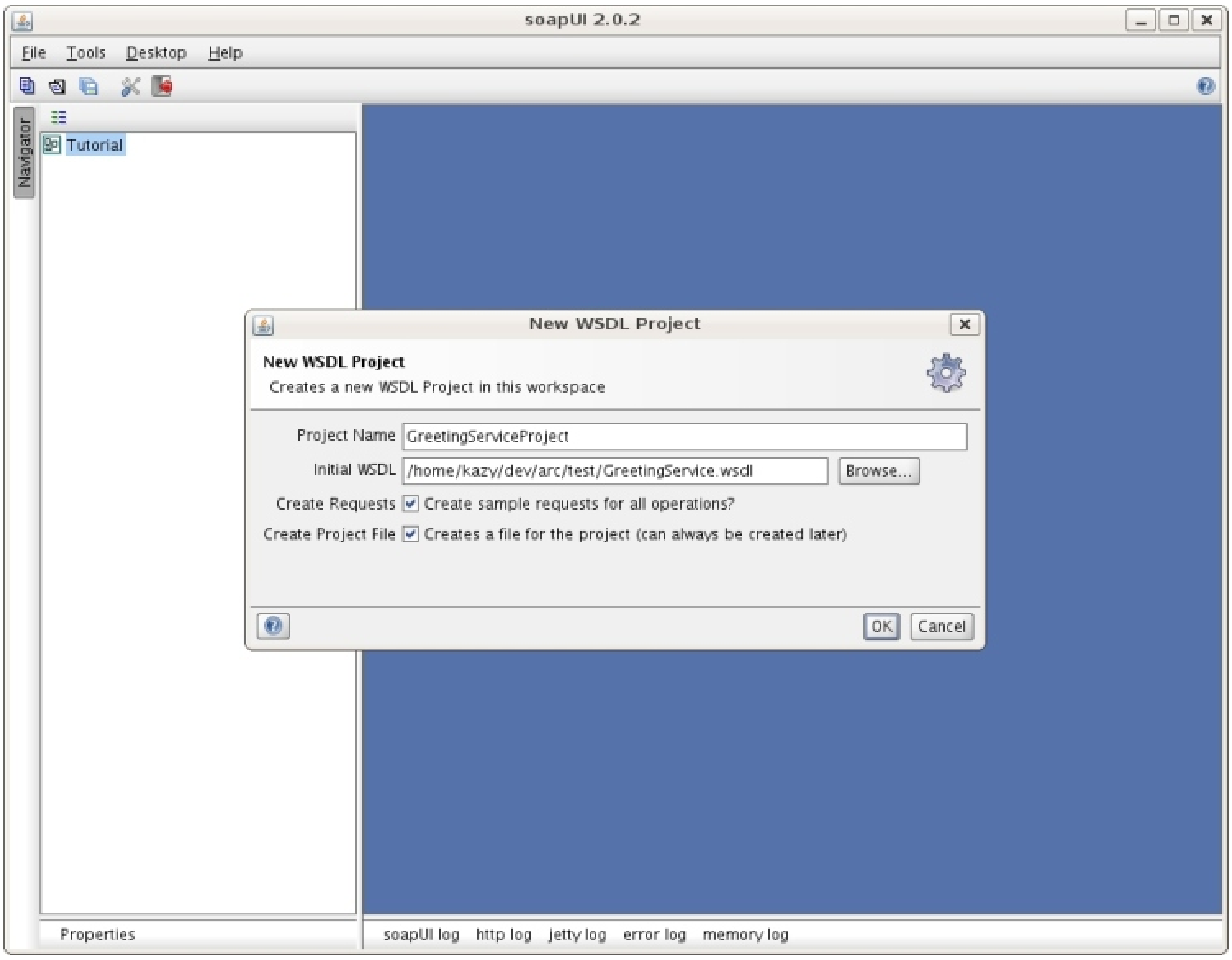
Basically, when considering which tool to use, we look for the following features:

- capability of sending SOAP messages and receiving answers;
- capability of checking XML schema compliance;
- capability of generating message from WSDL;
- possibility of using custom assertions;
- possibility of using parameters.

Our choice for testing the service had been soapUI. It is a widely used open source tool for Web Service Testing that has all the features (even more) we need.

5.1 Creating the Test Project

After starting soapUI, we can create a new WSDL project by pressing Ctrl+N or selecting *New WSDL Project* from the *File* menu. [See *Figure 1*]



New WSDL Project

New WSDL Project
Creates a new WSDL Project in this workspace

Project Name: GreetingServiceProject
Initial WSDL: /home/kazy/dev/arc/test/GreetingService.wsdl
Create Requests: ✓ Create sample requests for all operations?
Create Project File: ✓ Creates a file for the project (can always be created later)

Figure 1: New WSDL Project

Fill in the *Project Name* (it could be any of your choice) and browse for the WSDL that we created before. You should leave *Create Requests* checked. This means that requests are automatically generated from the WSDL. You may also check the *Create Project File* checkbox as it will allow you to save your project.

5.2 Adding the Test Request to a Test Case

Now you should see the project in the *Navigator view.* Navigate to *Request 1* in the tree and open it (press Enter or select *Show Request Editor* from the context menu). [See *Figure 2*]

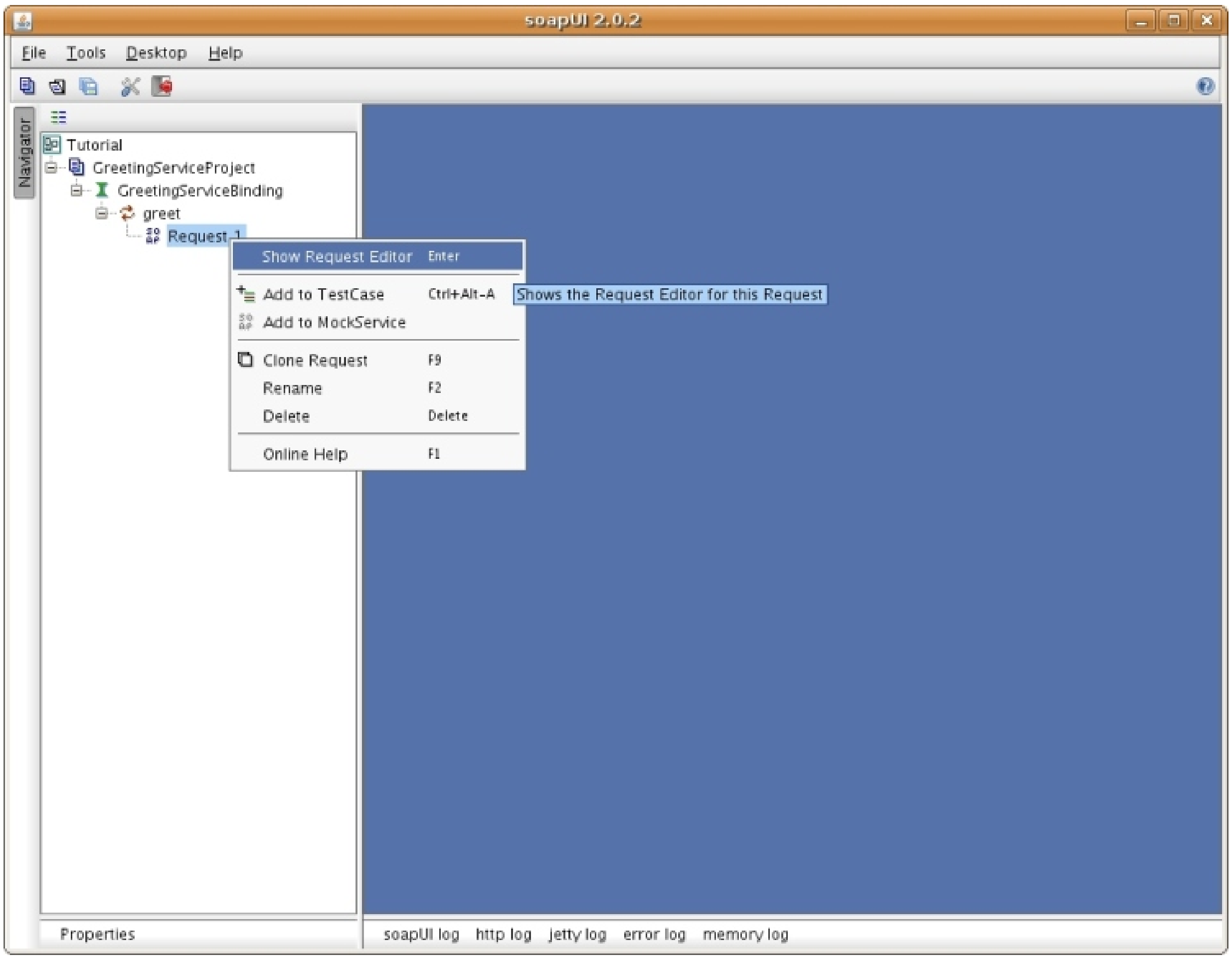


Figure 2: Show Request Editor

Next we add this request to a Test Case. Click on the second icon from the left in the Request Editor (tooltip says: Adds this request to a TestCase). [See *Figure 3*]

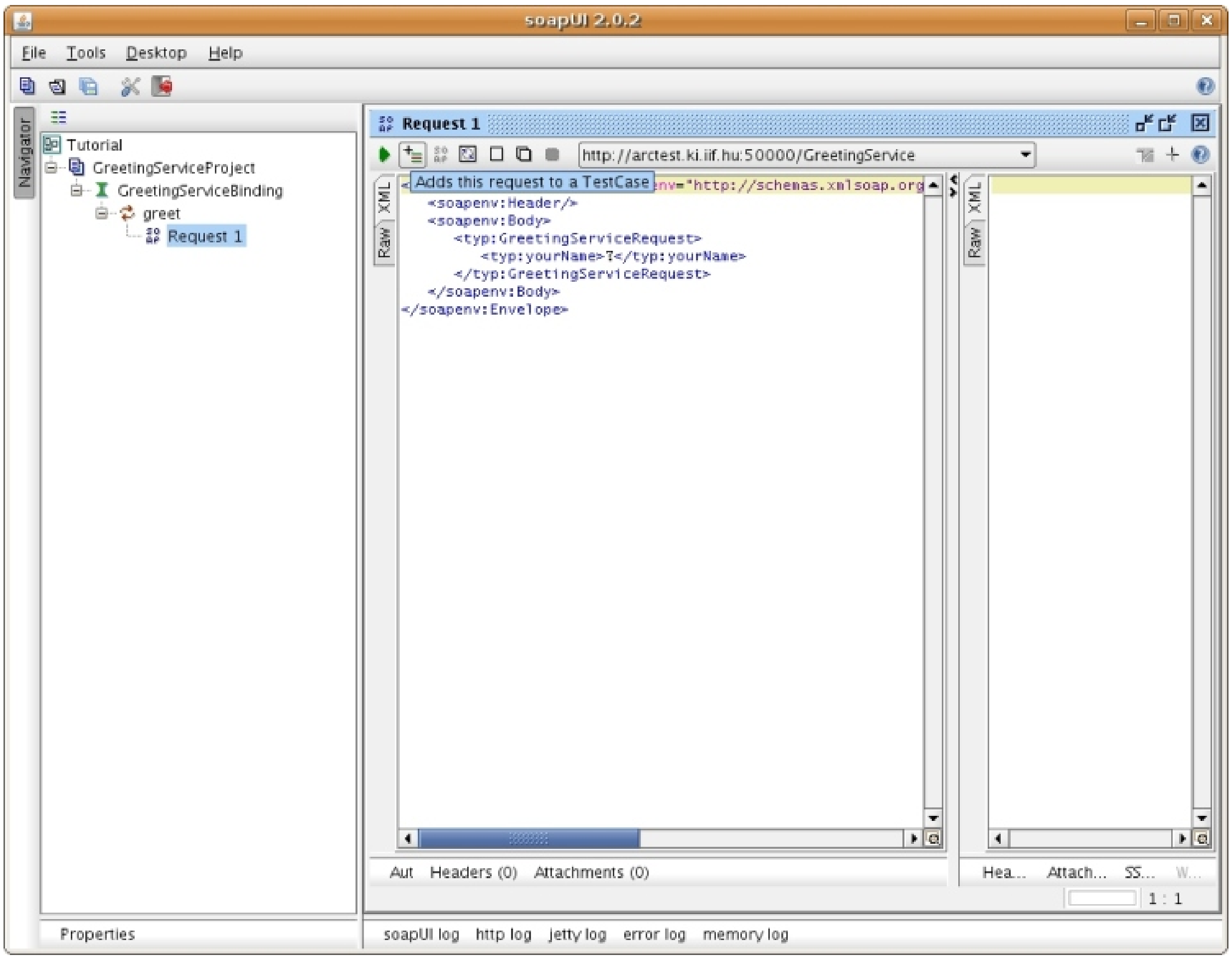
As we do not have a Test Suite yet, we need to create a Test Suite as well. Choose a name for your Test Suite and Test Case (e.g.: GreetingServiceTS and GreetingServiceTC1).

Now, a window appears. Here you can specify the name of the request (e.g.: greetRequest1) and add predefined assertions. For example, you may opt in the *Add Schema Assertion* checkbox to validate messages against our XML schema. [See *Figure 4*]

You should leave *Show TestCase Editor* checked as we will edit the Test Case in the following step.

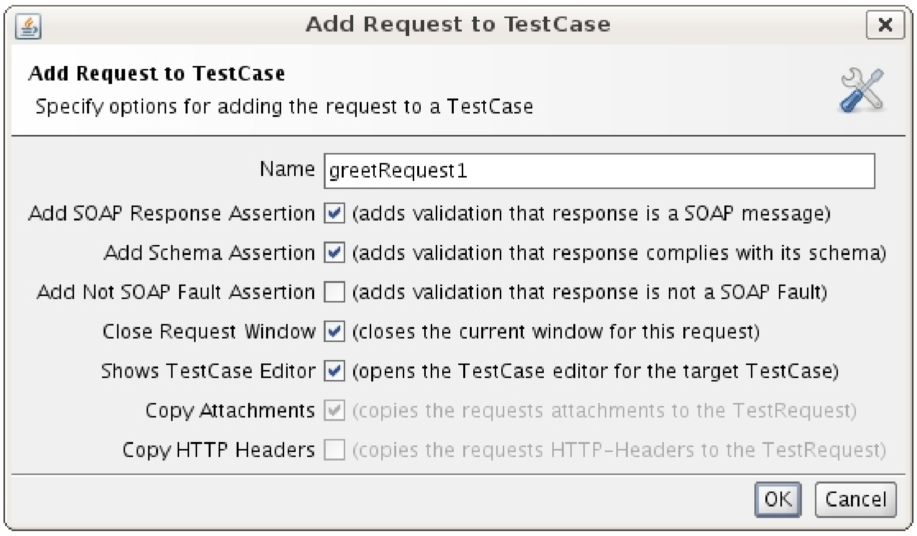
5.3 Editing the Test Case

After the Test Case Editor appears, click on greetRequest and select *Insert Step* → *Properties* in the context menu (tooltip says: Defines / Loads global TestCase properties). [See *Figure 5*]


soapUI 2.0.2
File Tools Desktop Help
Navigator
Tutorial
GreetingServiceProject
GreetingServiceBinding
greet
Request 1
Properties
Request 1
http://arctest.ki.iif.hu:50000/GreetingService
Adds this request to a TestCase
XML
Raw
Aut Headers (0) Attachments (0)
Hea... Attach... SS... W...
1 : 1
soapUI log http log jetty log error log memory log

```
<          env="http://schemas.xmlsoap.org
   <soapenv:Header/>
   <soapenv:Body>
      <typ:GreetingServiceRequest>
         <typ:yourName>?</typ:yourName>
      </typ:GreetingServiceRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 3: Adding Request to a Test Case 1



Add Request to TestCase
Specify options for adding the request to a TestCase

Name	greetRequest1
Add SOAP Response Assertion	✓ (adds validation that response is a SOAP message)
Add Schema Assertion	✓ (adds validation that response complies with its schema)
Add Not SOAP Fault Assertion	☐ (adds validation that response is not a SOAP Fault)
Close Request Window	✓ (closes the current window for this request)
Shows TestCase Editor	✓ (opens the TestCase editor for the target TestCase)
Copy Attachments	✓ (copies the requests attachments to the TestRequest)
Copy HTTP Headers	☐ (copies the requests HTTP-Headers to the TestRequest)

Figure 4: Adding Request to a Test Case 2

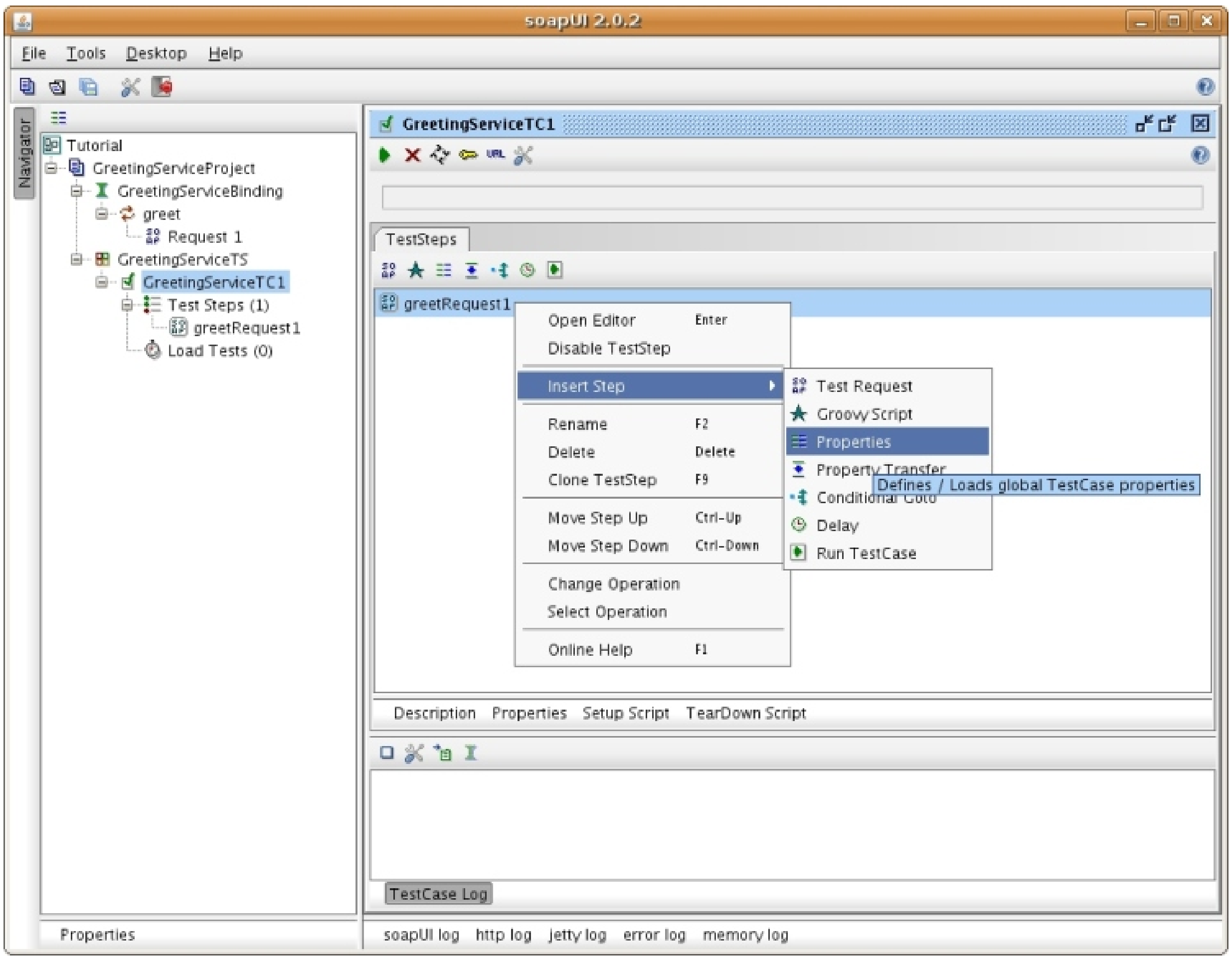


Figure 5: Insert Properties

Fill in the name of the step (e.g.: greetProperties). The properties window shows up. It has a table with two columns (Name and Value). Here you can add a property (e.g.: inputName) by clicking on the first icon from the left (tooltip says: Adds a property to the property list) that could be used in the request and in assertions as well. [See *Figure 6* and *Figure 7*]

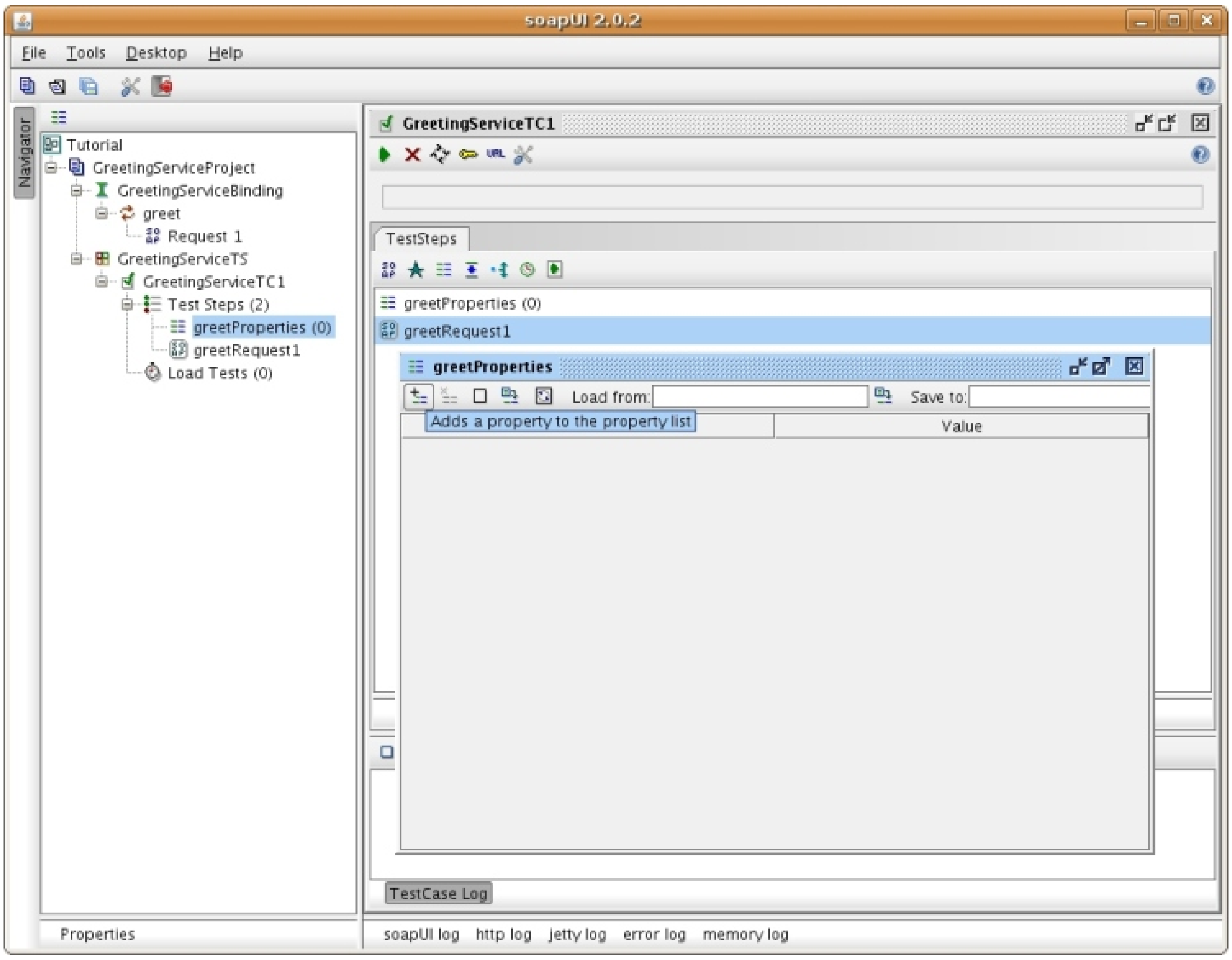


Figure 6: Adding a Property 1

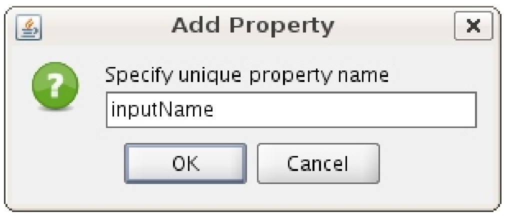


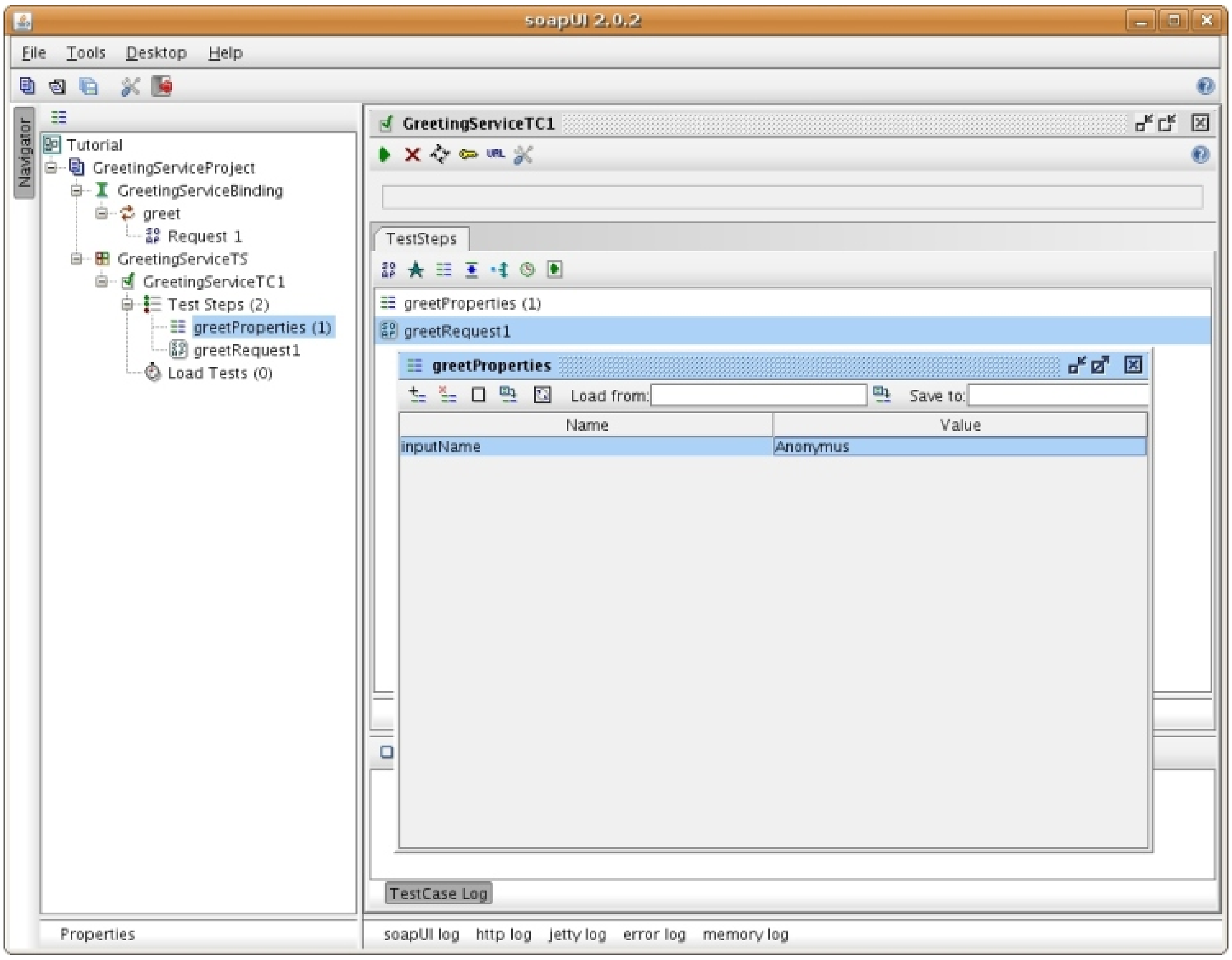
Figure 7: Adding a Property 2

Close the properties window after adding the property (and providing a value for it). [See *Figure 8*]

Now double click on greetRequest1; this opens the Request Editor. In `<yourName>`, write `${inputName}` instead of the default value (a question mark).

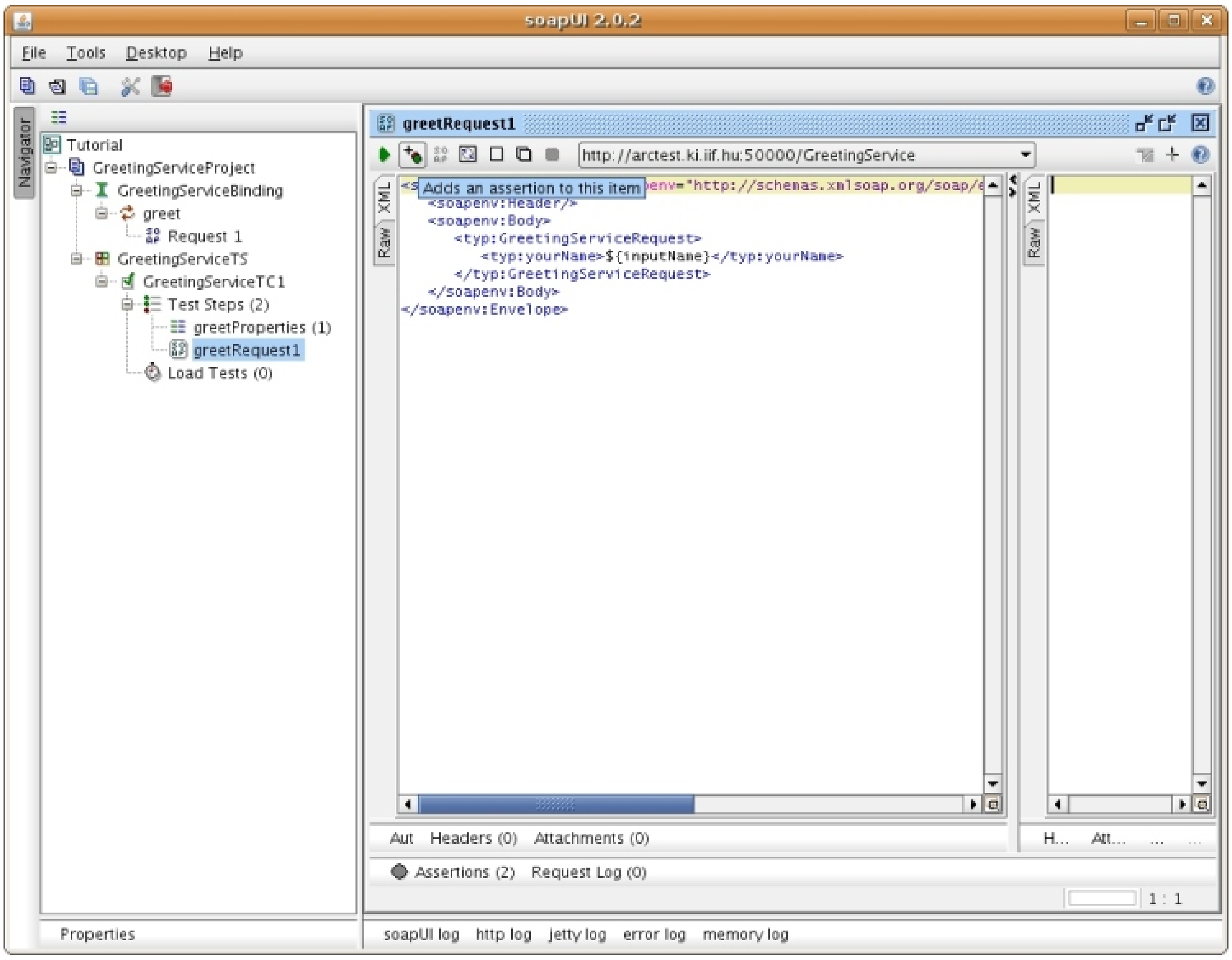
5.4 Assertions

Still in the Request Editor, click on the second icon from the left (tooltip says: Adds an assertion to this item). [See *Figure 9*]



Name	Value
inputName	Anonymus

Figure 8: Adding a Property 3



```
<s                          env="http://schemas.xmlsoap.org/soap/e
   <soapenv:Header/>
   <soapenv:Body>
      <typ:GreetingServiceRequest>
         <typ:yourName>${inputName}</typ:yourName>
      </typ:GreetingServiceRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 9: Adding an Assertion 1

Next, you have to choose the type of assertion. For now, this should be *Contains*. [See *Figure 10*]

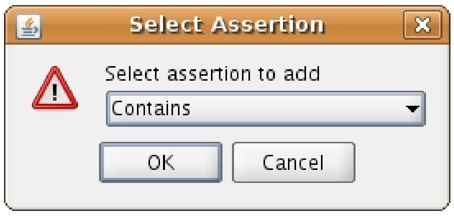
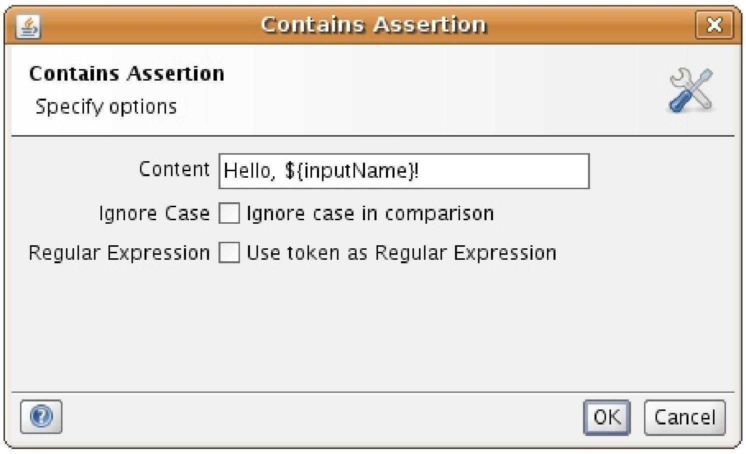


Figure 10: Adding an Assertion 2

In the next step, *Content* should be "`Hello, ${inputName}!`", so if we were to provide "Anonymous" as the value of `inputName`, we could require the answer to contain "Hello, Anonymous!". [See *Figure 11*]



Content: Hello, ${inputName}!

Figure 11: Adding an Assertion 3

5.5 Running the Test Case

Close the Request Editor. In the Test Case Editor, click on the first icon from the left (tooltip says: Runs this testcase). [See *Figure 12*]

5.6 Evaluating Results

If nothing went wrong, you would see the text FINISHED on a green stripe. Details of the results are shown in the *TestCase Log*. [See *Figure 13*]

Double clicking greetRequest1 allows you see the request and response message and the results of assertions as well. [See *Figure 14* and *Figure 15*]

Although the value of the inputName property is not shown in the request message, if you open the *http log*, you will see that the correct value has been used while creating the outgoing request. [See *Figure 16*]

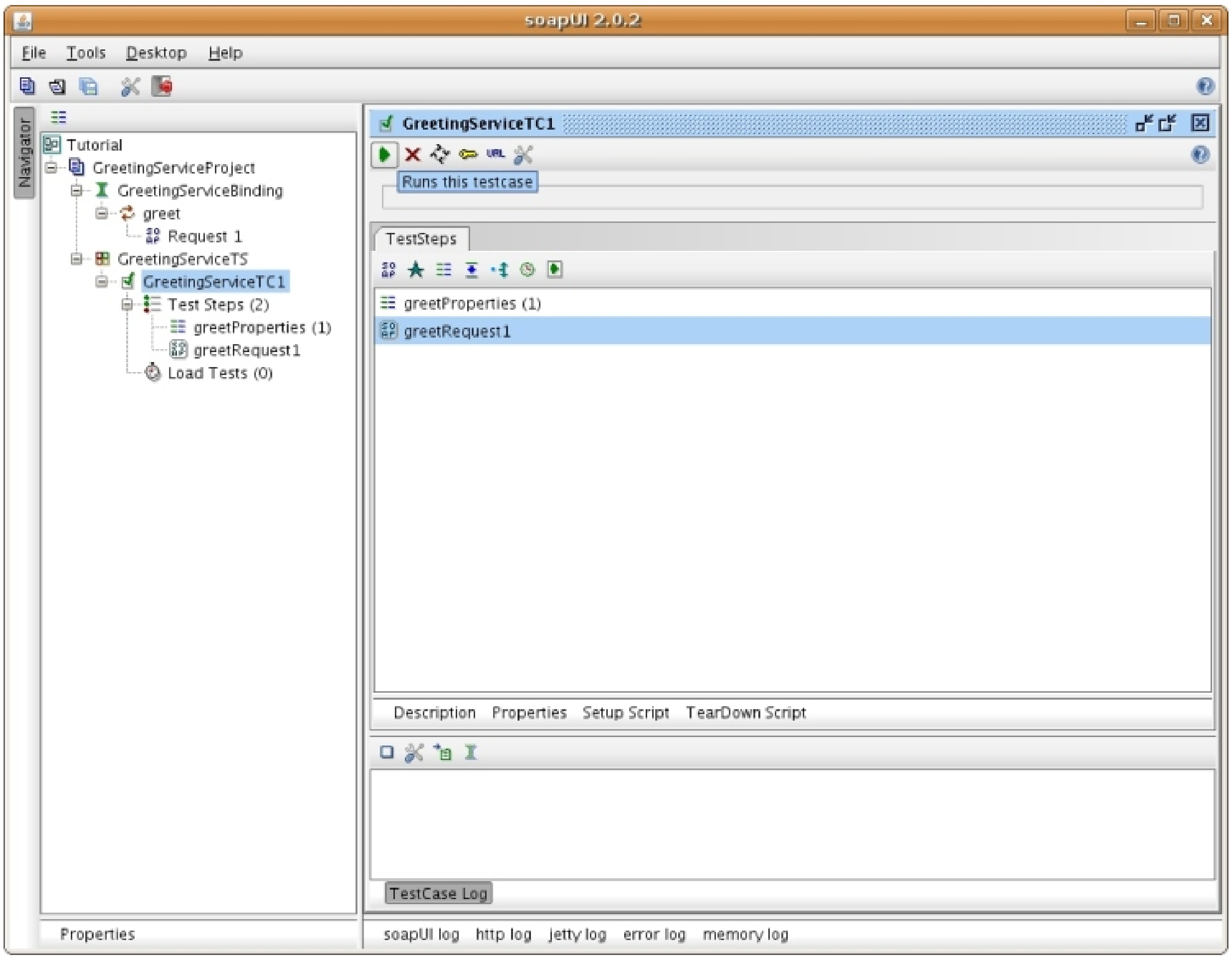
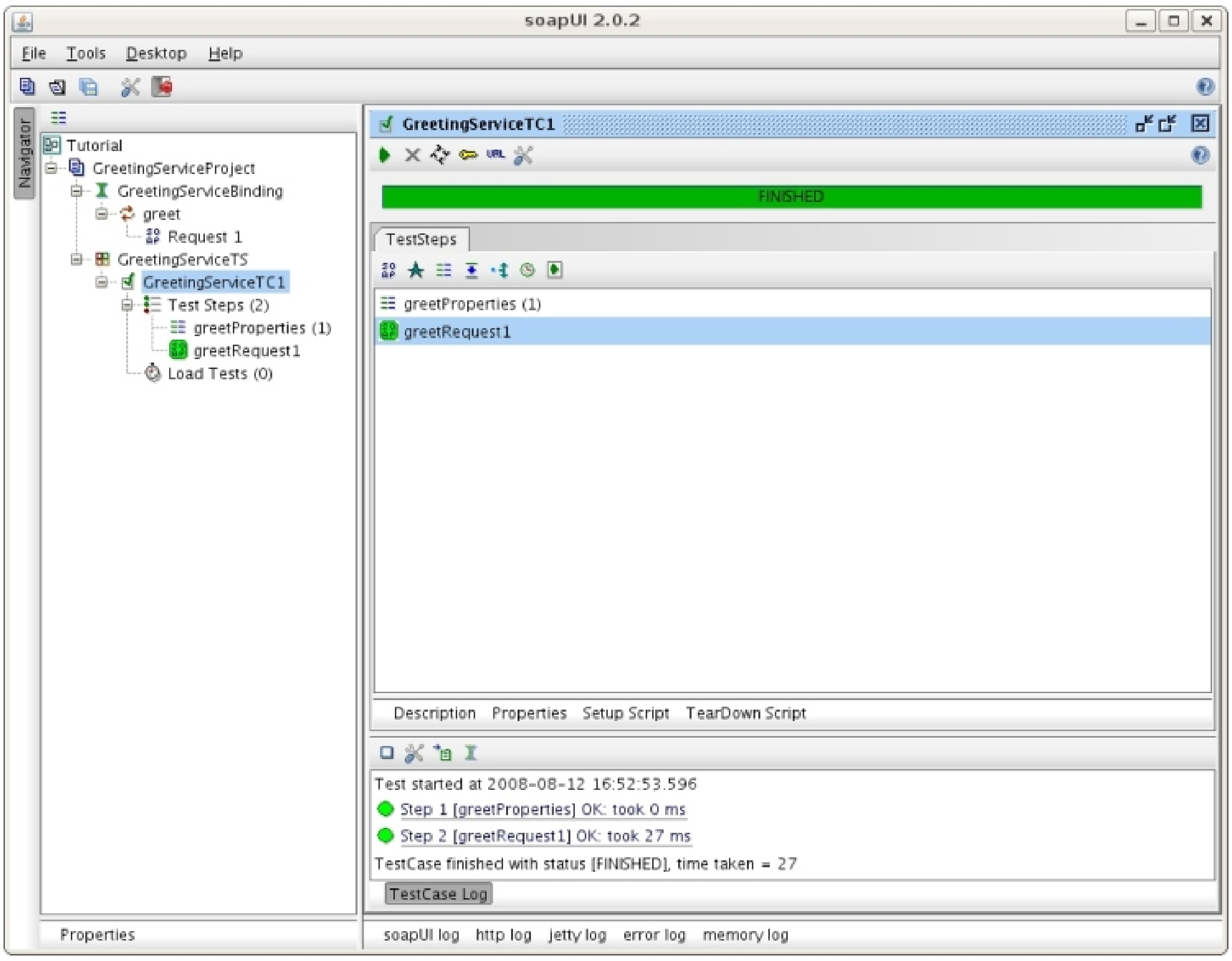
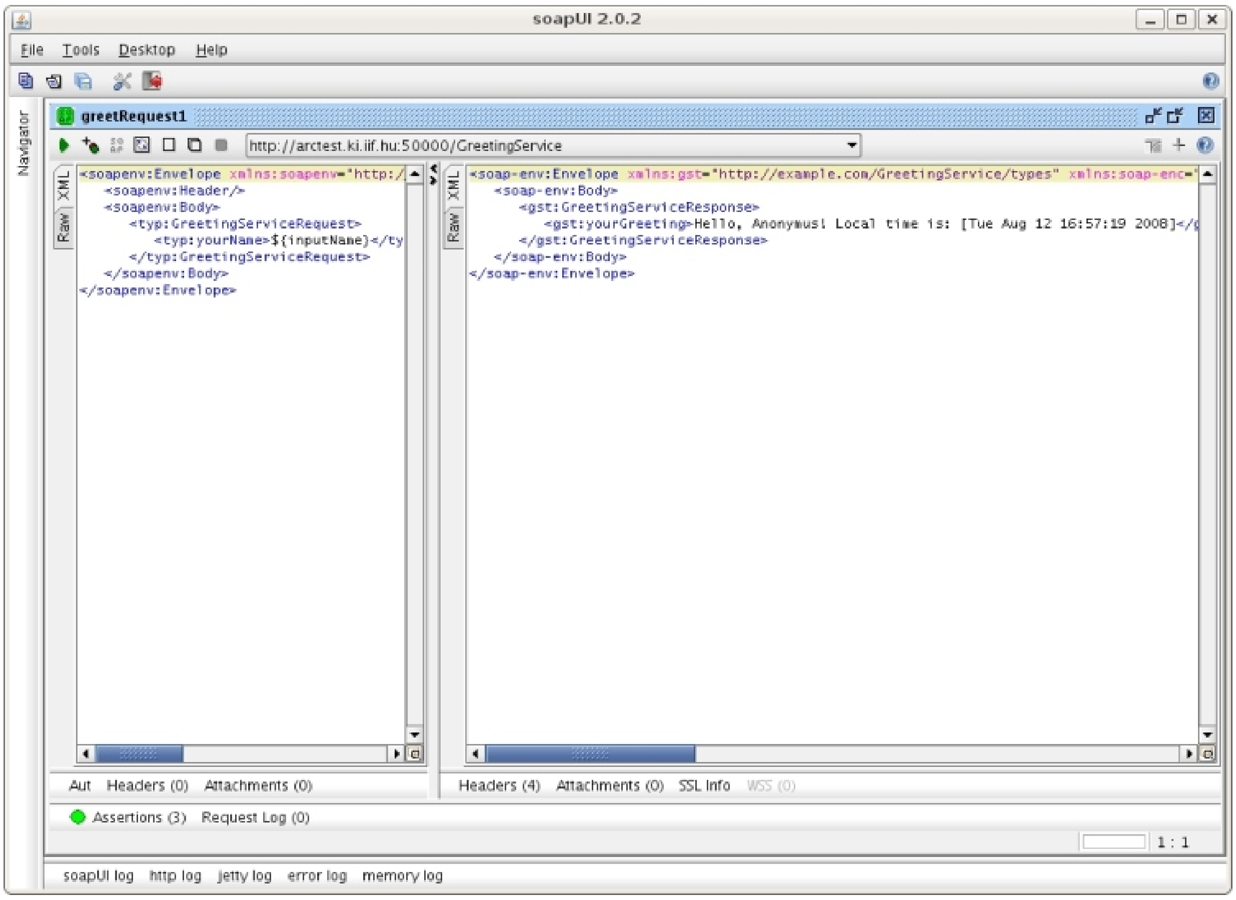


Figure 12: Running the Test Case



```
Test started at 2008-08-12 16:52:53.596
Step 1 [greetProperties] OK: took 0 ms
Step 2 [greetRequest1] OK: took 27 ms
TestCase finished with status [FINISHED], time taken = 27
```

Figure 13: Test Case Results



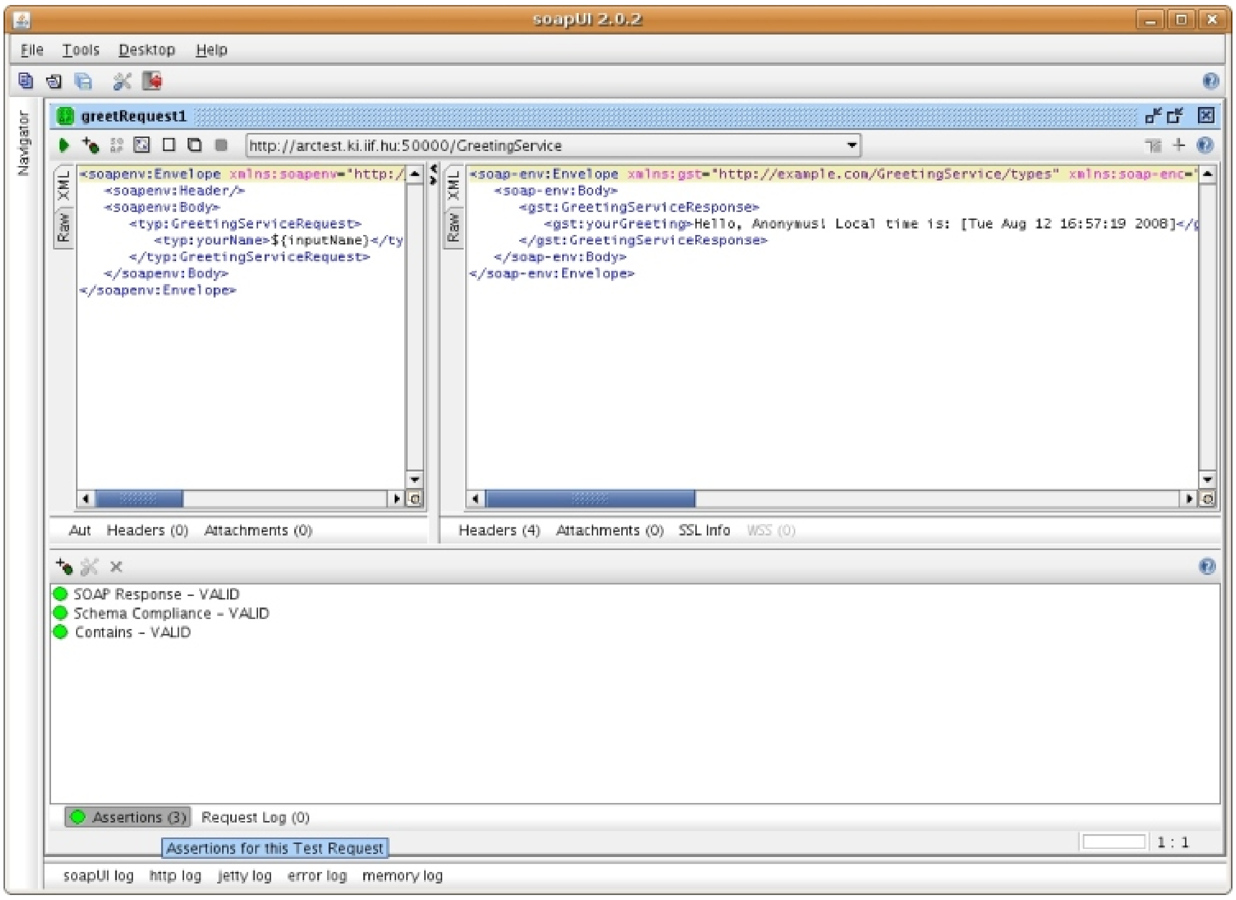
soapUI 2.0.2
File Tools Desktop Help
Navigator
greetRequest1
XML
Raw
Aut Headers (0) Attachments (0)
Headers (4) Attachments (0) SSL Info WSS (0)
Assertions (3) Request Log (0)
1 : 1
soapUI log http log jetty log error log memory log

```
http://arctest.ki.iif.hu:50000/GreetingService
```

```
<soapenv:Envelope xmlns:soapenv="http:/
   <soapenv:Header/>
   <soapenv:Body>
      <typ:GreetingServiceRequest>
         <typ:yourName>${inputName}</ty
      </typ:GreetingServiceRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

```
<soap-env:Envelope xmlns:gst="http://example.com/GreetingService/types" xmlns:soap-enc="
   <soap-env:Body>
      <gst:GreetingServiceResponse>
         <gst:yourGreeting>Hello, Anonymus! Local time is: [Tue Aug 12 16:57:19 2008]</g
      </gst:GreetingServiceResponse>
   </soap-env:Body>
</soap-env:Envelope>
```

Figure 14: Response Message

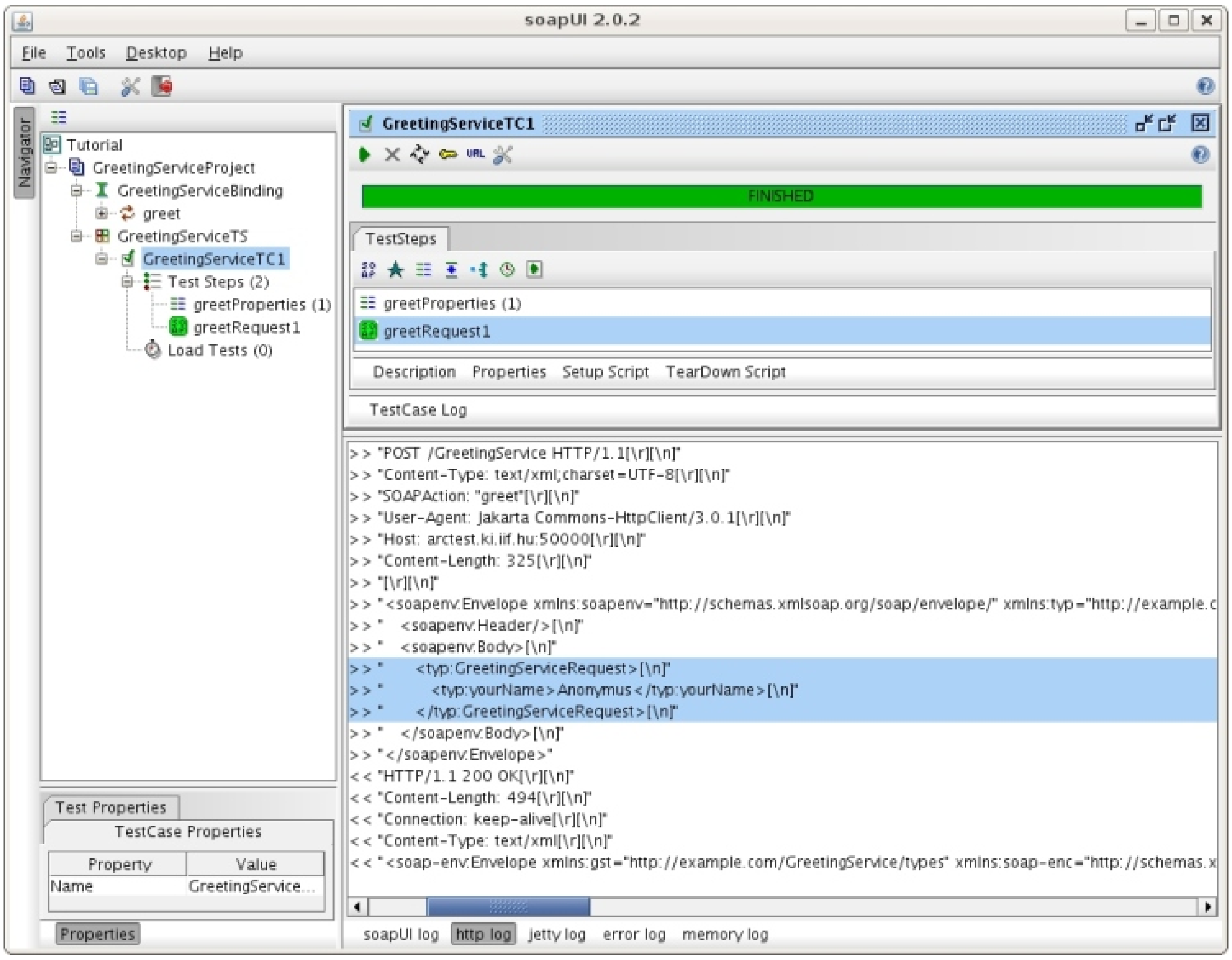

soapUI 2.0.2
File Tools Desktop Help
Navigator
greetRequest1
http://arctest.ki.iif.hu:50000/GreetingService
XML
Raw
Aut Headers (0) Attachments (0)
Headers (4) Attachments (0) SSL Info WSS (0)
Assertions (3) Request Log (0)
Assertions for this Test Request
1 : 1
soapUI log http log jetty log error log memory log

```
<soapenv:Envelope xmlns:soapenv="http:/
   <soapenv:Header/>
   <soapenv:Body>
      <typ:GreetingServiceRequest>
         <typ:yourName>${inputName}</ty
      </typ:GreetingServiceRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

```
<soap-env:Envelope xmlns:gst="http://example.com/GreetingService/types" xmlns:soap-enc="
   <soap-env:Body>
      <gst:GreetingServiceResponse>
         <gst:yourGreeting>Hello, Anonymus! Local time is: [Tue Aug 12 16:57:19 2008]</g
      </gst:GreetingServiceResponse>
   </soap-env:Body>
</soap-env:Envelope>
```

```
SOAP Response - VALID
Schema Compliance - VALID
Contains - VALID
```

Figure 15: Assertions



Property	Value
Name	GreetingService...

```
>> "POST /GreetingService HTTP/1.1[\r][\n]"
>> "Content-Type: text/xml;charset=UTF-8[\r][\n]"
>> "SOAPAction: "greet"[\r][\n]"
>> "User-Agent: Jakarta Commons-HttpClient/3.0.1[\r][\n]"
>> "Host: arctest.ki.iif.hu:50000[\r][\n]"
>> "Content-Length: 325[\r][\n]"
>> "[\r][\n]"
>> "<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:typ="http://example.c
>> "   <soapenv:Header/>[\n]"
>> "   <soapenv:Body>[\n]"
>> "      <typ:GreetingServiceRequest>[\n]"
>> "         <typ:yourName>Anonymus</typ:yourName>[\n]"
>> "      </typ:GreetingServiceRequest>[\n]"
>> "   </soapenv:Body>[\n]"
>> "</soapenv:Envelope>"
<< "HTTP/1.1 200 OK[\r][\n]"
<< "Content-Length: 494[\r][\n]"
<< "Connection: keep-alive[\r][\n]"
<< "Content-Type: text/xml[\r][\n]"
<< "<soap-env:Envelope xmlns:gst="http://example.com/GreetingService/types" xmlns:soap-enc="http://schemas.x
```

Figure 16: HTTP Log

A Appendices

A.1 GreetingService.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
  targetNamespace="http://example.com/GreetingService/types"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://example.com/GreetingService/types"
  elementFormDefault="qualified">

  <simpleType name="name">
    <restriction base="xsd:string"/>
  </simpleType>

  <simpleType name="greeting">
    <restriction base="xsd:string"/>
  </simpleType>

  <complexType name="GreetingServiceRequestType">
    <sequence>
        <element name="yourName" type="tns:name"
                 minOccurs="1" maxOccurs="1"/>
    </sequence>
  </complexType>

  <complexType name="GreetingServiceResponseType">
    <sequence>
      <element name="yourGreeting" type="tns:greeting"
               minOccurs="1" maxOccurs="1"/>
    </sequence>
  </complexType>

  <element name="GreetingServiceRequest"
           type="tns:GreetingServiceRequestType"/>
  <element name="GreetingServiceResponse"
           type="tns:GreetingServiceResponseType"/>
</schema>
```

A.2 GreetingService.wsdl

```
<?xml version="1.0"
 encoding="UTF-8"?>
<definitions
  targetNamespace="http://example.com/GreetingService"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:gs="http://example.com/GreetingService"
  xmlns:gst="http://example.com/GreetingService/types">

  <wsdl:types>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:import namespace="http://example.com/GreetingService/types"
                  schemaLocation="GreetingService.xsd"/>
    </xsd:schema>
  </wsdl:types>

  <wsdl:message name="GreetingServiceRequestMessage">
    <wsdl:part name="payload" element="gst:GreetingServiceRequest"/>
  </wsdl:message>

  <wsdl:message name="GreetingServiceResponseMessage">
    <wsdl:part name="payload" element="gst:GreetingServiceResponse"/>
  </wsdl:message>

  <wsdl:portType name="GreetingServicePT">
    <wsdl:operation name="greet">
      <wsdl:input name="GreetingServiceInput"
                  message="gs:GreetingServiceRequestMessage"/>
      <wsdl:output name="GreetingServiceOutput"
                   message="gs:GreetingServiceResponseMessage"/>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="GreetingServiceBinding" type="gs:GreetingServicePT">
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="greet">
      <soap:operation soapAction="greet"/>
      <wsdl:input name="GreetingServiceInput">
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="GreetingServiceOutput">
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="GreetingService">
    <wsdl:port name="GreetingServicePort"
               binding="gs:GreetingServiceBinding">
      <soap:address location="http://arctest.ki.iif.hu:50000/GreetingService"/>
    </wsdl:port>
  </wsdl:service>

</definitions>
```

A.3 greetingservice.py

```
import arc
import time

class GreetingClass:

    def __init__(self, cfg):
        # we want to use members of the 'http://example.com/GreetingService' namespace
        # so we need to know which namespace prefix was assigned to it
        # we get the namespace prefix into 'self.gs_ns'
        self.gs_ns = cfg.NamespacePrefix('http://example.com/GreetingService')

        # get part1 of greeting from the config XML
        self.part1 = str(cfg.Get(self.gs_ns + ':part1'))
        # get part2 of greeting from the config XML
        self.part2 = str(cfg.Get(self.gs_ns + ':part2'))

        # get part3 of greeting from the config XML
        self.part3 = str(cfg.Get(self.gs_ns + ':part3'))

    def process(self, inmsg, outmsg):

        # get the payload from the message
        inpayload = inmsg.Payload()

        # we want to use members of the 'http://example.com/GreetingService/types'
        # namespace so we need to know which namespace prefix was assigned to it
        # we get the namespace prefix into a variable named 'gst_ns'
        gst_ns = inpayload.NamespacePrefix('http://example.com/GreetingService/types')

        # then we can get 'gst_ns:GreetingServiceRequest/gst_ns:yourName'
        GreetingServiceRequest = inpayload.Get(gst_ns + ':GreetingServiceRequest')

        name = str(GreetingServiceRequest.Get(gst_ns + ':yourName'))

        localtime = time.localtime()

        strlocaltime = time.asctime(localtime)

        # forge the greeting message
        greet = self.part1 + name + self.part2 + strlocaltime + self.part3

        # create an answer payload
        outpayload = arc.PayloadSOAP(\
                        arc.NS({'gst':'http://example.com/GreetingService/types'}))

        # here we defined that 'gst' prefix will be the namespace prefix of
        # 'http://example.com/GreetingService/types' and
        # we create a node at '/gst:GreetingServiceResponse/gst:yourGreeting'
        # and put the string in it
        outpayload.NewChild('gst:GreetingServiceResponse').\
                        NewChild('gst:yourGreeting').Set(greet)

        # put the payload into the outgoing message
        outmsg.Payload(outpayload)

        # return with STATUS_OK
        return arc.MCC_Status(arc.STATUS_OK)
```