Triana User Guide

The Triana Team

Preface

Preface stuff to go here

Contents

1 Overview **1**
- 1.1 Introduction . . . 1

2 Getting Started **2**
- 2.1 Download, Installation and Configuration . . . 2
- 2.2 CVS Access . . . 3
  - 2.2.1 Conventions . . . 3
  - 2.2.2 CVSRoot and Passwords . . . 3
  - 2.2.3 A Word About Directory Structure . . . 4
  - 2.2.4 Easy Install . . . 4
  - 2.2.5 Developer Install . . . 6
  - 2.2.6 Other Install . . . 8
- 2.3 Workflow Basics . . . 8
  - 2.3.1 Simple Workflow Example . . . 8
  - 2.3.2 Group Tools . . . 8

3 Distributed Computing with Triana **9**
- 3.1 Overview . . . 9
- 3.2 Web Services . . . 9
  - 3.2.1 Web Service Configuration . . . 10
  - 3.2.2 Discovering Web Services . . . 11
  - 3.2.3 Importing Web Services . . . 12
  - 3.2.4 Conncecting Web Services . . . 12
  - 3.2.5 Bible Translation Example . . . 13
  - 3.2.6 Complex Data Types . . . 14
  - 3.2.7 Deploying Web Services . . . 15
- 3.3 P2PS . . . 17
- 3.4 GAT . . . 17

4 Applications and Case Studies **18**

4.1 Data Analysis with Triana 18
4.2 Audio Processing with Triana 18

5 Extending Triana **19**
5.1 Writing your own tools 19
5.1.1 Using the Tool Wizard 19
5.2 Advanced Tool Techniques 19
5.2.1 Showing and Hiding a Unit's Parameter Panel 19
5.2.2 Pausing Unit Execution 20
5.3 Triana as a Workflow Editor 23

6 Demos **24**

List of Figures

2.1 example *build.properties* file 8

3.1 Web Service Configuration Dialog. 10
3.2 Web Service tools in the Tool Tree. 11
3.3 Using StringGen and StringViewer to provide input to and display the output from a temperature conversion web service. . . . 12
3.4 A simple bible translation workflow. 14
3.5 Generating/viewing complex data types using WSTypeGen and WSTypeViewer. 15
3.6 Dialog for deploying a task/group task as a web service. 16
3.7 Using xsd tools to ensure standard input/output XML types are used when deploying a web service. 17

Chapter 1

Overview

1.1 Introduction

Chapter 2

Getting Started

2.1 Download, Installation and Configuration

Step 1 - Download
Download the latest Triana release from `http://www.trianacode.org`.

Step 2 - Set Environment Variable
Set an environment variable for your system, `$TRIANA` for Unix like systems or `%TRIANA%` for Windows, to point the top level `triana` directory, the location that you saved and unpacked the download to.

e.g. (from the command prompt)

`set TRIANA=C:\triana`	*Windows*
`setenv TRIANA=/home/user/triana`	*unix - tcsh*
`export TRIANA=/home/user/triana`	*unix - bash*

Step 3 - Build Triana
Run the Triana build script (`buildTriana`), which is located in the `triana/bin` directory.

e.g. (from the command prompt)

`C:\triana\bin\buildTriana.bat`	*Windows*
`/home/user/triana/bin/buildTriana`	*unix*

Step 4 - Run Triana
Run Triana using the `triana` script located in the `triana/bin` directory.

e.g. (from the command prompt)

`C:\triana\bin\triana.bat`	*Windows*
`/home/user/triana/bin/triana`	*unix*

2.2 CVS Access

This section describes the steps in correctly checking out the source code for the core Triana and Triana toolboxes from the cvs repository and building from the source code. If you don't know how to use a CVS client or don't have a CVS account, then it is recommended that you follow the instructions in section 2.1.

2.2.1 Conventions

Note: These instructions assume you have the necessary user accounts and passwords. It is aimed at command line cvs users. WinCVS or similar users should be able to follow the instructions by using the repository and module names within their particular CVS Client.

- `Text written this font` should be typed as command line input. Commands should be entered without line breaks unless explicitly instructed. (This includes line breaks due to book formatting)
- *Text written in this font* and surrounded by < ... > should be replaced by the users appropriate details.
- **Text in this font** signifies the unix command line prompt, the text following it will be the command to type.

2.2.2 CVSRoot and Passwords

The "CVSROOT" you should you will depend on whether you have read only "pserver" access or write "ext" access. This document assumes "pserver", if you don't know then ask the person who gave you your user name and password.

For the Core Triana and default Toolbxes repositories there is anonymous "pserver" access, use the username <*anonymous*> with no password, just hit return at the password prompt.

2.2.3 A Word About Directory Structure

For administration reasons Triana is split into a number of different packages which are stored in various CVS repositories. Some of these are project specific and not public so don't assume that because something is listed in this document that you will be able to get the source code from the CVS server.

The public packages are:

1. Core Triana. The Triana Environment itself.
2. Toolboxes. The default toolboxes that come with Triana.
3. Toolboxes-dev. The unstable toolboxes that developers are currently working with. Use at you own risk.

The project specific packages are:

1. GEO. The Gravitational Waves tools.
2. Gravity. Example tools from the book "Gravity From The Ground Up" by Bernard Schutz.
3. GriPhyN. Tools from collaboration work with the GriPhyN project.

There are two standard ways to structure a Triana distribution from CVS, dependant on: whether you expect to be making regular changes to the source code under your control and/or you want to do frequent updates for the latest version from CVS; Or you just want to check out the latest version and build it once. CVS allows checking modules out inside other modules which will give the same structure as the packaged version of Triana but it will complain if an `update` or `commit` is attempted in the source tree where there is a foreign module lower down the tree.

2.2.4 Easy Install

These instructions will checkout and install Triana to the same structure as the packaged release version. It is fine for most users however if you expect to use CVS for more than updating small numbers of files it is suggested you use the other set of instructions.

This install will put all of the various toolboxes inside the Triana source tree and they will need to be removed to perform a `cvs update` on the core Triana code and then checked back out again.

From the command line:

prompt$ `cvs -d :pserver:`<*username*>`@trianacode.org:/home/cvsroot/triana login`
(Logging in to username@trianacode.org)
CVS password: <*userpassword*>

prompt$ `cvs -d :pserver:`<*username*>`@trianacode.org:/home/cvsroot/triana checkout -P triana`

prompt$ `cd triana`

prompt$ `cvs -d :pserver:`<*username*>`@trianacode.org:/home/cvsroot/trianatools login`
(Logging in to username@trianacode.org)
CVS password:] <*userpassword*>

prompt$ `cvs -d :pserver:`<*username*>`@trianacode.org:/home/cvsroot/trianatools checkout -P toolboxes`

Note: The rest of the Triana modules from CVS are optional and depend on project permissions to access.

prompt$ `cd toolboxes`

prompt$ `cvs -d :pserver:`<*username*>`@trianacode.org:/home/cvsroot/trianatools checkout -P toolboxes-dev`

prompt$ `cvs -d :pserver:`<*username*>`@trianacode.org:/home/cvsroot/geotools login`
(Logging in to username@trianacode.org)
CVS password:] <*userpassword*>

prompt$ `cvs -d :pserver:`<*username*>`@trianacode.org:/home/cvsroot/geotools checkout -P GEO`

prompt$ `cvs -d :pserver:`<*username*>`@trianacode.org:/home/cvsroot/gravity login`
(Logging in to username@trianacode.org)
CVS password:] <*userpassword*>

prompt$ `cvs -d :pserver:`<*username*>`@trianacode.org:/home/cvsroot/gravity checkout -P GravityFromTheGroundUp`

prompt$ `cvs -d :pserver:`*<username>*`@trianacode.org:/home/cvsroot/`
`griphyntools login`
(Logging in to username@trianacode.org)
CVS password:] *<userpassword>*

prompt$ `cvs -d :pserver:`*<username>*`@trianacode.org:/home/cvsroot/`
`griphyntools checkout -P GriPhyN`

After those CVS commands you should have a directory structure like this:

```
triana/
      /bin/
      /toolboxes/
                /Audio
                /... other standard toolboxes
                /toolboxes-dev
                /GEO
                /GravityFromTheGroundUp/
                /GriPhyN
```

The toolbox structure is the import thing, there will be extra directories under `triana/` not mentioned here.

Build and Run

To build set an environment variable for your system, `$TRIANA` for Unix like systems or `%TRIANA%` for Windows, to point the top level `triana` directory. Run the build script, `buildTriana` for Unix or `buildTriana.bat` for Windows:

prompt$ `$TRIANA/bin/buildTriana`

Run the start script, `triana` or `triana.bat`:

prompt$ `$TRIANA/bin/triana`

2.2.5 Developer Install

If you are intending to write or modify any code within the various CVS modules or you just want to regularly update the modules from CVS. Then we suggest that you keep all the CVS modules in their own separate directory stuctures. The *Ant*[1] build file is written to be able to build from default either the previous

[1] `http://ant.apache.org/`

structure or a directory structure where all the cvs modules are at the same level in the file system.

For instance my directory structure looks like this:

```
project/triana/
project/toolboxes/
project/toolboxes-dev/
project/toolboxes_other
project/toolboxes_other/GEO
project/toolboxes_other/GravityFromTheGroundUp
project/toolboxes_other/GriPhyN
```

So from a sensible starting directory run the CVS commands from section 2.2.4 with the `cd` commands so that the Triana core and default toolboxes modules from cvs reside in your current directory.

Note: The GEO, Gravity and Griphyn toolboxes should really reside in a separate subdirectory. Here I've created a directory called "toolboxes_other", this is because Triana assumes that a toolbox contains packages, so GEO for instance is the top level package for the GEO tools. When Triana attempts to locate tools it uses the following path :-

[toolbox][tool package][toolname]*

e.g. `[/project/toolboxes/][SignalProc/][Input/][Wave]`
or `[/project/toolboxes_other/][GEO/][Algorithms/][SaturationMon]`

The build and run instructions are almost the same as for the easy install.

- Set the `TRIANA` environment variable to point to the `triana` directory.
- Either edit the build file (build.xml in the main Triana home directory) and set the appropriate toolbox location variables in the "User Editable" section, or the preferred method add a new file called "build.properties" to the Triana home directory with the properties and their values. the contents of my "build.properties" file can be seen in figure 2.1
- Run the `buildTriana` script.

```
javac.flag.debug=on
javac.flag.deprecation=on
triana.geo.tools=../toolboxes_other/GEO
triana.gravity.tools=../toolboxes_other/ \
    GravityFromTheGroundUp
triana.griphyn.tools=../toolboxes_other/GriPhyN}
```

Figure 2.1: example *build.properties* file

Note: Unlike the standard installation, Triana will not automatically pick up the toolboxes when it is started so you will have to add those within Triana from the menu *Tools, Edit Tool Box Paths*. In my case, I select the directories - `/project/toolboxes/`, `/project/toolboxes-dev/`, `/project/toolboxes_other/`.

2.2.6 Other Install

You can actually have your toolbox modules anywhere you like in your file system and still have the build process pick them up and compile them. In the file `build.xml` there is a commented section titled "USER EDITABLE PROPERTIES" within this section there are a number of commented out properties for the various toolbox modules. Uncomment any of these and set them to the appropriate location to override the default locations. Alternatively add a file called "build.properties" with the appropriate lines containing - `propertyname=value`.

2.3 Workflow Basics

2.3.1 Simple Workflow Example

2.3.2 Group Tools

Chapter 3

Distributed Computing with Triana

3.1 Overview

3.2 Web Services

An important current paradigm in distributed computing is web services. Web services are remote software components accessible using standard network protocols and with defined XML-based interfaces. Already a large number of web service components are available via the Internet, and increasingly legacy applications are being wrapped in web service interfaces.

In Triana terms, web services function exactly as locally available tools. A web service receives input data, performs some operation on that data, and returns results. Due to this similarity, Triana allows the user use web services within a workflow as if they are standard tools. Once a web service has been discovered or imported (see Sections 3.2.2 and 3.2.3), it appears as a tool in the tool tree alongside the other tools and can be connected into a Triana workflow in exactly the same manner of other tools.

As well as discovery and importing web services, Triana allows the user to deploy a workflow subsection as a web service for other users (including other Triana users) to access. We discuss this more in Section 3.2.7.

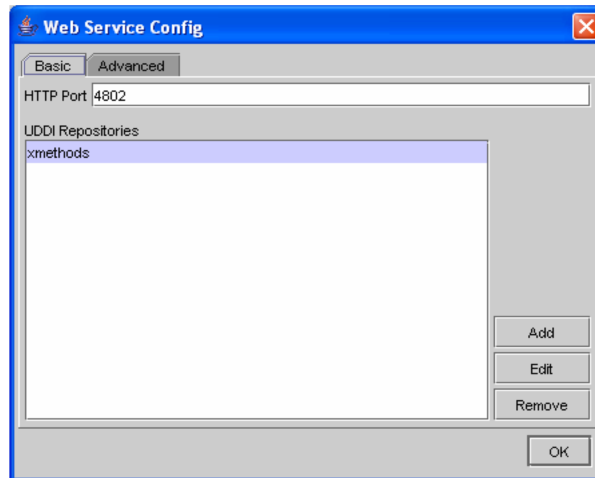

Figure 3.1: Web Service Configuration Dialog.

3.2.1 Web Service Configuration

Web service configuration can be done using the *Services*→*Configure* menu option, and the selecting *Configure* next to the web service option. It should be noted that Triana automatically loads the last used web service configuration and therefore this configuration step needs only be done if you are changing the web service options, e.g. using a different UDDI repository. It should also be noted that once web services features (e.g. discovery) have been utilized the configuration cannot be changed within the current running Triana. This will be indicated by a disabled *Configure* option.

The web service configuration dialog (see Figure 3.1) is split into two panels, *Basic* and *Advanced*. The *Basic* panel enables the web service HTTP port and the UDDI repository to be set. The HTTP port is the port used by services created within Triana (see Section 3.2.7) only. If using this feature then this port must be open on the local firewall otherwise users will not be able to access the services. For simply invoking external web services not port need to be opened.

The UDDI repository is used by Triana to discover services (see Section 3.2.2. By default the XMethods (`www.xmethods.net`) is set, but additional repositories can be added. Note that when adding a UDDI repository only the inquiry address is required, the publish address is only required if deploying services within Triana, and a username/password are only needed if required by the UDDI instance.

To publish in some UDDI repositories trust stores are required. If this is the case

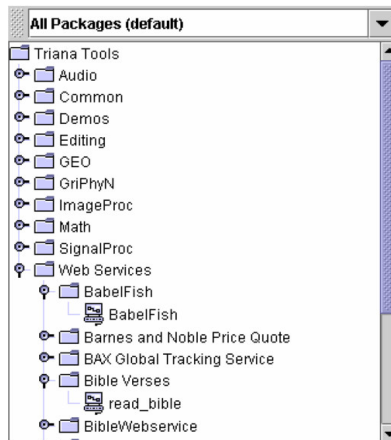

Figure 3.2: Web Service tools in the Tool Tree.

then the location of the trust store file should be specified in the `config.xml` which is located in the `/.wspeer/admin` directory. Dummy trust store files can be found in the `triana/system` directory, these work for some UDDI instances.

The advanced panel allows additional properties to be configured, such as proxy addresses and security. Information on these properties may be found on the WSPeer website (`www.wspeer.org`).

3.2.2 Discovering Web Services

Once that a UDDI repository has been configured, web services can be discovered very simply using the *Services→Discover Services* menu option. This option displays a dialog prompting for the name of the service to search for. This can either be the exact name or can include % as a wildcard character. For example, 'C%' will search for all web services beginning with the letter C.

When a web service is discovered, tools representing each of the operations on that web service are inserted into the *Web Services* package in the tool tree, alongside the existing locally available tools, as shown in Figure 3.2.

3.2.3 Importing Web Services

If a web service is not listed in a UDDI repository, then it can be imported directly into Triana from its WSDL description. This is done through selecting the *Services→Import Service* menu option, which causes a dialog requesting the service location to be displayed. This service location is the full http address of the WSDL document specifying the web service to be imported[1]. The web service at the service location will be imported into the the *Web Services* package in the tool tree.

3.2.4 Conncecting Web Services

Once a web service has been discovered/imported and appears in the user's tool tree, it can be instantiated by dragging the tool onto the main workspace (in the same way locally available tools are instantiated). Web services tasks appear in red on the workspace.

Each input node on an instantiated web service task represents an element of the input message for that operation, and each output node represents an output message element. Information on the required input/output types for a web service is displayed when the mouse is hovered over the task.

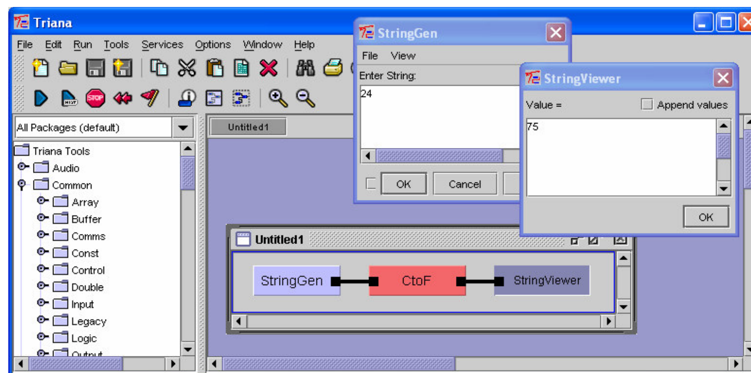

Figure 3.3: Using StringGen and StringViewer to provide input to and display the output from a temperature conversion web service.

Locally available tools can be used to provide the input to and display the output from web services. Two useful local tools are Common.String.StringGen

[1]For example, an web service interface to Altavista's BabelFish language is located at `http://www.xmethods.net/sd/2001/BabelFishService.wsdl`

and Common.String.StringViewer, which are used to generate string input and display string output respectively. These tools can also be used to input/output standard numerical data types (int, double etc.) as Triana automatically converts to/from the required type. In Figure 3.3 we show StringGen and StringViewer being used as input and output for a temperature conversion service. Other local tools can also be used as long as their output/input type is compatible with the type required by the web service, or alternatively the output from one web service can be directly piped to another. We look at handling complex data types in Section 3.2.6.

A workflow containing web services is executed as for a standard Triana workflow (i.e. by pressing the run button).

3.2.5 Bible Translation Example

In this section we demonstrate the creation of a simple bible translation workflow using third-party web services. This example uses the XMethods UDDI repository, so the following UDDI inquiry and publish addresses should be specified in the configuration (see Section 3.2.1):

Inquiry - `http://uddi.xmethods.net/inquire`
Publish - `https://uddi.xmethods.net/publish`

The two web services we wish to use are BabelFish[2], an interface to AltaVista's Babelfish service, and BibleVerses, a web service for extracting verses from the bible. The easiest way to import these web services is using the *Services→Disover Services* menu option, and then specifying 'B%' in the Discover Service dialog (this queries the UDDI for all tools beginning with B).

Once the BabelFish and BibleVerses web services have been discovered, they will appear as tools in the Web Services toolbox on the tool tree. Each of these services should be dragged onto the workspace, along with local StringGen and StringViewer tools, to create the workflow shown in Figure 3.4. StringGen and StringViewer are in the Common.Input and Common.Output packages respectively.

In this workflow StringGen provides the input for BibleVerses. If we double-click on StringGen and enter 'Genesis 1:1-7' in the input dialog, then this input

[2]Online documentation for BabelFish and BibleVerses can be found at `www.xmethods.net`.

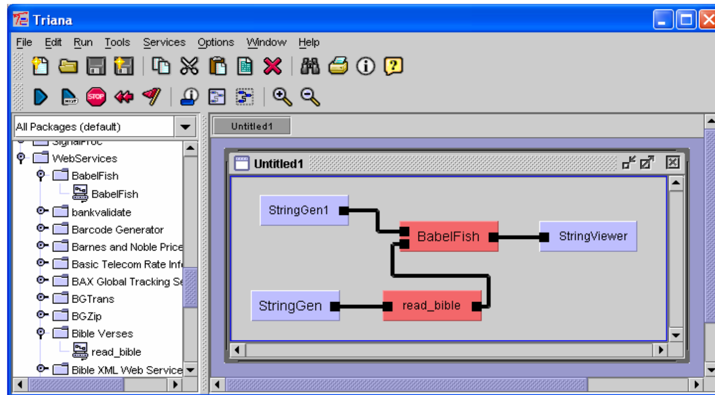

Figure 3.4: A simple bible translation workflow.

will cause BibleVerses to extract the first seven verses from the bible (Genesis chapter 1, verses 1 to 7). The output from BibleVerses is used to provide the second input for BabelFish, which is the text that is translated. The first input to BabelFish is provided by StringGen1. This is the languages that the text should be translated from/to. Using StringGen1 to specify 'en_fr' indicates we wish to translate from English to French. The output from BabelFish is sent to StringViewer.

Pressing the play icon on the tool bar will run the bible translation workflow we have created, and hopefully the 'Genesis 1:1-7' extract from the bible, translated into French, will be displayed in StringViewer (double-click on StringView view the result).

3.2.6 Complex Data Types

In Triana there are two ways to handle web services that require complex data types: use the dynamic web service type generator and viewer, or generate static type classes and create custom tools.

The input to a web service that requires a complex type can be automatically generated using the WSTypeGen tool, which resides in the Common.WebServices toolbox. Similarly, the complex output from a web service can be viewed using the WSTypeViewer tool, which is also found in the Common.WebServices toolbox. To use these two tools simply connect them to the web service task (as shown in Figure 3.5). The act of connecting them to the web service will cause a form for inputting/viewing the complex type to be dynamically generated. This form can be accessed by either double-clicking on the WSType-

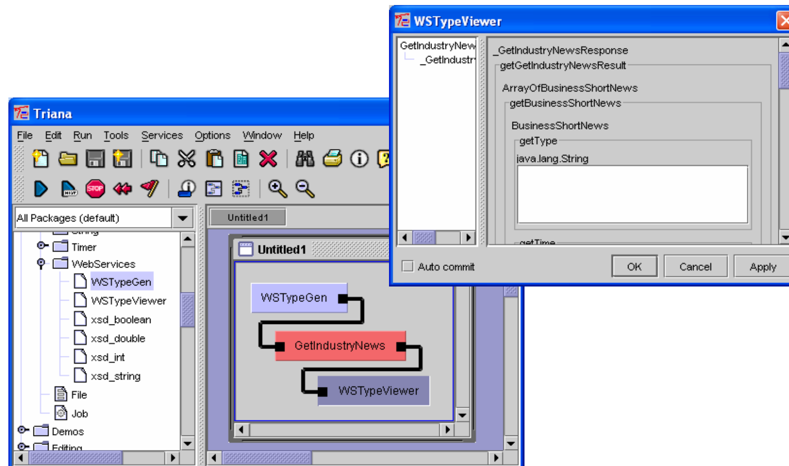

Figure 3.5: Generating/viewing complex data types using WSTypeGen and WSTypeViewer.

Gen/WSTypeViewer task, or by right-clicking and selecting *Properties* from the pop-up list.

Although WSTypeGen and WSTypeViewer are useful for testing web services that use complex types, for more long-term solutions it is generally required that static classes are generated for the complex types. Static type classes can easily be generated using a utility such as Apache's WSDL2Java[3]. Utilities such as WSDL2Java parse the WSDL description of the web service and generate a set of Java classes for the web service and the types used within that service. It is easiest if these classes are created in the same location as the tools that will access them (same Java base directory).

Once static classes have been created for the complex types, custom Triana tools for populating the types with data or converting between types must be created. These tools should be able to access the complex type classes in the same way as for any Java class. We describe creating custom Triana tools in section 5.1.

3.2.7 Deploying Web Services

As well as using external web services, Triana provides the mechanism for deploying user workflows as fully functions web services. These deployed web services can either be used within a local Triana workflow, allowing some of

[3]See `http://ws.apache.org/axis/java/user-guide.html`

the processing to be handled by a remote resource, or by third-party users. It should be noted that third-party users *do not* have to use Triana to access these web services.

In order to deploy web services Triana must first be configured with a UDDI to which you have both publish and inquiry access (see Section 3.2.1). This includes configuring the trust key store if required. Secondly, Triana launcher services must be run on the machine(s) that will host the deployed services. To do this Triana should be installed on those machines and the following command executed from the command-line:

```
TrianaService -ws
```

A web service can be created from either a single task or a group of tasks (see Section 2.3.2). To deploy the task/group task as a web service right-click on the task and select *Create Service*. This will cause a dialog to appear with a list of the locations where the web service can be hosted (as shown in Figure 3.6). This list will include the available launcher services (see above) along with a 'Local Service' option. Creating a local service means that the web service is hosted within the local Triana as opposed to on a remote site. Note that it can take a while for Triana to discover the available launcher services from UDDI; they automatically appear in the list once discovered.

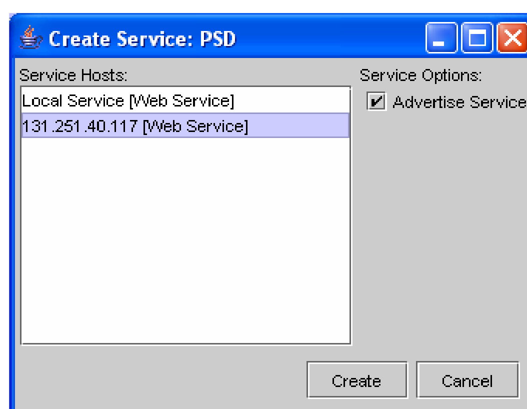

Figure 3.6: Dialog for deploying a task/group task as a web service.

Once a host has been chosen, Triana will attempt to deploy a web service on that host. While this process is taking place, the task in the workflow will appear with red stripes. The task becoming completely red indicates that the deployment has been successful and that the web service is ready to be used. As mentioned before, web services deployed using Triana can be used either within Triana or by third-party users.

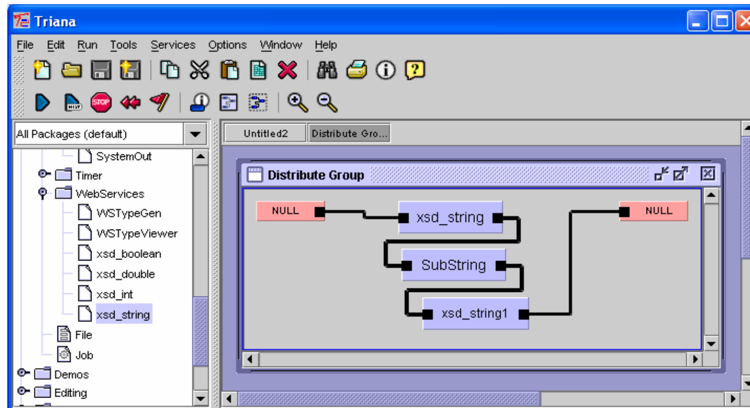

Figure 3.7: Using xsd tools to ensure standard input/output XML types are used when deploying a web service.

The input and output data types for deployed web services are automatically determined by Triana. If the input/output types from the task being distributed coincide with a standard XML type (e.g. java.lang.String, java.lang.Integer etc.) then that standard XML type is assigned. For non-standard types Triana assigns a string as the input/output type and assumes that the data received by the web service will have been serialized to a string using JSX. Such web services can be seamlessly handled by Triana but will be difficult for third-party users to use. It is recommended that standard types are used whenever possible, and that the tools xsd_string, xsd_double etc. are appended to the start of the workflow subsection being distributed to ensure the correct standard type is used, as illustrated in Figure 3.7. These tools can be found in the Common.WebServices toolbox.

3.3 P2PS

3.4 GAT

Chapter 4

Applications and Case Studies

4.1 Data Analysis with Triana

4.2 Audio Processing with Triana

Chapter 5

Extending Triana

5.1 Writing your own tools

5.1.1 Using the Tool Wizard

5.2 Advanced Tool Techniques

In this section we look at some more advanced techniques that a tool developer may wish to use.

5.2.1 Showing and Hiding a Unit's Parameter Panel

Problem

You want to simulate the user double clicking on a task to display the unit parameter panel.

Solution

Call the method `public void showParameterPanel()` to display the unit's parameter panel and `public void showParameterPanel()` to hide the panel again. These methods are inherited from the `Unit` superclass.

Discussion

5.2.2 Pausing Unit Execution

Problem

You want to pause the execution of your unit programmatically until some event happens.

Solution

Pause the thread that is running your unit and interrupt on your desired event to resume processing.

Discussion

Say for instance that you wish to pause the execution of your unit, between the point in the main `process()` method where the unit gets input from its input node, and the point where it outputs the result. The execution should halt until a particular parameter has been updated.

The preferred mechanism for doing this is to cause the current thread to sleep, interrupting the thread on the parameter being updated.

```
package UserGuide;

import triana.unit.Unit;

/**
 * An example unit that pauses for a parameter to be updated
 */
public class PauseExample extends Unit {

    // parameter data type definitions
    private String someValue;

    // private reference to the current thread
    private Thread currentThread;

    // flags
    private boolean stopped;
    private boolean selecteddone;

    /*
     * Called whenever there is data for the unit to process
     */
    public void process() throws Exception {
        java.lang.Object input = (java.lang.Object) getInputAtNode(0);

        stopped = false;
        selecteddone = false;
```

```
        // Insert main algorithm for PauseExample
        currentThread = Thread.currentThread();

        while (!(selecteddone || stopped)) {
            try {
                Thread.sleep(Long.MAX_VALUE);
            }
            catch (InterruptedException except) {
            }
        }
        output("I'm finished");
    }

    /**
     * This is called when the network is forcably stopped by the user. This should be over-ridde
     * with the desired tasks.
     * <p/>
     * The current thread needs to be interupted in the case of the user halting the execution.
     */
    public void stopping() {
        super.stopping();
        stopped = true;
        currentThread.interrupt();
    }


    /**
     * Called when the unit is created. Initialises the unit's properties and parameters.
     */
    public void init() {
        super.init();

        // Initialise node properties
        setDefaultInputNodes(1);
        setMinimumInputNodes(1);
        setMaximumInputNodes(1);

        setDefaultOutputNodes(1);
        setMinimumOutputNodes(0);
        setMaximumOutputNodes(Integer.MAX_VALUE);

        // Initialise parameter update policy and output policy
        setParameterUpdatePolicy(PROCESS_UPDATE);
        setOutputPolicy(CLONE_MULTIPLE_OUTPUT);

        // Initialise pop-up description and help file location
        setPopUpDescription("An example unit that pauses for a parameter to be updated");
        setHelpFileLocation("PauseExample.html");
```

```
    // Define initial value and type of parameters
    defineParameter("someValue", "", USER_ACCESSIBLE);

    // Initialise GUI builder interface
    String guilines = "";
    guilines += "Choose one $title someValue Choice [choice1] [choice2]\n";
    setGUIBuilderV2Info(guilines);
}

/**
 * Called when the unit is reset. Restores the unit's variables to values specified by the
 * parameters.
 */
public void reset() {
    // Set unit variables to the values specified by the parameters
    someValue = (String) getParameter("someValue");
}

/**
 * Called when the unit is disposed of.
 */
public void dispose() {
    // Insert code to clean-up PauseExample (e.g. close open files)
}


/**
 * Called a parameters is updated (e.g. by the GUI)
 */
public void parameterUpdate(String paramname, Object value) {
    // Code to update local variables
    if (paramname.equals("someValue")) {
        someValue = (String) value;
        selecteddone = true;
        if (currentThread != null)
            currentThread.interrupt();
    }
}


/**
 * @return an array of the types accepted by each input node. For node indexes not covered th
 *         types specified by getInputTypes() are assumed.
 */
public String[][] getNodeInputTypes() {
    return new String[0][0];
}
```

```
    /**
     * @return an array of the input types accepted by nodes not covered by getNodeInputTypes().
     */
    public String[] getInputTypes() {
        return new String[]{"java.lang.Object"};
    }


    /**
     * @return an array of the types output by each output node. For node indexes not covered the
     *         types specified by getOutputTypes() are assumed.
     */
    public String[][] getNodeOutputTypes() {
        return new String[0][0];
    }

    /**
     * @return an array of the input types output by nodes not covered by getNodeOutputTypes().
     */
    public String[] getOutputTypes() {
        return new String[]{"java.lang.Object"};
    }

}
```

5.3 Triana as a Workflow Editor

Chapter 6

Demos