# OODSS Tutorial

Here we give an example service which demonstrates how to use the OODSS library for creating a network service. The service shown is a History Echo Service in which a server takes requests and then simply responds with a message containing the request message just received in addition to the client's previous request message.

We first show the example message declarations, embedded with `ecologylab.xml` metalanguage, then the creation of a translation scope, which specifies what services are provided by a server. We conclude by walking through the flow of control between a Java OODSS client and server, showing the messages that will be passed through the network and resulting program output.

## Request Messages

We define our `RequestMessage` subclass first. This message carries a `String` to be echoed by the server (`newEcho`) from a client application. It stores `newEcho` in the `ECHO_HISTORY` binding in the client session scope after retrieving its previous value (if any).

This service does not use the application object scope, because each client only needs information about its own context. One could imagine a broader service, that echoed the previous message sent by any client, which would store the new message in the application object scope instead of the client session scope.

```
public class HistoryEchoRequest extends RequestMessage
{
	/*
	 * ECHO_HISTORY binds to a String representing the previous string
	 * sent from the client application
	 */
	final static String ECHO_HISTORY = "ECHO_HISTORY";
	@xml_attribute String newEcho;

	/**
	 * Constructor used on server.
	 * Fields populated automatically by ecologylab.xml
	 */
	public HistoryEchoRequest() {}

	/**
	 * Constructor used on client.
	 * @param newEcho a String that will be passed to the server to echo
	 */
	public HistoryEchoRequest(String newEcho)
	{
		this.newEcho = newEcho;
	}

	/**
	 * Called automatically by LSDCS on server
	 */
	@Override public HistoryEchoResponse performService(Scope cSScope)
	{
		/*
		 * retrieve, from the session object registry,
		 * the last-sent string
		 */
		StringBuffer prevEcho;
		String prevEchoTmp;

		/*
```

```
             * Retrieve, from the object registry,
             * the last-sent string. In the case that
             * we are running a server in which echo_history
             * has already been instantiated in the application scope
             * then we will use and update a application level history value.
             */
            if(cSScope.get(ECHO_HISTORY) == null)
            {
                  /*
                   * In the case that the sever hasn't then we instantiate
                   * our own, in the session scope.
                   */
                  cSScope.put(ECHO_HISTORY, new StringBuffer());
            }

            prevEcho = (StringBuffer) cSScope.get(ECHO_HISTORY);

            // Temporarily store the previous echo string
            prevEchoTmp = new String(prevEcho);

            /*
             *  replace it with the new one
             */
            prevEcho.replace(0,prevEcho.length(),newEcho);

            /*
             * use both messages to create a new response
             */
            return new HistoryEchoResponse(prevEchoTmp, newEcho);
      }
}
```

**Response Messages**

We now define our `ResponseMessage` subclass, an instance of which is returned by the `performService(…)` method of `HistoryEchoRequest`. This class represents the result of the service execution on the server, and an instance of it will be serialized and returned to the client. Because we are using a Java OODSS client, this message's `processResponse(…)` method will be called automatically, printing both the previous (`prevEcho`) and most recent (`echo`) strings sent to the server.

```
public class HistoryEchoResponse extends ResponseMessage
{
      @xml_attribute String echo;
      @xml_attribute String prevEcho;

      /**
       * Constructor used on client.
       * Fields populated automatically by ecologylab.xml
       */
      public HistoryEchoResponse() {}

      /**
       * Constructor used on server
       * @param prevEcho a String which indicates the previous echo
       * received by the server.
       *
       * @ echo echo of the string just received by the History server.
       */
```

```
	public HistoryEchoResponse (String prevEcho, String echo)
	{
		this.prevEcho = prevEcho;
		this.echo = echo;
	}

	/**
	 * Called automatically by OODSS on client
	 */
	@Override public void processResponse(Scope appObjScope)
	{
		System.out.println("2nd To Last Message: " + prevEcho +
				"\nLast Message: " + echo);
	}

	/**
	 * Checks that the message does not have an error condition,
	 *  for now we assume it doesn't */
	@Override public boolean isOK()
	{
		return true;
	}
}
```

**Creating a Translation Scope**

The History Echo Services server and client each need a translation scope. We define a single `TranslationScope` instance that we can use for both. In addition to history echo messages, the application must translate messages for establishing a session. The translations for such messages are normally automatically added to the scope, but we will demonstrate manual assembly. In practice, a class is usually defined for a service which has a static `get()` accessor which will perform a similar process, returning a translation scope for the associated service.

```
	Class[] historyEchoClasses = { HistoryEchoRequest.class,
	                               HistoryEchoResponse.class };
	/*
	 * Get base translations with static accessor
	 */
	TranslationScope baseServices = DefaultServicesTranslations.get();

	/*
	 * compose translations, to create the “histEchoTrans”
	 * space inheriting the base translations
	 */
	TranslationScope histEchoTranslations =
		TranslationScope.get("histEchoTrans",
		                     historyEchoClasses,
		                     baseServices);
```

**Server/Client Instantiation**

We now initialize an instance of DoubleThreadedNIOServer. We must first instantiate a Scope for the server to use as an application/client scope. The constructor also requires a list of all local host addresses. After the instance has been constructed, we may just start the server which will then handle all incoming requests.

```
	/*
	 * Create a scope for the server to use as an application scope
	 * as well as individual client session scopes.
	 */
```

```
Scope applicationScope = new Scope();

/* Acquire an array of all local ip-addresses */
InetAddress[] locals = NetTools.getAllInetAddressesForLocalhost();

/* Initialize and start the server so that it can
 * accept incoming connections.  We must also specify the idle
 * connection timeout, and MTU for the server.
 */
DoubleThreadedNIOServer historyServer =
      DoubleThreadedNIOServer.getInstance(2107,
      locals,
      histEchoTranslations,
      applicationScope,
      idleTimeout,
      MTU);

historyServer.start();
```

If we want the server to act as a global history echo server we may instantiate the echo history object within the application scope. In this case the `HistoryEchoRequest` message will use an application level value for the message history causing the server to always return the last received message by any client.

```
/*
 * Initialize the ECHO_HISTORY registry in the application scope
 * so that the performService(...) of HistoryEchoRequest modifies
 * the history in the application scope.
 */
applicationScope.put(HistoryEchoRequest.ECHO_HISTORY,
                     new StringBuffer());
```

In order to initialize the client we must also specify a TranslationScope and an application scope for the client application to use. In addition, we must also give the address and port number of the server we wish the client to connect to. Once the client is initialized we must call the connect method in order for the client to create a socket connection to the server. The server responds to the new connection by instantiating a `ClientSessionManager` for the client. The client then automatically sends a request for a new session, and the server responds with a session token.

```
Scope clientScope = new Scope();

client = new NIOClient(serverAddress, portNumber,
                       histEchoTranslations,clientScope);

client.connect();
```

All that is left at this point is to read messages from STDIN, create instances of the HistoryRequestMessages with the given input and have the client send the message to the server.

```
while(true)
{
      String input = scan.nextLine();

      if(input.trim().toLowerCase().equals("exit"))
            break;

      HistoryEchoRequest echoRequest = new HistoryEchoRequest(input);
      try
```

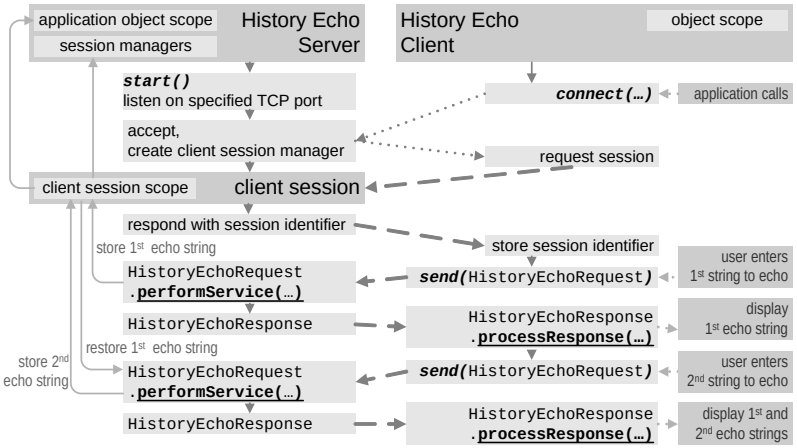


**Figure 1. Echo server with history flow of control.** Flow moves from top to bottom, alternating sides. The example application echoes the client's input, along with the previous input. Bold entries are method calls; underlined calls are overridden by the developer to define functionality and invoked automatically by LSDCS; italicized calls are invoked manually by the LSDCS application authored by the developer.

```
        {
                client.sendMessage(echoRequest);
        }
        catch (MessageTooLargeException e)
        {
                System.err.println("The message you sent was too large!");
                e.printStackTrace();
        }
}

client.disconnect();
```

## Message Flow Explanation

In order to make clear the exchanges in an OODSS little semantic web service, we will walk through the flow of control using a Java OODSS client and server (see **Figure 1.**).

Once the client application logic reads a string from the user, it then passes the message to the constructor of `HistoryEchoRequest`. The instance of `HistoryEchoRequest` is transmitted to the server with the `send(…)` method, which automatically serializes the instance to XML using `ecologylab.xml`. An HTTP-like header is created containing metadata about the message (content length, etc.) and is pre-pended. The resulting bytes are then sent over the network to the server.

We assume the client provides the string "V. Bush". Because we used `@xml_attribute` in `HistoryEchoRequest` to create compact XML representing a scalar `String` (`new_echo`), the XML exchange begins:

```
<history_echo_request new_echo="V. Bush"/>
```

The server reads the incoming header bytes and decodes them. Parsing the header indicates to the server how to decode the remainder of the message and its length. The incoming bytes are translated back into a XML, which is deserialized by `ecologylab.xml` into a `RequestMessage` instance populated with the client-specified data (in this case, a `HistoryEchoRequest`).

The client manager on the server calls the message's `performService(…)` method using its client session scope. This extracts the previous string sent by the client (`null` in the first execution) from the session scope and stores it. It then places the new string into the scope for later, and instantiates a `HistoryEchoResponse` containing both strings.

*On the first execution,* `previousEcho` *will be* `null`*, and thus will not be serialized in the XML in the server's response:*

```
<history_echo_response echo="V. Bush"/>
```

*The client continues by sending "T. Berners-Lee":*

```
<history_echo_request new_echo="T. Berners-Lee"/>
```

*The server then sends back both of the strings, after retrieving the previous one from the client's session scope:*

```
<history_echo_respnse echo="T. Berners-Lee" prev_echo="V. Bush"/>
```

As with `HistoryEchoRequest` on the client, the `HistoryEchoResponse` is serialized using `ecologylab.xml`, a header of metadata is prepended, encoded, and sent over the network. As with the server, the client then decodes this message and deserializes it into a `HistoryEchoResponse` instance. Its `processResponse(…)` method is invoked, printing both of the user's strings. *The final output from our example will read:*

```
V. Bush
T. Berners-Lee
```

**Notes:**

More information concerning ecologylab.xml may be found at http://www.ecologylab.net/research/xml/. Also, access to the entire ecologylab fundamental project source is available through anonymous SVN access (username: anonymous, no password) here: http://www.csdl.tamu.edu/ecologylabsvn/ecologylabFundamental. The source for this tutorial is located under trunk/ecologylabFundamental/ecologylab/tutorials/.