

# The Soapmill Manual

version 0.15.1

Eric Kow

30 April 2004

# Contents

0.1	Introduction . . . . .	3
0.2	How to read this document . . . . .	3
<b>I</b>	<b>User's Manual</b>	<b>4</b>
<b>1</b>	<b>Basic Manual</b>	<b>5</b>
1.1	Before you start . . . . .	5
1.2	Getting started . . . . .	6
1.3	Starting and stopping your agent . . . . .	6
1.4	Receiving incoming data . . . . .	7
1.5	Sending outgoing data . . . . .	8
1.6	Dealing with Errors . . . . .	12
<b>2</b>	<b>Advanced Manual</b>	<b>15</b>
2.1	Receiving responses . . . . .	15
2.2	Configuring your agent . . . . .	18
2.3	Loading Files . . . . .	18
2.4	Thread Safety . . . . .	19
2.5	Logging . . . . .	19
<b>3</b>	<b>Packaging Manual</b>	<b>21</b>
3.1	Version numbers . . . . .	21
3.2	The directory WEB-INF . . . . .	21
3.3	File etc/log4j.snippet (optional) . . . . .	23
3.4	Data directories . . . . .	23
3.5	The package . . . . .	23
<b>4</b>	<b>Integration Manual</b>	<b>24</b>
4.1	Install the Soapmill . . . . .	24
4.2	Install the agents . . . . .	24
4.3	Configure multiple servers . . . . .	25
4.4	Run the server(s) . . . . .	25
4.5	About upgrading . . . . .	25
<b>II</b>	<b>Conclusion</b>	<b>26</b>
<b>5</b>	<b>Questions and Problems</b>	<b>27</b>
5.1	Tips and Clarifications . . . . .	27
5.2	Troubleshooting . . . . .	28

---

<b>6 Conclusion</b>	<b>29</b>
6.1 Acknowledgments . . . . .	29
<b>A Programmer's Glossary</b>	<b>30</b>
<b>B Example message types</b>	<b>31</b>
B.1 MPEG-7 types . . . . .	31
B.2 MMIL types . . . . .	31

## 0.1 Introduction

The Soapmill library was written to make it faster and easier to build complex applications out of a common supply of reusable parts. Other frameworks (Jade, OAA or GATE) have similar goals, but they are not mutually compatible, and thus divide the pool of otherwise available parts. We propose to overcome these disagreements through the enthusiastic use of standards. This library shall aim to demonstrate that standards such as SOAP and XML *can* be used in a simple and effective manner.

## 0.2 How to read this document

This document is addressed at two different audiences. Which and how many of these are you?

### 0.2.1 Integrators

An **integrator** can be seen by the developer as an end-user. Integrators create complex applications by putting together the agents written by other developers. They also organise testing of the combined application and its internal components. Integrators should read both *The Soapmill Report* [6] and the integrator's manual in chapter 4.

### 0.2.2 Developers

A **developer** writes the software components that integrators put together. Developers are also recommended to read *The Soapmill Report* before using this manual. Here is one way you might tackle this manual a developer:

1. **Learn basic features (chapter 1)**. Next, follow the basic manual to write a simulator for your agent. Starting with a simulator will simplify your life because it separates Soapmill problems from agent problems. Since you could always fill in your agent code later, the creation of a simulator does not actually entail any duplicate effort.
2. **Package your software (chapter 3)**. Follow the instructions in the packaging manual (chapter 3). They will show you how to put your simulator into a nice package that will make it easy to export your agent to other people, and will simplify the integration process. The packaging process may seem like a pain, but you will probably only have to do it once, and can carry over the necessary pieces into your real agent.
3. **Test integration (chapter 4)**. Write or obtain simulator packages for the agents for your component could interact with. Follow the integrator's manual (4) and test those agents with your own. Once you have worked out all the integration problems, and have verified that your simulator combines nicely with other agents (or simulators), you are ready to start working.
4. **Learn advanced features (chapter 2)**. Flip through the advanced manual and take note of any features you might need to use. If you require advanced features, you should first extend and test your simulator to make sure you understand how these features work.
5. **Flesh it out**. Finally, return to the manuals (chapters 1 and 2) and start filling out the functionality.

**Part I**

**User's Manual**

# Chapter 1

## Basic Manual

### 1.1 Before you start

#### Key ideas

In Soapmill, your software communicates with other software by the exchange of XML documents through one-way communication. All documents are identified with a **message types** so that the receiver can choose to use or ignore it.

#### Not using Java?

This document was written for the Java implementation of Soapmill. If you are not using Java, you need to find or implement the Soapmill in your favourite language. See *Implementing Soapmill* for details. You can also write wrapper software, but this approach is costlier in the long run.

#### Java

Are you experienced with writing software which uses XML? If you are not, I recommend that you go away, read up on sites like Sun's Java XML tutorial, practice writing some small programs for a week and then pick up this manual again.

## 1.2 Getting started

The first task is to create a subclass of `SoapmillAgent` and implement the **callback method** named `subscriptions`. This method will be invoked by soapmill to find out what message types you are interested in receiving.

```
import soapical.soapmill.SoopmillAgent;
...

public class ExampleAgent extends SoapmillAgent {

    public ExampleAgent() throws SoapmillException {
        super();
        // your initialisation code here
    }

    public void subscriptions() {
        subscribe("http://www.miamm.org/mmil/semantic_representation");
        subscribe("http://www.miamm.org/mmil/visualisation_request");
    }
    ...
}
```

### Notes

1. The Soapmill only uses a single instance of your agent. To retrieve this instance, for example, when unit-testing, call the `instance()` method.
2. To implement a universal agent, you can call `subscribeAll` instead of `subscribe`.

## 1.3 Starting and stopping your agent

Your agent must also implement methods which Soapmill can call when it wants to initialise or shut down your agent.

### 1.3.1 start

The `start` method allows you to run any initialisation code for your agent. It is not recommended that you place this code in your constructor method, because your agent might only be constructed once, whereas, it might get started several times.

### 1.3.2 stop

The `stop` method allows you to do any cleanup before your agent shuts down. This would be a good time to write any data to disk and to shut down any peripherals that are connected to your agent.

### 1.3.3 start/stop example

```
public class ExampleAgent extends SoapmillAgent {
    ...

    public void start() throws SoapmillException {
        // HERE: any initialisation code for your agent
    }

    public void stop() throws SoapmillException {
        // HERE: any cleanup code
    }

    ...
}
```

#### Notes

1. If you need to pass arguments to the Java virtual machine, or if your agent needs to load some files, see section 2.2.
2. The stop method is not yet called by anything, but you should implement it anyway.

## 1.4 Receiving incoming data

### 1.4.1 Simplest case

To receive messages from the outside world, you should implement the **callback method** named `receive`. If the Soapmill knows that your agent is interested in messages like the incoming message, it will call this method and pass it a `SoapmillMessage`.

You can get the contents of the message through its `getContents()` method, which returns a `java.io.Reader` with XML data. You can do anything you want with this data. Most likely, your first step would be to parse it (for example, through a SAX or a DOM parser or with something like Castor [2] or JAXB [3]).

Let's try a simple example.

```
public class ExampleAgent extends SoapmillAgent {
    ...

    public void receive(SoapmillMessage message)
    {
        MyParser parser = new MyParser();
        ExampleThing thing = parser.parse( message.getContents() );
        doSomethingClever(thing);
    }

    ...
}
```

#### Notes

1. The Soapmill processes incoming messages one at a time. The Soapmill will call `receive` method and wait for it to finish before it calls it again. If this behaviour is not acceptable, you will have to create your own Threads and deal with thread-safety issues yourself.

### 1.4.2 Different kinds of events

If your agent is supposed to handle more than one kind of message, you might consider calling the `getType()` method to avoiding having to wade through its contents.

```
public class ExampleAgent extends SoapmillAgent {
    final static String SEMANTIC_REPRESENTATION =
        "http://www.miamm.org/mmil/semantic_representation";

    ...

    public void receive(SoapmillMessage message)
    {
        // parse the message contents
        MyParser parser = new MyParser();
        ExampleThing thing = parser.parse( message.getContents() );

        // get the metadata
        String type = message.getType();
        if ( SEMANTIC_REPRESENTATION.equals(type) ) {
            doSemanticRepresentation(thing);
        } else {
            doSomeOtherThing(thing);
        }
    }

    ...
}
```

#### Notes

1. Look in the Javadoc under class `SoapmillMessage` for more details.
2. We plan to have more sophisticated access to message metadata in the future.

## 1.5 Sending outgoing data

Perhaps if your component was some kind of log agent or history agent, it would be acceptable to only do things on the receiving end. Most likely, however, you are also interested in *sending* messages to other components in the architecture. Sending messages is a two-step process: getting the data, and publishing it.

### 1.5.1 Get the data

The first step is to provide the contents for a message as XML through a `java.io.Reader`. The following paragraphs provide examples of the many ways you could populate a message.

**... with an XML String**

You might be tempted to hand-write the XML to a `String` or preferably a `StringBuffer`. This could cost you much more effort and debugging time in the long run, so it would be wiser to explore the many other means. If you absolutely insist on strings, then use `StringReader`.

```
public class ExampleAgent extends SoapmillAgent {
    ...

    // populate the message
    StringBuffer outgoingMMIL = new StringBuffer();
    outgoingMMIL.append("<mmil:mmilComponent" +
        " xmlns:mmil=\"http://www.miamm.org/mmil\">");
    outgoingMMIL.append("<!-- blah blah -->");
    outgoingMMIL.append("</mmil:mmilComponent>");

    // prepare the message for publishing
    Reader contents = new StringReader(outgoingMMIL);

    // NEXT: we publish the message

    ...
}
```

**... with SAX in reverse**

A slightly safer way to output XML would be to use the SAX parser in reverse, by directly invoking its callback methods. Do this by combining the following helper classes:

1. `ReaderWriter` - writes data to a `java.io.Reader`
2. `SpecificXMLWriter` - simplifies namespace generation

```
import soapical.util.ReaderWriter;
import soapical.util.SpecificXMLWriter;

class MMILWriter extends SpecificXMLWriter {
    public MMILWriter(Writer out) {
        super(out, "http://www.miamm.org/mmil", "mmil");
    }
}

public class ExampleAgent extends SoapmillAgent {
    final public String MMIL_COMPONENT = "mmilComponent";

    ...

    // populate the message
    ReaderWriter messageWriter = new ReaderWriter();
    SpecificXMLWriter handler = new MMILWriter(messageWriter);

    // note: you most likely want to make a seperate method out of this
    handler.startDocument();
    handler.startElement(MMIL_COMPONENT);
    handler.endElement(MMIL_COMPONENT);
    handler.endDocument();

    // prepare the message for publishing
    Reader contents = messageWriter.toReader();

    // NEXT: we publish the message

    ...
}
```

### ... with a DOM Document

If your agent stores the outgoing data in a DOM representation, you might take advantage of the Soapical Soapkit (included with Soapmill):

```
import soapical.util.Soapkit;

public class ExampleAgent extends SoapmillAgent {
    ...

    // prepare the message for publishing
    String messageStr = Soapkit.toString(document);
    Reader contents = new StringReader(messageStr);

    // NEXT: we publish the message

    ...
}
```

**... from Castor or similar**

Similarly, you might use an XML binding framework such as Castor to output the data from automatically generated source code. You would also use the `ReaderWriter` to do this job.

```
import org.exolab.castor.xml.Marshaller;
import soapical.util.ReaderWriter;
import org.xml.sax.Reader;

public class ExampleAgent extends SoapmillAgent {
    ...

    ReaderWriter writer = new ReaderWriter();
    Marshaller.marshal(mmil, writer);
    // prepare the message for publishing
    Reader contents = writer.toReader();

    // NEXT: we publish the message
    ...
}
```

**... from an external source**

One possibility is that the outgoing data might already be available from some external source such as file or a Socket connection from elsewhere. The most convenient thing is to use the `InputStream` or a `Reader` that comes from that source.

```
import java.io.FileReader;

public class ExampleAgent extends SoapmillAgent {
    ...
    // populate the message
    Reader content = new FileReader(outgoingData);

    // NEXT: we publish the message
    ...
}
```

**1.5.2 Publish the data**

Next, you'll want to publish a `SoapmillMessage` containing your data and some metadata so that other agents know what to do with it.

**Construct SoapmillMessage**

The `SoapmillMessage` constructor takes two arguments:

1. the contents of the message (`Reader`)
2. the **message type** (semantic representation, visualisation request, etc).

The message types are merely Strings. One good strategy to assure that these types remain unique throughout time is to make each one a URI (see appendix ??). In the MIAMM project, we base our message types on the namespace URIs of MMIL and MPEG-7.

```
public class ExampleAgent extends SoapmillAgent {
    final static String WORD_GRAPH =
        "urn:mpeg:mpeg7:soapmill:word_graph";
    final static String MACUMBA_EVENT =
        "http://www.something.org/macumba_event";
    ...

    // PREVIOUSLY: we got a source of data

    // create the message
    SoapmillMessage message = new SoapmillMessage(WORD_GRAPH, contents);

    // NEXT: we publish the message
}
```

### Notes

1. See appendix B for some example message types.
2. If your message is a response to a previous message, you should use the alternate constructor, which takes the **message type** and the `SoapmillMessage` you are responding to.

### Publish the message

Once you have the message contents and its metadata, you invoke the `publish` method. This method will publish the message and return immediately. Responses to the message will be treated like regular messages unless you use the **await** mechanism in section 2.1.

```
public class ExampleAgent extends SoapmillAgent {
    // PREVIOUSLY: we defined the message type WORD_GRAPH
    //                and got a source of data

    // create the message
    SoapmillMessage message = new SoapmillMessage(WORD_GRAPH, contents);

    // publish the message
    publish(message);

    ...
}
```

## 1.6 Dealing with Errors

I expect that being a responsible programmer, you not only unit test your code, but have it peer-reviewed to discover hidden bugs. While your diligence shall certainly pay off in bountiful ways, alas, the occasional bug will find its way in. Tough luck. Your three options are to do nothing, to handle the error, or to report it.

### 1.6.1 Doing Nothing

The laziest and least desirable approach is to simply throw a `SoapmillException`. Soapmill will display an error message, plus a brief stack trace, so that you can go correct the error and write a unit test [1] to assure yourself that it never happens again.

```
public class ExampleAgent extends SoapmillAgent {
    ...

    try {
        doSomeStuff();
    } catch (NastyException ne) {
        String error = "my, what a nasty exception!";
        throw new SoapmillException(error, ne);
    }

    ...
}
```

#### Notes

1. You can eventually download a tool called “bpmonitor” which lets you keep track of your agents and signals any exceptions thrown.

### 1.6.2 Handling the Error

A more disciplined approach is to try and recover from the damage. If the error was minor enough, it might even be as simple as catching it, logging it and moving on.

```
public class ExampleAgent extends SoapmillAgent {
    ...

    try {
        doSomeStuff();
    } catch (VeryNastyException vne) {
        String error = "my, what a nasty exception!";
        logger.error(error);
        doSomeRecovery();
    }

    ...
}
```

### 1.6.3 Reporting the Error

If you expect other agents to be able to understand your errors, you should consider creating and reserving a message type for errors. Whenever an error comes up that you expect other agents to handle, you can publish it as a message. Naturally, other agents will need to subscribe to that message type.

```
public class ExampleAgent extends SoapmillAgent {
    ...

    try {
        doSomeStuff();
    } catch (VeryNastyException vne) {
        String error = "<error>my, what a nasty exception!</error>";
        SoapmillMessage message =
            new SoapmillMessage( "urn:macumba:example_error", error );
        publish( message );
    }

    ...
}
```

#### 1.6.4 Future Plans

I will look into using SOAP Faults as the format to use for reporting errors to other agents.

## Chapter 2

# Advanced Manual

### 2.1 Receiving responses

In the Basic Manual we assumed that agent communication was strictly asynchronous, that is, with simple one-way messages. But what can you do if you wanted some kind of synchronous communication, that is, if you wanted to have traditional two-way communication? Soapmill offers the **await** mechanism which simulates two-way communication by letting you pull the response from soapmill instead of hoping that it gets pushed to your agent.

#### 2.1.1 The await mechanism

To pull the response to message, we will use the `publishAndAwait` method instead of `publish`.

```
import soapical.soapmill.SoapmillMessage;

public class ExampleAgent extends SoapmillAgent {
    ...

    // PREVIOUSLY: we created a SoapmillMessage

    // publish the message with its metadata and wait for its response
    SoapmillMessage response = publishAndWait(message);

    // NEXT: we attempt to make use of the response
    ...
}
```

#### Notes

1. You can pull subsequent responses with the `awaitResponse` method.

#### 2.1.2 Using the response

Now that we have the response, what do we do with it? The response is in the form of a `SoapmillMessage`, so we treat as we would when receiving any normal event.

```

public class ExampleAgent extends SoapmillAgent {
    ...

    // PREVIOUSLY: we awaited the response

    // extract the contents from the response
    Reader responseXML = response.getContents();

    // make use of them any way you like
    Parser parser = new MyParser();
    ExampleThing thing = parser.parse(responseXML);
    doOtherExampleThing(thing);

    ...
}

```

### 2.1.3 Unexpected responses

Once you **await** a response, the Soapmill will assume that you want to handle all responses to the message with the same mechanism; however, you will only know about as many responses as you await. To await more than one response, we introduce the `awaitResponse` method:

```

// first response
response1 = publishAndAwait(message);
// second response
response2 = awaitResponse(message);
// third response
response3 = awaitResponse(message);

```

What happens if you receive a fourth response to the message? Nothing. It gets queued for the next time you call `awaitResponse` on that message. If you do not ever again await responses to that message, this is effectively the same as silently discarding responses, except that if you receive many responses to the message, and you don't await all of them, they will pile in the responses queue, eating up memory.

To avoid this, you should consider letting the soapmill treat responses as normal once you have received those which interest you.

```

// third response
response3 = awaitResponse(message);
// treat it normally from now on
setAwaitable(message, false);

```

This might be less than satisfactory, because you will still have deal with response, but within the regular `receive` framework. If you feel that it will very useful truly drop responses, please request it for future Soapmill releases.

### 2.1.4 Timing out

The **await** mechanism has a default timeout of five minutes. If there is no response within the timeout, you will receive a null result.

If you wish to change the timeout, you can pass a second argument specifying the number of milliseconds you wish to wait. See the API for details.

If you do not wish to timeout at all, and would rather wait indefinitely, you can also pass it a timeout of zero. However, this means that if the response never comes, your agent will be effectively hung. One way to prevent this possibility is to perform the `await` in a separate thread, which has the side benefit that your agent will also be able to simultaneously receive other messages and responses while waiting. This, however, includes the natural caveat that you will have to be careful to avoid the problems which plague any software trying to do more than one thing at once. Please learn how to deal with threads properly.

### 2.1.5 Three cases

Please make sure you have read section 2.1.4.

#### Handling one response

Call `awaitResponse` and make use of the `SoapmillMessage` it returns, as we saw in the previous sections.

```
import soapical.soapmill.SoopmillMessage;

public class ExampleAgent extends SoapmillAgent {
    ...

    // await the response to the message
    SoapmillMessage response = publishAndAwait(message);
    // make use of it
    doSomethingClever(response)
    ...
}
```

#### Handling several responses

If you can predict how many responses to expect, you should call `awaitResponse` a finite number of times. Each time you call `awaitResponse`, the current thread will wait for the next response to the message.

```
public class ExampleAgent extends SoapmillAgent {
    ...

    // await the first response to this message
    SoapmillMessage response1 = publishAndAwait(message);
    doSomethingClever(response1);

    // await the first response to this message
    SoapmillMessage response2 = awaitResponse(message);
    doSomethingClever(response2);
    ...
}
```

#### Handling many responses

If you do not know how many responses to expect, and if you want to receive them all, your first idea might be to sit in a loop and continue receiving responses, but this is problematic

because, your agent does not receive any other messages in the meantime; and if your loop never terminates, the agent is effectively dead. This is the same problem as discussed in section 2.1.4, but here the risk of this happening is significantly higher. Your options are either to never write such a loop, or to do so in a separate thread:

```
public class ExampleAgent extends SoapmillAgent {
    ...
    Thread t = new Thread() {
        while (! someExampleConditionSatisfied) {
            // await the a response to this message
            SoapmillMessage response = publishAndAwait(message);
            doSomethingClever(response);
        }
    };
    t.start();
    ...
}
```

### Notes

1. Be careful if you use multiple threads. See section 2.4.

## 2.2 Configuring your agent

Sometimes, you need to configure your agent at run time instead of resorting to hard-coded constants. You could either do this by Java system properties or with servlet initialisation parameters.

Servlet initialisation is a useful alternative to system properties, because servlet parameters are local to each agent, so you don't have to worry as much about naming collisions.

System properties are mostly supported for backwards compatibility, but they might also be useful if you want to take advantage of their globality or if your software is also meant to be used as a standalone application. See sections 3.2.1 (especially 3.2.1) for more details.

## 2.3 Loading Files

It is very reasonable that your agent might want to load some files while it was running. You can use the method `getAgentRoot` to access the document root of your agent.

```
public class ExampleAgent extends SoapmillAgent {
    ...

    loadFile(getAgentRoot(), "data/myfile.xml");

    ....
}
```

If you were using Soapmill before version 0.15, you might be wondering about the system properties we recommended beforehand. These are still supported for backwards compatibility, but you'll have to modify your agent packaging (see section 3.2.1), by replacing your `soapmill_prerun` script with a `web.xml` file, and replacing all references to `${PWD}` with `AGENT_ROOT`.

## 2.4 Thread Safety

The Soapmill only processes incoming messages (or responses) for your agent one at a time; therefore, you will not need to worry about making your agent thread-safe. However, if you wish to be able to receive multiple messages at the same time, you can do so (for example) by creating a new Thread every time `receive` is called. If you do so, please learn about the various safety issues which exist in any situation where you are trying to do more than one thing at the same time [4].

## 2.5 Logging

Every programmer, even those with the fanciest tools, probably uses the tried and true `printf` approach to debugging. But when doing so, do you find yourself commenting and uncommenting print statements, as you race around your code removing and re-entering "i was here" statements as they either become cluttersome or suddenly useful again? You might have even promised to yourself over and over again that you really ought to get around to learning to use that fancy debugger one day, but never got around to it.

Maybe there's an easier way to soothe your headaches: embrace logging! Logging takes just as little effort as your friend, the `println`, but many times as much flexibility. Ever wanted to change the verbosity on the fly? Sort out your messages so that Foo messages go to the Foo log and Bar messages go to the Bar log? How about enabling or disabling log messages for just that one package that was bother you?

```
class ExampleAgent {
    Logger logger = Logger.getLogger(ExampleAgent.class);

    public void doSomethingClever() {
        logger.debug("Enter: doSomethingClever");
        blahBlah();
        logger.debug("Blah blah reached!");
        if (error) {
            logger.error("Uh-oh! There was an error!");
        }
        otherBlah();
        logger.info("Something you should know about, sir!");
        logger.debug("Exit: doSomethingClever");
    }
}
```

Now we can get some serious debugging done. Let's say you've got the test version of the code and you really want to find out what it's doing. Just tell the log configuration file that you are interested in all log entries from "debug" on up. Now imagine scenario two, where after a long night's debugging, you finally prove that your code works exactly as it should, so you don't want to bother users with all sorts of weird trace messages. Just configure the log so that it only prints messages of type "error" or up.

### 2.5.1 Log4J

I hope this section has enticed you into using a logging tool. There are a couple of good loggers out there, and rather arbitrarily, I have selected Apache's `log4j` [5] for the Soapmill. If you are not already tied to your system, you should consider using this instead. That way, you won't have very much to do for configurability.

## 2.5.2 Configuring Log4J

Soapmill already has a default log4j configuration. If you use the logger and do not configure it, things will work fine, except that you'll see a lot of messages you might not want.

To start limiting the verbosity of the logs, or to create specialised log files, you can create the file `log4j.snippet` in your package's `etc` directory. Refer to the log4j documentation for how it's actually done.

Here is an example of `log4j.snippet`:

```
# CONSOLE and GLOBAL are predefined for soapmill
log4j.logger.org.miamm.fakemiamm=INFO, CONSOLE, GLOBAL, MYAPPENDER

log4j.appender.MYAPPENDER=org.apache.log4j.FileAppender
log4j.appender.MYAPPENDER.File=${soapical.soapmill.logdir}/example-log.html
log4j.appender.MYAPPENDER.layout=org.apache.log4j.HTMLLayout
```

### Notes

1. Use the `soapical.soapmill.logdir` environment variable as in the example, if you want your log to appear in the central log directory.
2. Set environment variable `SOAPMILL_LOGDIR` to set the central log directory
3. The default value of `SOAPMILL_LOGDIR` is the `temp/log` subdirectory of your servlet container.
4. Set environment variable `SOAPMILL_LOGLEVEL` to control the global minimum verbosity threshold.
5. Soapmill's log4j configuration lives in `etc/log4j.snippet` (like it should for your agent), if you wish to take a look.

## Chapter 3

# Packaging Manual

For the sake of your integrators' sanity, you should put your agent in a nice, self-contained package known as a **web application**.

This section will walk you through the steps in creating a webapp from your agent. If you follow the steps in this chapter, integrators will be able to use your software almost as easily as copying it a single directory..

We assume that you have a development directory (`/devel/myagent`), which contains a subdirectory for your source code. To avoid potential confusion it is helpful to treat your development directory as something completely different from the package you give your integrators.

### 3.0.1 kowey-generic

The steps in this chapter are neatly captured by `kowey-generic`, a tool to help developers create software packages of all kinds. Please see `kowey-generic` for more details (<http://unannoy.sourceforge.net/kowey-generic>).

## 3.1 Version numbers

The first thing to keep in mind for your integrators' sanity is that you will very likely make modifications to your agent and distribute these modifications to other people. To prevent any confusion, you should apply a version number to your package (for example `myagent-0.1`). This way, tracking the interaction between several components is much easier, especially when the components are evolving.

### Notes

1. `Kowey-generic` does this for you. See its doc.

## 3.2 The directory WEB-INF

### 3.2.1 The file WEB-INF/web.xml

Your package should contain a file called `web.xml` in the directory `WEB-INF`. Here is the skeleton for this file.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">
```

```
<web-app>
  <display-name>ExampleAgent</display-name>

  ...

</web-app>
```

### Initialisation parameters

The first servlet you should enable is the ParamReader. This converts servlet initialisation parameters into system properties (sections 2.2 and 2.3), and does some important initialisation work for your agents.

```
<web-app>
  ...

  <servlet>
    <servlet-name>ExampleParamReader</servlet-name>
    <servlet-class>soapical.soapmill.engine.ParamReader</servlet-class>

    <init-param>
      <param-name>com.foobar.parameter</param-name>
      <param-value>some-value</param-value>
    </init-param>

    <init-param>
      <param-name>com.foobar.config-file</param-name>
      <param-value>AGENT_ROOT/etc/config.xml</param-value>
    </init-param>

    <load-on-startup>1</load-on-startup>
  </servlet>

  ...
</web-app>
```

### Notes

1. Normally in Java, servlet initialisation parameters are different from system properties. But Soapmill introduces a mechanism that sets system properties for each initialisation parameter that it sees.
2. The text `AGENT_ROOT` will be replaced with the document root of your servlet. It is mostly there for backwards compatibility, since the `agentRoot()` method does the same thing.
3. `SOAPMILL_TMP` will be replaced with the temp directory.

### Agents (servlets)

Each agent that you want to launch should be listed in `web.xml` as a servlet.

```
<web-app>
  ...
```

```
<servlet>
  <servlet-name>ExampleAgent</servlet-name>
  <servlet-class>com.foobar.ExampleAgent</servlet-class>
  <load-on-startup>5</load-on-startup>
</servlet>

...
</web-app>
```

### 3.2.2 Jar files or classes

Your package should include a directory `WEB-INF/classes` and/or `WEB-INF/lib` which contain the classes of your agent and any special jar files not already used by Soapmill. To avoid version conflicts, you should *NOT* include the soapmill jars or any jars used by Soapmill.

This poses a potentially tricky question: how would I, the developer, compile my agent using the soapmill jars without including these jars in the final package? There are two potential solutions. One is to create your package by hand, which avoids any problems posed by automation; however, this gets rather tiring if you make many modifications to your agent and release new versions of your package very frequently. A better solution is to use kowey generic (see 3.0.1).

## 3.3 File etc/log4j.snippet (optional)

You can also include a log4j configuration file. See section 2.5.2 for more details.

## 3.4 Data directories

Don't forget to include any data directories that you use. It is often handy to put miscellaneous files in the `/etc` directory and simply include that.

## 3.5 The package

The final part of creating a package would be to put it together into a standard `.war` file. If you are not using kowey-generic you should refer to online documentation on creating web applications. Kowey-generic users can accomplish most of the above steps relatively easily:

1. Edit `etc/make/config` and set `enable.webapps`
2. Edit `web.xml` in `web/WEB-INF`
3. Run `bin/build.sh webapps`.
4. The resulting `.war` file is in `/devel/myagent/dist/myagent-0.N/webapps`.

# Chapter 4

## Integration Manual

This section describes a way for getting Soapmill to painlessly run a few agents together.

### 4.1 Install the Soapmill

Install Apache Tomcat or some other Java servlet container. Download and install the Soapmill (<http://soapical.sourceforge.net/soapmill>):

1. `cd /whatever`
2. `tar -xzvf /tmp/downloads/jakarta-tomcat-4.1.24.tgz`
3. `tar -xzvf /tmp/downloads/soapmill-bin-0.15.1.tgz`
4. `cd soapmill-0.15.1`
5. `cp webapps/soapmill-0.15.1.war /whatever/tomcat-4.1.24/webapps`
6. `mkdir -p /whatever/tomcat-4.1.24/shared/lib`
7. `cp lib/*.jar lib/axis-1.1/*.jar /whatever/tomcat-4.1.24/shared/lib`

#### Notes

1. The Soapmill has been tested on Apache Tomcat 4.1.24.
2. Check the script if you want to do this installation by hand; it is a rather quick process.

### 4.2 Install the agents

Copy the `.war` files for each agent into the servlet container's `webapps` directory.

1. `cd /tmp/downloads`
2. `cp soapmeter-0.5.war /whatever/tomcat-4.1.24/webapps`
3. `cp some-agent-0.4.war /whatever/tomcat-4.1.24/webapps`

#### Notes

1. The soapical Soapmeter is an extremely useful agent to obtain, as it will help you to debug the soapmill and your agents. See <http://soapical.sourceforge.net/soapmeter>.

### 4.2.1 Disable unwanted agents

Some packages you download might include undesirable agents. Uncompress the webapps and comment out the unwanted agents in the `web.xml` file.

1. `cd /whatever/tomcat-4.1.24/webapps`
2. `mkdir soapmeter-0.5`
3. `cd soapmeter-0.5`
4. `jar xvf ../soapmeter-0.5.war`
5. `vi WEB-INF/web.xml`

## 4.3 Configure multiple servers

If there are other soapmill servers in the picture, you'll need to tell the soapmill where they are. Edit the `etc/remote.agents` in the soapmill webapp (for example `/whatever/tomcat-4.1.24/webapps/soapm`

```
# this is /whatever/tomcat-4.1.24/webapps/soapmill-0.15.1/etc/remote.agents

remote http://server2.foo.org:8080/soapmill/services/Delegator
remote http://server3.foo.org:8080/soapmill/services/Delegator
```

## 4.4 Run the server(s)

All you have to do is start the servlet container on each server, for example, by running `/whatever/tomcat-4.1.24/bin/catalina.sh run`.

If you want to leave the Soapmill running over the long term, (for example, if you are done testing) you should use the `startup.sh` and `shutdown.sh` scripts instead:

1. `/whatever/tomcat-4.1.24/bin/startup.sh`
2. `tail -f /whatever/tomcat-4.1.24/log/catalina.out`

## 4.5 About upgrading

A final note about upgrading the soapmill or your agents. Please take care to fully remove any prior versions from the servlet container before you install the new versions.

# Part II

# Conclusion

# Chapter 5

## Questions and Problems

This chapter will discuss the finer points and pitfalls of using the Soapmill.

### 5.1 Tips and Clarifications

#### 5.1.1 How to I debug my agent?

Use the soapmeter: <http://soapical.sourceforge.net/soapmeter>. It has special soapmill-related functionality. On a general note, you should use techniques like unit testing to verify that your agent works *before* deploying and debugging it.

#### 5.1.2 I hate setting JAVA\_HOME!

So don't.

Be careful here, you might be tempted to simply set this thing for good inside your `.bashrc` or `.cshrc`, but doing so might cause confusing problems for other applications. Here's a nicer way: set an alias.

#### What's your shell?

In your terminal, type `setenv`. If you see a long list of variables you are using (t)csh. If it says "command not found", you are likely using bash.

#### Bash

Open `~.bash_profile` in your favourite text editor and enter the following:

```
alias jh='export JAVA_HOME=/path/to/your/java/home'
```

#### (T)csh

Open `~.cshrc` in your favourite text editor and enter the following:

```
alias jh='setenv JAVA_HOME /path/to/your/java/home'
```

#### No more nonsense!

Open a new terminal. From now on, all you have to do is type "jh" instead of huge long JAVA\_HOME path mess.

### 5.1.3 How many SoapmillAgents?

Only one instance your SoapmillAgent is used by the Soapmill, the one returned by your agent's `instance()` method.

### 5.1.4 How can I log my agent's activities?

See section 2.5

## 5.2 Troubleshooting

### 5.2.1 Nothing Happens

1. Did you remember to allow the `message` type which you are interested in (section 1.3.1)?
2. Did you put your agent in the integration directory? (section 4.2)

### 5.2.2 Ugly error messages

1. Is your code compiled with Java 1.4.1?  
Soapmill requires its agents to use the same version of Java as it.
2. Are you sure your code is the right Java version?  
It would be good to erase all your compiled files and recompile from scratch, making sure you are using the right tools
3. Are you using the right libraries?  
see 3.2.2
4. Is the server running?

### 5.2.3 Bind exception

1. You probably tried to run the server when it was running already. Close down the first instance and try again. Another possibility is that you have a different server running on the same port. You could either stop that server, or modify Soapmill's tomcat configuration so that it runs somewhere else.

### 5.2.4 Cannot connect

1. Check the `.agents` files

Is the name of the server correct in those `.agents` files? You could verify against the agent's deployment descriptors. Launcher properties are an unfortunate and temporary crutch. They will hopefully disappear when complementary agents become available for the soapmill.

## Chapter 6

# Conclusion

This concludes the Soapmill manual. I should point out again that this is by no means a final draft of the report, the specification, or the library. I am aiming to make this report as concise and approachable as possible. Please let me know where the document is unclear or could be more helpful. Please contact me (`kow at loria point fr`) if you encounter any problems or mistakes in the document.

### 6.1 Acknowledgments

Dirk Fedeler of DFKI created a SOAP-bypass and the simple visualiser which was eventually incorporated in the Soapmeter. This has immensely boosted performance and allowed for some nice simplifications. His clean code and simple solutions have taught me more than a thing or two.

An appreciative nod to Ashawni Kumar, Laurent Romary and Annalisa Ricci for play-testing the product. Special thanks to Frederic Landragin for his indispensable help in turning the document into something vaguely readable.

This product includes software developed by the Apache Software Foundation (<http://www.apache.org>).

## Appendix A

# Programmer's Glossary

**await** Soapmill's approach to simulating RPC via asynchronous messaging. For the programmer, this means you can have something which acts like a traditional function call, two way communication.

Origin: eric

**callback method** A method that you implement when using libraries like SAX or Soapmill. This method is called by the library at some arbitrary point.

Origin: computer programming community.

**URI** Like URLs, but more general. See <http://www.w3.org/Addressing/>

## Appendix B

# Example message types

Here are some examples of message types, incidentally, the ones used in the MIAMM project.

### B.1 MPEG-7 types

These types are relative to the URI `urn:mpeg:mpeg7:soapmill`. For example, `word_graph_fr` should really be read as `urn:mpeg:mpeg7:soapmill:word_graph_fr`

**word\_graph\_fr** a word lattice for French, most likely from a speech recogniser (speech interpretation)

**word\_graph** a word lattice for any other language, most likely from a speech recogniser (speech interpretation)

### B.2 MMIL types

These types are relative to the URI `http://www.miamm.org/mmil`.

For example, `goal_representation` should really be read as

`http://www.miamm.org/mmil/goal_representation`

**language\_generation\_request** an utterance which needs to be realised as natural language (language generation)

**speech\_generation\_request** information that a speech synthesiser can use to generate speech (speech synthesis)

**semantic\_representation** the semantic representation of an utterance (multimodal fusion)

**goal\_representation** a visualisation task which needs to be planned (action planner)

**visualisation\_request** a visualisation task which needs to be realised (vishaptac)

**visualisation\_status** the state of a visualisation request (multimodal fusion and/or action planner)

**domain\_model\_query** a query to the domain model (domain model)

**domain\_model\_response** a response to a domain\_model\_query (action planner)

**mp3player\_request** a request for some mp3 to be played

**mp3player\_status** the state of the mp3 player

**media\_request** a request for playable media (mpeg database)

**media** playable media (mp3 player?)

# Bibliography

- [1] JUnit Homepage. <http://www.junit.org>.
- [2] The Castor Project. <http://castor.exolab.org>.
- [3] The Java Architecture for XML Binding. <http://java.sun.com/xml/jaxb/index.html>.
- [4] The Java Tutorial. <http://java.sun.com/docs/books/tutorial/essential/threads/>.  
See especially “Synchronizing Threads”.
- [5] Ceci Guelcue. Log4J Short Manual. <http://jakarta.apache.org/log4j/docs/manual.html>.
- [6] Eric Kow. The Soapmill Report. <http://soapical.sourceforge.net> or docs/technical\_report in your Soapmill distribution.