XINS User Guide

Anthony Goubard <anthony.goubard@japplis.com>

Table of Contents

Introduction ... 3
- What is XINS? ... 3
- Notation ... 3
- Requirements ... 3
- Installation ... 4
- Code conventions ... 4

Setting up a new project ... 5
- xins-project.xml ... 5
- api.xml ... 5
- MyFunction.fnc ... 6

Compiling and running the project ... 6
- The HTML specification documentation ... 7
- The web application. ... 7
- The server-side Javadoc ... 10
- The client API ... 10
- The client-side Javadoc ... 11
- The Open Document Format ... 11

Implementing the method ... 12
- Defining Input, Output. ... 12
- Defining the test environments. ... 13
- Defining the examples. ... 13
- Implementing your function. ... 14
- Executing your function ... 14

The error code ... 14

The runtime properties ... 16
- The properties ... 17
- Getting properties value ... 18
- Defining properties ... 19

The bootstrap properties ... 20

Integration with CVS / Subversion ... 21

Managing dependencies ... 22

Adding other files to the WAR file ... 23

Multiple implementations and stubs ... 23

Testing ... 24

Categories ... 24

Managing logs ... 25
- Managing the logs on the server side. ... 25
- Adding your own logs. ... 26
- Configuring the properties files for specific logs ... 27

Managing load balancing and fail over ... 28

XINS types ... 29
- Predefined types. ... 29
- Defined types ... 30
- Shared types ... 31

Grouping types ... 31

Param combos 31
Attribute combos 32
Combos based on values 33
The data section. 34
The specification 34
The implementation 35
The result 35
Data section for the input 36
Data section on the client side 36
Function accesses 37
ACLs 38
Enable/Disable a function 39
HTTPS 39
The meta functions 39
Management 40
JMX 41
Versioning 41
Shared instance 41
Calling Convention 42
How to define the calling convention 42
Standard calling convention 43
XSLT calling convention 43
SOAP calling convention 43
XML calling convention 44
XML-RPC calling convention 44
JSON-RPC calling convention 45
JSON calling convention 45
SOAP-MAP calling convention 46
Custom calling convention 46
Examples calling convention 47
Utility classes 47
Ant tasks 48
callxins task 48
xins task 49
XINS targets 49
XINS tools 50
Performances 54
Compressed data 54
Cached data 55
Not modified data 55
Content of the XINS package 56
Documentation 56
Examples 56
Program 57
Integration with IDE 57
Integration with Eclipse. 58
Integration with NetBeans 59
Integration with other frameworks 60
Spring framework 60
Dojo toolkit 61
Google Web Toolkit (GWT) 61
AJAX 61
Mule 62
ESB and JavaEE 62

Ant 62
Maven 62
Script languages (Groovy, Ruby, PHP, Perl...) 63
Google App Engine (work in progress) 63

Introduction

This document explains the XINS functionalities. It starts by explaining how to create and run a simple API and continue by explaining the more advanced features.

What is XINS?

XINS is an open-source Web Services framework.

XINS supports POX-RPC, SOAP, XML-RPC and more. It consists mainly of an XML-based specification format and a Java-based implementation framework. From its specifications, XINS can generate HTML, WSDL, client-side code, server-side code, test forms and more.

Notation

This document uses the Windows file system. If you are installing XINS in a Unix operating system like Mac OS X, linux or Solaris, change all back slash characters ('\') with a forward slash character ('/') in the file names of this document.

The documentation provides some Javadoc links that only work in the HTML generated user guide.

Requirements

XINS requires the following software:

- Java Development Kit 1.5 or higher: http://www.oracle.com/technetwork/java/javase/overview/index.html.

  If not done by the installation, you should set the environment variable `JAVA_HOME` to the directory where you have installed Java. Also add the `%JAVA_HOME%\bin` directory to the `PATH` environment variable. You can check the Java version installed by executing `java -version`.

  If you want to use XINS with Java 1.3.1 or Java 1.4, you can download XINS 2.3.

  XINS has been tested with Java 5.0, Java 6.0 and java 7.0.

- Ant: http://ant.apache.org/, version 1.6.2 or higher.

  You should set the environment variable `ANT_HOME` to the directory where you have installed Ant. Also add the `%ANT_HOME%\bin` directory to the `PATH` environment variable. You can check the Ant version installed by executing `ant -version`.

  ### Caution

  If you have a putDocumentInCache error message in Eclipse, you may need to copy xercesImpl-2.6.2.jar [http://mirrors.ibiblio.org/pub/mirrors/maven2/xerces/xercesImpl/2.6.2/xercesImpl-2.6.2.jar] in the `eclipse\plugins\org.apache.ant_1.6.5\lib` directory.

If you're using the Ant distribution included with NetBeans and have putDocumentInCache error messages, you should change the Ant home location to Ant 1.7.0 or higher (in Tools -> Options -> Miscalleneous -> Ant.

- Optional: A servlet container. For example: Tomcat [http://tomcat.apache.org/] or Jetty [http://jetty.codehaus.org/jetty/index.html]. Any J2EE server can also deploy web applications. For example: Glassfish [https://glassfish.dev.java.net/], Orion [http://www.orionserver.com/], JBoss [http://www.jboss.org/], WebSphere [http://www.ibm.com/software/info1/websphere/index.jsp], WebLogic [http://www.bea.com/products/weblogic/server/index.shtml], Resin [http://www.caucho.com/resin/] or JRun [http://www.macromedia.com/software/jrun/?promoid=home_prod_jr_100803].

  For the installation of the servlet container, please refer to the documentation of the downloaded product.

  XINS contains it's own servlet container implementation for basic testing purposes only.

- Optional: A version control system. For example CVS [http://www.cvshome.org/] or Subversion [http://subversion.tigris.org/].

This document will assume that you have installed Java and Ant.

Installation

XINS can be installed using the Windows installer or by downloading the `.tgz` file and unzipping the file. Both options are explained below.

Windows Installer

- Download xins-3.0.exe [http://prdownloads.sourceforge.net/xins/xins-3.0.exe?download] and execute it. A new directory `c:\Program Files\xins` is created.
- If you choose to compile and run the demo, you can go to the link provided in the `README` file, then click on `MyFunction` and on the `MyComputer` links in the examples section or use the test form link provided on this page.

Using the .tgz file

- Download XINS 3.0 [http://prdownloads.sourceforge.net/xins/xins-3.0.tgz?download].
- Unpack the downloaded file (`xins-3.0.tgz`) to a directory.
- Set the environment variable `XINS_HOME` to the `xins` directory.
- Add the path `%XINS_HOME%\bin` to your `PATH` environment variable

Note

On Windows, the new environment variable will not be set until you have rebooted or logged on again. If you don't want to reboot your computer, you can set the variables with **set XINS_HOME=c:\Program Files\xins** and **set PATH=%PATH%;%XINS_HOME%\bin** in a DOS prompt.

You are now ready to create your first XINS project.

Code conventions

API names should be in lowercase. E.g. `billing` (and not `Billing`).

Functions, types and result codes should use the hungarian naming convention and start with an uppercase. E.g `CheckStatus`, `Customer` and `NotFound`.

Parameters, functions, types and result codes starting with an underscore ('_') are reserved for XINS.

Parameter names should start with a lowercase. E.g: `message` and `houseNumber`.

The Java code used for the generated templates and in the examples has private variables starting with underscore ('_') and contains some comment for the CVS tags. For the rest the Sun Java code conventions and the Javadoc documentation style rules apply.

Setting up a new project

To create a minimal project with one API containing one function, you need 3 files :

- `xins-project.xml` : This file contains the names and properties of the different APIs.
- `api.xml` : This file contains the declaration of the functions, types and result codes used in your project.
- `<function name>.fnc` : This file contains the input parameters, the output parameters, the result codes and some examples of your function.

xins-project.xml

- Create a new directory where your APIs based on XINS will be created. For example `c:\projects`.
- Create a new `xins-project.xml` file in this directory with the content :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE project PUBLIC "-//XINS//DTD XINS Project 3.0//EN"
          "http://www.xins.org/dtd/xins-project_3_0.dtd">
<project name="myprojects"
         rcsversion="$Revision: 1.106 $" rcsdate="$Date: 2013/01/23 10:00:59 $"
         domain="com.mycompany">
</project>
```

api.xml

To create a new `api.xml` execute in the `projects` directory **xins create-api**.

The command will ask you for the name and the description of your api. The script will then create a new `api.xml` file in the directory `apis\<api name>\spec` with the content :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE api PUBLIC "-//XINS//DTD XINS API 3.0//EN"
          "http://www.xins.org/dtd/api_3_0.dtd">

<api name="myproject" rcsversion="$Revision: 1.106 $" rcsdate="$Date: 2013/01/23 1

  <description>Description of the API.</description>

</api>
```

Then the command will ask you if you want to create an implementation of the api. This will create the skeleton java file where you will write the code of your function. If you answer yes, a new file `impl.xml` is created in the `apis\<api name>\impl` directory with the content:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE impl PUBLIC "-//XINS//DTD Implementation 3.0//EN"
          "http://www.xins.org/dtd/impl_3_0.dtd">

<impl>
</impl>
```

Finally the command will ask you if you want to define some environments with your api. If you answer yes, a new file `environments.xml` is created in the `apis\<api name>` directory with the content:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE environments PUBLIC "-//XINS//DTD Environments 3.0//EN"
          "http://www.xins.org/dtd/environments_3_0.dtd">

<environments>

  <environment id="localhost" url="http://127.0.0.1:8080/myproject/" />

</environments>
```

Note

In this guide we will use the `xins` command to create the different files of the project, but it is also possible to do it manually using a text editor.

If you created the file manually, add the line `<api  name="myproject"/>` in the `xins-project.xml` file. If the API has an implementation, add the element `<impl/>` in the `<api>` element and if it defines some environments add `<environments/>` in `<api>`.

MyFunction.fnc

Now that you've created the API definition, we need to define a function within this API.

To create a new function execute **xins create-function**.

The command will ask you for the name of your api and the name and the description of your function. The script will then create a new `<function name>.fnc` file in the directory `apis/<api name>/spec` with the content :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE function PUBLIC "-//XINS//DTD Function 3.0//EN"
          "http://www.xins.org/dtd/function_3_0.dtd">

<function name="MyFunction" rcsversion="$Revision: 1.106 $" rcsdate="$Date: 2013/0

  <description>Description of the function.</description>

</function>
```

If the file was created without calling the `create-function` target, add the line `<function name="MyFunction"/>` in the `api.xml` file.

Compiling and running the project

Now we have a simple project with a function that, for the moment, does nothing.

From the specification files XINS can generate :

- *Specification documentation.*

  The specification documentation is a set of static HTML pages containing the description of your specification. It lists all APIs, functions, types and error codes. The pages also provide an easy way to call your function using the web application.

- *A web application*.

  The web application is packaged as a WAR file that you can deploy in any servlet container. Once the WAR file is deployed, you can access your API through HTTP.

- *The client-side API*.

  The client API is a JAR file that can be used to invoke remotely the API functions from Java programs. It features various advanced features, such as load-balancing, fail-over, extensive logging, etc...

- *The Javadoc*.

  The Javadoc contains the definition of the API for the Java classes, including the generated classes. The javadoc can be generated either for the server side classes or the client side classes.

The HTML specification documentation

To generate the HTML specification documentation (also called `specdocs`) execute **xins specdocs-myproject**.

This will generate a set of static HTML pages that you can view by opening the file `build\specdocs\index.html`.

The HTML pages contain:

- An overview with a description of the different functions, types and error codes used in your API.
- A link for each environment to the meta functions `_GetVersion`, `_GetStatistics` and `_GetSettings`.
- For each function, a description of the input parameters, output parameters, the validation rules and the possible result codes for this function.
- For each function, a test form page to test the function.
- For each example in the function, a description of the URL request, the expected result for this example and the links to execute the example on the given environments.
- For each of the types, a description of the type. If the type is an enumeration, the possible values are written. If the type is a pattern then the regular expression for the pattern is written along with a link to test the provided pattern.

The web application.

Create the web application

The web application is packaged as a WAR file. When this WAR file is deployed on the servlet container, your functions are accessible through HTTP (for example using Internet Explorer).

To create the web application, execute **xins war-myproject**.

This command will create a `myproject.war` file in the directory `build\webapps\myproject\`.

This target also generates the skeleton file for the implementation, if the file did not exist. In our example the file will be `apis\myproject\impl\com\mycompany\myproject\api\MyFunctionImpl.java`.

You can set some compilation and running properties by creating a `build.properties` file in your project directory and set any of the following properties in the file:

Table 1. build.properties properties

Property name	Description	Example
build.compiler	The compiler to use to compile the classes. (Defaults to javac)	build.compiler=jikes
build.deprecation	Warn if deprecated methods are used. (Defaults to true)	build.deprecation=false
build.java.version	The version of Java where the code will be executed.	build.java.version=1.3
org.xins.server.config	The location to the runtime properties file. Slash and backslash in the property value are translated with the system path separator in XINS. The location can also be a URL.	org.xins.server.config=../xins.properties
servlet.port	The port number for the XINS servlet container (Defaults to 8080).	servlet.port=8181
jmx.port	The port number for JMX monitoring (Defaults to 1090).	jmx.port=2222
test.environment	The environment upon which the tests should be executed (Defaults http://localhost:8080/).	test.environment=http://integration.mycompany.com/myproject/
test.start.server	Flag to indicate that the API should be started when the tests are executed (Defaults to false).	test.start.server=true
wsdl.endpoint	The endpoint to use in the generated WSDL file (Defaults to the first entry of the `environment.xml` file)	wsdl.endpoint=http://www.mycompany.com/myproject/
reload.stylesheet	A work around property for Xerces library that produces sometimes putDocumentInCache errors.	reload.stylesheet=true

These properties can also be passed to the system by using `-D<property name>=<property value>` on the command line.

Run the web application

To run the web application, execute **xins run-myproject**.

It is adviced to execute the web application with a specified runtime properties file. If no file is specified, only the local machine will have access to the API. For more information go to the runtime properties section.

You can also start the API using **java [-Dorg.xins.server.config=<runtime property file>] -jar myproject.war [-port=<port number>] [-gui]**. It is not needed to have XINS or Ant installed to run the API. The requester or the API or the testers for the API can receive the WAR file and execute it. The -gui option will start a user interface where the system output will be logged in a console.

To start the WAR file using the `Jetty` servlet container, create a file named `myproject.xml` int the projects directory with the following content :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC
 "-//Mort Bay Consulting//DTD Configure 1.2//EN"
 "http://jetty.mortbay.org/configure_1_2.dtd">

<Configure class="org.mortbay.jetty.Server">

  <Call name="addListener">
    <Arg>
      <New class="org.mortbay.http.SocketListener">
        <Set name="port">8080</Set>
      </New>
    </Arg>
  </Call>

  <Call name="addWebApplication">
    <Arg>/myproject/</Arg>
    <Arg>build/webapps/myproject/myproject.war</Arg>
  </Call>

</Configure>
```

Then start **java -Dorg.xins.server.config=xins.properties -jar %JETTY_HOME%\start.jar myproject.xml** in the `projects` directory.

Now execute your function by going to the address http://localhost:8080/myproject/?_function=MyFunction.

As nothing is implemented in your method, your function will just return a successful result.

If you want to get more information on your function or on XINS run the following meta functions :

- http://localhost:8080/myproject/?_function=_GetVersion
- http://localhost:8080/myproject/?_function=_GetStatistics
- http://localhost:8080/myproject/?_function=_GetSettings

Note

If you want to run the example in tomcat, you need to copy the `demo\build\webapps\myproject\myproject.war` to the `tomcat\webapps` directory and then start tomcat using the startup script. The URLs to access the functions will then start with `http://localhost:8080/myproject/`.

The server-side Javadoc

To generate the Javadoc on the server-side execute **xins javadoc-api-myproject**.

This will generate a set of static HTML pages that you can view by opening the file `build\javadoc-api\myproject\index.html`.

The client API

To generate the jar file for the client API execute **xins jar-myproject**.

This will generate a jar file located at `build\capis\myproject-capi.jar`.

The following code shows how to use the generated CAPI to call a function. In this example, the `myproject` API is running on the same computer.

```
import org.xins.common.service.TargetDescriptor;

import com.mycompany.myproject.capi.CAPI;
import com.mycompany.myproject.capi.MyFunctionResult;

public class TestMyFunction {
   public final static void main(String[] args) throws Exception {

      // Create the descriptor for the service
      TargetDescriptor descriptor =
         new TargetDescriptor("http://localhost:8080/myproject/", 20000);

      // Create the CAPI instance
      CAPI project = new CAPI(descriptor);

      // Invoke the function
      MyFunctionResult result = project.callMyFunction();

      // No exceptions thown
      System.out.println("Call successful: " + result.getOutputMessage());
   }
}
```

It's also possible to invoke your API without using the generated CAPI file but by using the `XINSServiceCaller`. Examples on how to do it are provided in the XINSServiceCaller Javadoc [javadoc/org/xins/client/XINSServiceCaller.html].

The generated `callMyFunction` method can throw several kind of exceptions. For more information, refer to the generated CAPI Javadoc. Note that a subclass of the exception could be thrown. For example, if you want to catch the exception only when the function returns an error code, catch the `org.xins.client.UnsuccessfulXINSCallException` which is a subclass of the `org.xins.client.XINSCallException`.

You can also catch all the exception at once by catching the class `org.xins.common.service.CallException`.

```
      try {
         // Invoke the function
         project.callMyFunction();
      } catch (UnsuccessfulXINSCallException ex) {
         System.out.println("A standard error occured: " + ex.getErrorCode());
      } catch (CallException ex) {
         System.err.println("Execution failed: " + ex.getMessage());
      }
```

Most of the time your function will have input parameters to pass to the function. This can be done by passing the parameters to the callMyFunction method:

```
import com.mycompany.myproject.capi.CAPI;
import com.mycompany.myproject.types.Gender;
...
   MyFunctionResult result = project.callMyFunction(Gender.MALE, "Doe");
```

or to use the generated MyFunctionRequest object:

```
import com.mycompany.myproject.capi.CAPI;
import com.mycompany.myproject.types.Gender;
...
   MyFunctionRequest request = new MyFunctionRequest();
   request.setGender(Gender.MALE);
   request.setPersonLastName("Doe");
   MyFunctionResult result = project.callMyFunction(request);
```

In the second case, you don't need to call the set method for the optional parameters that are not set whereas in the first case you need to pass null values.

A new package `org.xins.client.async` has been added in XINS 1.4.0. This package provides you with some objects which facilitates the call to remote API asynchronously. For more information, read the article published at http://xins.sourceforge.net/asynchronous.html.

As of XINS 1.5.0, it is also possible to call the API without using the HTTP connection but directly as a function call. You just need to pass the location of the war file containing the API. As no connection is involved, the time-out represents the total time-out:

```
TargetDescriptor descriptor =
         new TargetDescriptor("file:///home/myuser/myproject.war", 20000);
```

It is possible to specify the HTTP method to use (`POST` or `GET`, default is `POST`), or whether redirection returned HTTP code should be followed (default is `false` by using the `XINSCallConfig` class.

The client-side Javadoc

To generate the Javadoc on the client-side execute **xins javadoc-capi-myproject**.

This will generate a set of static HTML pages that you can view by opening the file `build\javadoc-capi\myproject\index.html`.

The Open Document Format

The Open Document Format [http://en.wikipedia.org/wiki/OpenDocument] is a standard for word processors documents (similar to .doc). XINS contains a target that can generate the specifications of an API in this format.

To create the document, execute **xins opendoc-myproject**.

This will generate a `build\opendoc\myproject\myproject-specs.odt` file than can be opened with for example Open Office [http://www.openoffice.org/]. Note that Open Office is free and can save your document as a MS Word document or as PDF.

Implementing the method

For the moment our function has no input, no output and no implementation, we will adapt our previous project to add a minimum of functionalities.

Defining Input, Output.

Let's add to our project three input fields and a result with one output field.

When you are adding input and output parameters to your function, you should at the same time declare the type of your parameter. There are two kinds of types, the ones already defined in XINS and the ones you create. The types already defined in XINS are explained in the next chapter.

To create a new type for your project execute **xins create-type**.

The command will ask you for the name of the API, the name of the type and the description of the type. It will then create a new `<type name>.typ` file with the content :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE type PUBLIC "-//XINS//DTD Type 3.0//EN"
"http://www.xins.org/dtd/type_3_0.dtd">

<type name="TypeName"
rcsversion="$Revision: 1.106 $" rcsdate="$Date: 2013/01/23 10:00:59 $">

  <description>Description of the type.</description>

</type>
```

The different possibilities of types that could be defined are listed in the section called “Defined types”.

Let's create two types for our project, one with the type `lastName` and one with the type `gender`.

Edit the `LastName.typ` file with

```
  <description>Last name of a person.</description>
  <pattern>[A-Za-z ]{1,50}</pattern>
```

Edit the `Gender.typ` with

```
  <description>A gender.</description>
  <enum>
    <item name="male"   value="m" />
    <item name="female" value="f" />
  </enum>
```

If the name attribute is not specified it will be the same as the value attribute.

Now that the types are created, we need to add the declaration in `api.xml` and to set them as input parameters in our function. For the output we will use a predefined type so there is no need to define the type in `api.xml`.

Add `<type name="Gender"/>` and `<type name="LastName"/>` to `api.xml`.

Add to the file `MyFunction.fnc` :

```
<input>
  <param name="gender" required="true" type="Gender">
    <description>The gender of the person.</description>
  </param>
  <param name="personLastName" required="true" type="LastName">
    <description>The last name of the person.</description>
  </param>
</input>
<output>
  <param name="message" required="true" type="_text">
    <description>The message returned to this person.</description>
  </param>
</output>
```

You can also assign a default value for an optional parameter using the `default` attribute.

```
  <param name="gender" required="false" type="Gender" default="m">
    <description>The gender of the person.</description>
  </param>
```

Defining the test environments.

You can specify in XINS the environments where you want to test the API.

As seen in the section called “api.xml”, it's possible create the file containing the test environments by using the **xins create-api** or by creating the file with a text editor and editing the `xins-project.xml`.

When you generate the specification documentation, for each function defined in the API a link below "Test forms" that points you to a HTML form which is used to fill the input parameters of the function. When you then click on Submit the function will be executed on the specified environment. This requires of course that the application is installed on the defined environment.

The environments are also listed on the main page of the specification documentation along with some links that provide direct access to some meta functions.

If you want to add a new environment, just add for example `<environment id="production" url="http://www.mycompany.com:8080/my-project/" />` to `environments.xml`.

Defining the examples.

You can also define some examples for your API in the specification. The definition of the examples are done in the function specification file.

Let's add some examples to `MyFunction.fnc` by adding the following text after the `output` section:

```
<example resultcode="_InvalidRequest">
  <description>Missing parameter : lastName</description>
  <input-example name="gender">m</input-example>
</example>
<example resultcode="_InvalidRequest">
  <description>Invalid parameter</description>
  <input-example name="gender">m</input-example>
  <input-example name="personLastName">Bond 007</input-example>
</example>
<example>
  <description>Message returned.</description>
  <input-example name="gender">f</input-example>
  <input-example name="personLastName">Lee</input-example>
  <output-example name="message">Hello Miss Lee</output-example>
</example>
```

This example shows how to add some examples to a function with `input` and `output` parameters but it's also possible to define examples for a function that contains a data section or returns a result code that contains some parameters. For more information, look at the function specification defined in the `allinone` project.

The examples are shown on the generated specification documentation and if you have also defined some environments, you can execute the example by just clicking the provided links.

In XINS 1.4.0, you can create an example using **xins create-example**. The API should be running on http://localhost:8080/<api name>/. The create-example target will ask you for the name of the API and the request URL. The example is automatically added to the function.

Implementing your function.

You can edit the new `MyFunctionImpl.java` generated file located in the `apis\myproject\impl\com\mycompany\myproject\api` directory with

```
   public final Result call(Request request) throws Throwable {

      String nomination = null;
      if (request.getGender().equals(com.mycompany.myproject.types.Gender.MALE)) {
         nomination = "Mister";
      } else {
         nomination = "Miss";
      }
      SuccessfulResult result = new SuccessfulResult();
      result.setMessage("Hello " + nomination + " " +request.getPersonLastName());
      return result;
   }
```

Executing your function

We first need to rebuild the WAR file, the spec docs, the client jar and the javadoc by executing **xins all-myproject**.

Now you must restart the servlet container server and reopen the file `build\specdocs\index.html`. You can test your function by either clicking on the link provided with the examples or by using the test forms.

The error code

An error code (previously named result code) contains the description of an error that occurred when executing the implementation code.

When you generate the specification documentation, you can already notice that some error codes already exist by default. These error codes are returned by the system when the problem occurs.

You should create custom error codes whenever the implementation of your function would encounter an error condition that should be distinguishable for the caller. For example, if you connect to another service, if you perform I/O operations or if the input fields should match fields in a database.

We will now extend our example with a `NoVowel` error code. Note that this error could be also detected by improving the `lastName` pattern.

Create a error code by executing **xins create-rcd**.

The command will ask you for the name of your api and the name and the description of your error code. The script will then create a new `NoVowel.rcd` file in the directory `<specsdir>/<api name>` with the content :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE resultcode PUBLIC "-//XINS//DTD Result Code 3.0//EN"
          "http://www.xins.org/dtd/resultcode_3_0.dtd">

<resultcode name="NoVowel"
            rcsversion="$Revision: 1.106 $" rcsdate="$Date: 2013/01/23 10:00:59 $"

  <description>The name does not contain any vowels.</description>

</resultcode>
```

Once this file is created, add the line `<resultcode name="NoVowel" />` in the `api.xml` file. A reference to this error code also needs to be defined in the function that may return this error code. In our case `MyFunction.fnc`. Also add the line `<resultcode-ref name="NoVowel" />` at the beginning of the `output` section of `MyFunction.fnc`.

Now we need to adapt our Java code to detect if the given last name does not include any vowels and return the error code when it is the case. Add the following code at the beginning of your `call()` method.

```
String lastNameLower = request.getPersonLastName().toLowerCase();
if (lastNameLower.indexOf('a') == -1 && lastNameLower.indexOf('e') == -1 &&
    lastNameLower.indexOf('i') == -1 && lastNameLower.indexOf('o') == -1 &&
    lastNameLower.indexOf('u') == -1 && lastNameLower.indexOf('y') == -1) {
      return new NoVowelResult();
}
```

Now we need to adapt the specification of your function to include this result code in the examples. We moved the example 3 to example 4 as we wanted to keep the order "unsuccessful results" "successful results" in our examples and we added the example 3 with

```
<example resultcode="NoVowel">
  <description>The name does not contain any vowels.</description>
  <input-example name="gender">f</input-example>
  <input-example name="personLastName">qslkfj</input-example>
</example>
```

You can now regenerate everything with **xins all-myproject** and after restarting the servlet container server, test the new example using the generated specification documentation.

It is possible to have parameters and a data section in the error code. To do it, add an `output` element to the error code. For the examples in the specification and setting the values in the implementation, just do it as you would do for a function with output parameters or data section. The `allinone` example contains a `AlreadySet` error code with one parameter.

Since XINS 1.4.0, it is possible to specify if an error code is functional or technical. Functional error code are error that may occur such as `InvalidPassword` where as technical errors indicate a problem such as `DatabaseCorrupted`. By default an error code is technical. To specify the error code as functional add the attribute `type="functional"` to the `resultcode` element.

If you want to reuse an error code already defined in another API, you don't need to copy the file. You can also point to the error code of the API. Its uses the same system as in the section called “Shared types”.

XINS also has predefined error code that are returned when something went wrong.

Table 2. XINS defined error code

Error code	Description
_DisabledFunction	The function is currently disabled.
_InternalError	There was an internal error.
_InvalidRequest	Indicates that the request is considered invalid since it does not match the criteria defined by the specification.
_InvalidResponse	Indicates that the implementation produced a response that does not match the criteria defined by the specification. Typical callers should treat this like an internal error.
_DisabledFunction	The function is currently disabled.
_DisabledAPI	The API is currently disabled.
_FunctionNotSpecified	Indicates that the request does not contain which function should be called.
_FunctionNotFound	Indicates that the function of the request does not exist.
_InvalidRequestFormat	Indicates that the input format is invalid for the calling convention.
_NotAllowed	Indicates that the user is not allowed to call the function.

The runtime properties

You may want to add some flexibility to your application by changing a property without needing to recompile the code or restarting the server. This way you can distribute the same WAR file of your web application to different places. You can think at the following situations:

- A file location may differ on the computers on which you want to deploy your application.
- The location of a database may differ depending on whether you want to test your application or deploy it in a production environment.
- A username/password which you don't know the value and should be entered by the administrator.
- A link to another XINS service, in case of a distributed environment.

- You would like to adjust settings that influence performances.
- You would like to adjust the debugging level.

The properties for your APIs are defined in the runtime properties file often named `xins.properties`. This file location is passed to the application server with the system property `org.xins.server.config`.

To start the WAR file using the XINS servlet container, just execute **xins -Dorg.xins.server.config=xins.properties run-myproject**. You can also set the location of the `xins.properties` in the `build.properties` and then execute **xins run-myproject**.

Note

If you want to run the example in tomcat, you need to add in the file `tomcat\conf\catalina.properties` the line `org.xins.server.config=c:\\Program\ Files\\xins\\demo\\xins.properties`.

Example of runtime properties file:

```
#_______________________________________________________________________________
# General XINS properties

# Check configuration file every 60 seconds
org.xins.server.config.reload=60

# Access rules
org.xins.server.acl=allow 127.0.0.1 *;

# Logging properties
org.xins.logdoc.locale=en_US
```

An example is also provided in the `demo\xins.properties` file.

It's also possible to define a different runtime property file for each API you are running on the servlet container. To define a specific runtime property file for an API, use org.xins.server.config.<api name> instead of org.xins.server.config.

The properties

Some properties defined in this file are already interpreted by the system:

Table 3. xins.properties runtime properties

Property name	Required	Description	Example
org.xins.server.config.reload	no, defaults to 60.	Interval in seconds for check the xins.properties file for any changes.	org.xins.server.config.reload=60
org.xins.server.acl	no, defaults to localhost.	Access rules for the functions	org.xins.server.acl=allow 127.0.0.1 *; \ allow 192.168.0.0/24 MyFunction
org.xins.server.acl.<api name>	no	Access rules for the functions of the specified	org.xins.server.acl.myproject=allow 127.0.0.1 *; \

		API. The specified rules are executed before the generic one.	deny 192.168.0.0/24 MyFunction
org.xins.server.config.include	no	A comma separated list of runtime property files to include. The files should use relative paths.	org.xins.server.config.include=../ default-xins.properties
org.xins.logdoc.locale	no, defaults to en_US.	The locale to be used for the logging messages.	org.xins.logdoc.locale=en_US
org.xins.server.jmx	no, false by default.	Enables the management of the API using JMX (Java Management eXtension).	org.xins.server.jmx=true
org.xins.server.contextID.filter	no, all types of context ID are accepted by default.	The regular expression pattern for the expected format of the context ID. If the context ID is invalid, a new one will be generated.	org.xins.server.contextID.filter=[a-zA-Z]{5}
org.xins.server.logging.init	no, true by default	Initialize the logdoc logging	org.xins.server.logging.init=false
org.xins.server.contextID.push	no, true by default	Push the contextID to the NDC	org.xins.server.contextID.push=false
org.xins.logdoc.stackTraceAtMessageLevel	no, defaults to false meaning that the stack traces are logged at DEBUG level.	Flag indicating that the exception stack trace should be logged at the same level as the message.	org.xins.logdoc.stackTraceAtMessageLevel
log4j.*	no, defaults to console.	Logging properties used to adapt the debug level, formatting or output to your needs.	log4j.rootLogger=DEBUG, console
log4j.<api name>.rootLogger	no, defaults to log4j.rootLogger.	Root logger for the specified API.	log4j.myproject.rootLogger=INFO, logfile_myproject

Getting properties value

Now we will adapt the `xins.properties` for the `MyFunction` implementation by adding the following lines:

```
# Salutation message for the person
salutation=Hello
```

We now need to change the implementation to read and use the property. Note that the property should be able to be changed at runtime. Add the following code to `MyFunctionImpl.java`

```
import org.xins.common.collections.MissingRequiredPropertyException;
import org.xins.common.collections.InvalidPropertyValueException;
import org.xins.common.manageable.BootstrapException;
import org.xins.common.manageable.InitializationException;
```

```
...
   /**
    * The salutation message.
    */
   private String _salutation;

   protected void initImpl(Map<String, String> properties)
   throws MissingRequiredPropertyException,
          InvalidPropertyValueException,
          InitializationException {

      // Get the salutation
      _salutation = properties.get("salutation");
      if (_salutation == null || _salutation.trim().equals("")) {
         throw new MissingRequiredPropertyException("salutation");
      }
      // Here you can also chack the value and throw an
      // InvalidPropertyValueException if needed
   }

   public final Result call(Request request) throws Throwable {
...
      result.setMessage(_salutation + " " + nomination + " " +
                        request.getPersonLastName());
      return result;
   }
```

Now recompile your code, restart the server, execute the function, change the value in `xins.properties` to Hi, wait for the server to reload the properties file and execute the function again.

The API class defines a `deinitImpl()` method. This method is used to release resources when you stop the API.

Note

When the xins.properties file has changed, only the `initImpl()` method is invoked. The `deinitImpl()` is called only when you stop the servlet container.

The API class has a `bootstrapImpl2(Map<String, String> buildSettings)` method that is called the API is started. The buildSettings passed as parameters are the properties stored in the `web.xml` file included in the deployment war file.

The API class has a `reinitializeImpl()` method. This method can be invoked in your implementation to ask the framework to reinitialize the API.

Defining properties

XINS 1.3.0 includes a new system that allows to define runtime properties that you will use in your API in the `impl.xml` file. In this file you define the name of the property, its description, its type and whether is property is required or optional. The value of the property still needs to be defined in the `xins.properties` file.

Example:

```
<impl>
  <runtime-properties>
    <property name="myproject.eurodollar.rate" type="_float32" required="true">
      <description>The price in dollars of 1 euro.</description>
    </property>
  </runtime-properties>
</impl>
```

Now when the specification documentation is generated, a new page is available containing the list of the runtimes properties used by the API along with their description, their type and whether the property is required or optional. This page makes it easier to deploy an API with the correct runtime properties set.

Another advantage of defining the properties this way is that you don't need to implement the `initImpl` method to retreive the property values. A class is generated that checks and retreives the properties. If the value of a property is incorrect or a required property is missing then the API will fail with a description of the problem. The `getAPI().getProperties()` method can also be called from the `initImpl` method of your function, if you want for example initialize another object based on a runtime property.

Example on how to use the generated class:

```
   // No imports needed, no initImpl method needed

   public final Result call(Request request) throws Throwable {
      SuccessfulResult result = new SuccessfulResult();
      result.setPriceInEuro(request.getPriceInDollar() /
         ((RuntimeProperties) getAPI().getProperties()).getMyprojectEurodollarRate
      return result;
   }
```

If the property was optional, a `java.lang.Float` object would have been returned.

The bootstrap properties

The bootstrap properties are the properties defined in the generated `web.xml` file. These properties cannot be modified at runtime. You can view the properties by calling the calling the meta function `_GetSettings`. When the WAR file is created XINS set some default properties which are:

org.xins.api.name	The name of the API.
org.xins.api.build.version	The version of XINS with which the WAR file was created.
org.xins.api.version	The version of the API (may be empty).
org.xins.api.build.time	The time at which the WAR file was created.
org.xins.api.calling.convention	The name of the default calling convention.

You can also define your own properties. Just add `<bootstrap-properties>` and `<bootstrap-property>` elements to the `impl.xml` file.

Example:

```
<impl>
  <bootstrap-properties>
    <bootstrap-property name="xiff.login.page">Login</bootstrap-property>
    <bootstrap-property name="xiff.default.command">DefaultCommand</bootstrap-prop
  </bootstrap-properties>
</impl>
```

Note

Bootstrap properties starting with `org.xins.` are reserved except for the `org.xins.server.config` bootstrap property. This property could be set to indicate the location of the runtime properties file in the case that you cannot pass it in the command line of the Servlet container.

The values of the properties will be available in the `bootstrapImpl(Map<String, String> bootstrapProperties)` method of your functions and shared instances.

Note

Since XINS 2.2 it is also possible to define in `impl.xml` extra XML elements you want to add to the generated `web.xml` file. Tou can think at elements such as `context-param`, `filter`, `listener`, ...

To do it, use the following syntax (after runtime-properties):

```
<impl>
  <web-app element="context-param">
    <![CDATA[
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/applicationContext.xml,/WEB-INF/daoContext.xml</par
    ]]>
  </web-app>
  <web-app element="listener" id="spring">
  <![CDATA[
    <listener-class>org.springframework.web.context.ContextLoaderListener</list
  ]]>
  </web-app>
</impl>
```

Integration with CVS / Subversion

As you've probably already noticed all of our specification files have `rcsversion="$Revision $"` and `rcsdate="$Date$"` when the files are stored in a control version system such as CVS, the `$Revision$` is replaced by something like `$Revision 1.2 $` and `$Date$` with something like `$Date 2006/11/13 15:16:47 $`. This is particularly useful when you want to keep track of the version and the date of the last changes of the specification.

XINS allows you also to freeze the different parts of the specification. This is particularly useful if the person or department using the API is not the same as the one designing it or implementing the API. This way the different parties can assume the required input or output parameters. If for some reasons a function or a type changes, the specification documentation would mark the function or type in red with a tag "broken freeze".

To freeze a function, a type or a result code add the attribute freeze next to the name in `api.xml`. The value of this attribute should be the version of the frozen specification.

For example

```
<api name="myproject" owner="johnd"
```

```
rcsversion="$Revision: 1.106 $" rcsdate="$Date: 2013/01/23 10:00:59 $">
  <function name="MyFunction" freeze="1.3" />

  <type name="Gender" freeze="1.1" />
  <type name="LastName" />

  <resultcode name="NoVowel" freeze="1.2" />
</api>
```

You can also specify in the `xins-projects.xml` the root of the `cvsweb` location:

```
<cvsweb href="http://cvs.mycompany.com/cvsweb/myprojects" />
```

If this element is provided, broken freeze will provide you with a link where you will be able to see the changes done for this item since it has been frozen. If you use `viewcvs` only the first provided link will work.

Managing dependencies

For advanced APIs, you would probably need to use some external libraries or to use other services. This can be done by adding dependencies to your API.

In `xins-projects.xml` you can add an attribute `dependenciesdir` to indicate in which directory the jar files used by your project will be included. For example

```
<project name="myprojects"
         rcsversion="$Revision: 1.106 $" rcsdate="$Date: 2013/01/23 10:00:59 $"
         dependenciesdir="../other-projects"
         domain="com.mycompany">
</project>
```

In `apis\<api>\impl\impl.xml` you add the jar files needed for your project using the dependency element. For example

```
<impl>
  <dependency dir="utils" />
  <dependency dir="capis" includes="project2-capi.jar" />
  <dependency dir="spring" includes="spring.jar" deploy="false" />
</impl>
```

In the previous example when your implementation will be compiled the following jar files will be in the classpath: `..\other-projects\utils\*.jar;..\other-projects\capis\project2-capi.jar;..\other-projects\xins\xins-client.jar`. These files will also be included in the generated WAR file except for spring.jar. This `deploy` attribute has been introduced in XINS 2.2.

An example of dependencies is provided in XINS with the `filteredproject` demo. This API calls another API (myproject) using CAPI.

Note

The order of the elements in the imp.xml is important, if the elements are not in the correct order the validation of `impl.xml` with the DTD will fail.

The order is `logdoc, bootstrap-properties, runtime-properties, content, dependency, calling-convention, instance.`

Adding other files to the WAR file

It is possible to add any kind of files to the WAR file. This is particulary useful if you want to deploy a web site at the same time as your API.

To do it add content elements to the impl.xml file. Note that you'll need to also add a `web-path` attribute to the impl element because the API is by default mapped with the root path. So in order to access the file you have added to the WAR file, you will need to associate the API with another path.

Example:

```
<impl web-path="store">
  <content dir="apis/petstore" includes="xslt/*.xslt" />
  <content dir="apis/petstore/web" includes="*.html *.js" />
  <content dir="apis/petstore/web" includes="*.jpg" web-path="images" />
</impl>
```

Note

If you only want to add files to the `WEB-INF` directory, for example to use them in your API, then you don't need to specify the `web-path` attribute to the `impl` element.

Multiple implementations and stubs

XINS can support multiple implementations for the same API. To add a new implementation for the API add in the `xins-projects.xml` an `impl` element with a `name` attribute to the api. For example

```
<project name="myprojects"
         domain="com.mycompany">
  <api name="allinone">
    <impl />
    <impl name="mystub" />
  </api>
</project>
```

New targets are created for this implementation. Only server side targets (war, javadoc-api, run, ...) are created for this new implementation. The other targets only depend on the specification which doesn't change. The new target syntax is `<action>-<api name>-<implementation name>`. For example to compile the new implementation of the example above execute **xins classes-allinone-mystub**.

Note

The list of the API names with the possible implementation is printed at the end of the execution of the **xins help** command.

Having the possibility to have more than one implementation is useful when you want to create a stub (fake API) so that other systems can test the API before being implemented.

XINS offers the possibility to generate a stub implementation of the API. The stub is generated based on the examples defined in the function specification.

To generate the stub implementation execute **xins stub-<api name>-<implementation name>**. Then the stub file are generated in the directory `apis\<api name>\impl-<implementation name>\<package name>`. You can also edit the generated code, for example to be more flexible with some input parameters or to return other values. Note that the stub target won't replace existing files.

Testing

XINS offers different ways to test your API.

The first way is using the generated test forms or the links provided with the examples on the function page of the generated specdocs.

XINS offers also the possibility to test the API using the JUnit library. XINS will generate the unit test based on the examples defined in the functions specification. To be able to test your API proceed as follow:

- Add a `test` element to the `api` element in the `xins-projects.xml`. For example:

```
<project name="myprojects"
         domain="com.mycompany">
  <api name="allinone">
    <impl />
    <test />
  </api>
</project>
```

- Run the API. For example: **xins run-allinone**.
- Execute the test target. For example: **xins test-allinone**.

The test target will generate the unit tests in the directory `apis\<api name>\test\<package name>`. The main test case executed is `APITests.java` which includes the function tests. One example matches one test.

The generated unit tests are editable. You can adapt it to better fit the expected result or to add your own unit tests. Note that the test target won't overwrite files if the `test` directory already exists.

The result of the unit tests is located in the directory `build\testresults`. Results are provided in XML and HTML. The XML contains the configuration settings and the logs generated. The HTML contains the results of the test in a more human readable format.

To be able for the test target to work correctly, you have to copy the `junit.jar` file located in xins `lib` directory to the `lib` directory of ant.

The test target also have an option to run the API before starting the tests. To use it, add `test.start.server=true` to the `build.properties`. If this option is set, you don't need to execute the **xins run-<api name>** command.

If you want to execute the tests on another environment than you local machin, you can specify it in the `build.properties` by setting the properties `test.environment`.

Note

The `<test />` element should be added after the `<impl />` and `<environments />` elements if defined.

Categories

Sometimes an API can contains a lot of functions. It become then more difficult to find the function in the specdocs. XINS 1.3.0 introduced the notion of categories. With categories you can specify functions that belong together.

Example:

Define the category names at then end of `api.xml`:

```
<api name="allinone">
...
  <category name="DataSections" />
</api>
```

Then create a category file with the name `<category name>.cat`. This file will define the function that are in the same group.

Example `DataSections.cat`:

```
<?xml version="1.0" encoding="US-ASCII"?>
<!DOCTYPE category PUBLIC "-//XINS//DTD Category 1.3//EN" "http://www.xins.org/dtd
<category name="DataSections">
  <description>Data section related functions.</description>
  <function-ref name="DataSection" />
  <function-ref name="DataSection2" />
  <function-ref name="DataSection3" />
</category>
```

Now if you regenerate the specdocs, The categories will appear at the top of the API page. You can then more easily find the function you wanted.

Managing logs

XINS includes it's own logging system called logdoc which is based on log4j.

Managing the logs on the server side.

You can also manage the debugging settings of XINS by changing the xins.properties file. Here are the main logging settings that can be changed:

Table 4. Log4j properties

Property	Value examples	Description
log4j.rootLogger	DEBUG, console, logfile	Indicates the level of logging and the modules that will receive the output.
log4j.<api name>.rootLogger	INFO, logfile2	Indicates the level of logging and the modules that will receive the output for the specified API.
log4j.appender.console	org.apache.log4j.ConsoleAppender	Indicates where the output is redirected for the module.
log4j.appender.console.layout	org.apache.log4j.PatternLayout	Indicates the pattern for the output header (text that precedes the output text).
log4j.appender.console.layout.ConversionPattern	%d %-5p %c %x - %m%n	Indicates the format of the output header.

log4j.logger.org.xins.common.expiry.3401	ERROR	Changes the log level for a message. This can be useful if you want to hide a message or if you want to show a message.

The log levels are `DEBUG` < `INFO` < `NOTICE` < `WARN` < `ERROR` < `FATAL`. This means that if you set the log level to `ERROR`, only the `ERROR` and `FATAL` messages will be logged.

For more information on the possible characters in the conversion pattern, visit the log4j website [http://logging.apache.org/log4j/docs/api/org/apache/log4j/PatternLayout.html].

The nested diagnostic context (NDC - %x) is set to the value of the `_context` parameter of the query. If no `_context` parameter is passed a default context is created as `apiName@localHost:yyyyMMdd-HHmmssNNN:random` where:

- apiName is the name of the API.
- localHost is the IP address of the computer that is running the api.
- yyyyMMdd-HHmmssNNN is the date of the request (year - month - day - hour - minute - second - millisecond).
- random is a 5 digits long random hexadecimal generated number.

For more information on the possible logging properties and values, visit log4j documentation page [http://logging.apache.org/log4j/docs/manual.html].

Adding your own logs.

It's sometimes useful to log events that may happen in your function. To do so, you need to create what is called a logdoc (log documentation).

To create a new logdoc execute **xins create-logdoc**.

The command will ask you for the name of your api. The script will then create a new `log.xml` file in the directory `apis\<api name>\impl` with the content:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log PUBLIC "-//XINS//DTD XINS Logdoc 2.3//EN"
          "http://www.xins.org/dtd/log_2_3.dtd">

<log>

  <translation-bundle locale="en_US" />

  <group id="exampleid" name="Example">
    <entry id="10000" level="DEBUG">
      <description>Example of logdoc with some parameters.</description>
      <param name="parameter" />
      <param name="number" nullable="false" type="int32" />
    </entry>
    <entry id="10001" level="ERROR" exception="true">
      <description>Example with an exception.</description>
    </entry>
  </group>
```

```
</log>
```

The group id is used for the creation of the log key (e.g. `com.mycompany.myproject.api.exampleid.10001`) and the group name is used for the description of the group.

It also creates a default translation file `translation-bundle-en_US.xml` with the content:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE translation-bundle PUBLIC "-//XINS//DTD XINS Translation Bundle 2.3//EN"
          "http://www.xins.org/dtd/translation-bundle_2_3.dtd">

<translation-bundle>
  <translation entry="10000">Example of logdoc with the parameter <value-of-param
  <translation entry="10001">Example of an exception.</translation>
</translation-bundle>
```

You can now use the logdoc in your function implementation by calling the method `Log.log_10000(myParam, 36);`

The command will also add the element `<logdoc />` to the impl element of the `impl.xml` file. If you want to use the logs in packages other than the default one add the attribute `accesslevel="public"` to the `logdoc` element.

The Logdoc consist of a main `log.xml` file which defines the possible locale for the logs of this API and the groups of entries. Each entry has an `id` that is by convension a number higher than 10000 as numbers between 1000 and 9999 are reserved for XINS. Ids don't have to be consecutive numbers. It's even advisable to have numbers gaps between groups.

If the entry has the attribute `exception` with the value `true`, then the exception should be the first parameter when invoking the method. For example the id 10001 should be invoked with `Log.log_10001(myException);`.

The possible types for a parameter are `boolean`, `int8`, `int16`, `in32`, `int64`, `text` and `object`.

By default for a parameter the nullable attribute is set to `true` and the type attribute is set to `text`.

These types match respectively in Java the types `boolean`, `byte`, `short`, `int`, `long`, `String` and `Object`.

If you use logdoc in your API, the list and the description of the defined log entries will be available in the specification documentation.

In the `log.xml` the warning level is set with `level="WARNING"` but the level is translated as the `WARN` log4j level.

Note that when the specification documentation (specdocs) is generated the HTML pages describing the logging message is also generated. You can see the XINS logdoc at http://xins.sourceforge.net/logdoc.html.

Configuring the properties files for specific logs

Sometimes too much log information is printed and you would like to remove some of the messages printed by the log.

There are several ways to do this. The best ways is to hide the logging messages you don't want.

The following example will hide the `HTTPClient DEBUG` logs:

```
log4j.logger.org.apache.http=INFO
```

The following example will hide all XINS messages which are not at least at the `WARN` level:

```
log4j.logger.org.xins=WARN
```

The following example will hide the specific logging message because the message is at `DEBUG` level:

```
log4j.logger.org.xins.server.lifespan.init.3402=INFO
```

The locale can also be specified in the XINS properties file:

```
# Locale used for logging
org.xins.server.log.locale=en_US
```

The logdoc with the locale `en_US` and `fr_FR` are provided in XINS.

An example of a XINS properties file with a customized logging system can be found in the `demo` directory.

Managing load balancing and fail over

XINS can also manage load balancing and fail over when you want to invoke your function using CAPI. To do so, your function needs to be started on 2 computers. Our code is based on the CAPI example done in the section called “The client API”.

The first possibility would be to use the `GroupDescriptor` (javadoc [javadoc/org/xins/common/service/GroupDescriptor.html]) instead combined with a `TargetDescriptor` (javadoc [javadoc/org/xins/common/service/TargetDescriptor.html]):

```
// Create the descriptor for the service
TargetDescriptor server1 = new TargetDescriptor("http://192.168.0.20:8080/my-proje
TargetDescriptor server2 = new TargetDescriptor("http://192.168.0.21:8080/my-proje
Descriptor[] servers = {server1, server2};

// Create the descriptor for load balancing
GroupDescriptor descriptor = new GroupDescriptor(GroupDescriptor.RANDOM_TYPE, serv
/* If you wanted to have fail over replace the above line with
   GroupDescriptor descriptor = new GroupDescriptor(GroupDescriptor.ORDERED_TYPE,
```

The drawback of this method is that it's not flexible, any changes of IP addresses, time-out or if you want to add a new server, you would need to adapt the code, recompile and then redeploy it.

A better method would be to have the URLs and the time-out in a property file. The only code you need then to create your descriptor is

```
// Build the descriptor
Descriptor descriptor = DescriptorBuilder.build(properties, "capis.myproject");
```

where properties is the `Map<String, String>` object that contains your parameters.

If you only have one server (for example your computer for testing), set the properties as

```
capis.myproject=service, http://127.0.0.1:8080/my-project/, 20000
```

If you want to have load balancing, set the properties as

```
capis.myproject=group, random, server1, server2
capis.myproject.server1=service, http://192.168.0.20:8080/my-project/, 20000
capis.myproject.server2=service, http://192.168.0.21:8080/my-project/, 20000
```

If you want to have fail over, set the properties as

```
capis.myproject=group, ordered, server1, server2
capis.myproject.server1=service, http://192.168.0.20:8080/my-project/, 20000
capis.myproject.server2=service, http://192.168.0.21:8080/my-project/, 20000
```

It's possible to combine load balancing and fail over. For example in the following code the invocation is balanced between server1 and server2 and if one of both is down back-up will be used.

```
capis.myproject=group, random, system1, system2
capis.myproject.system1=group, ordered, server1, back-up
capis.myproject.system2=group, ordered, server2, back-up
capis.myproject.server1=service, http://192.168.0.20:8080/my-project/, 20000
capis.myproject.server2=service, http://192.168.0.21:8080/my-project/, 20000
capis.myproject.back-up=service, http://192.168.0.22:8080/my-project/, 20000
```

The fail-over is not always executed when an error occur, the condition upon which the fail-over is executed are listed in the XINSServiceCaller Javadoc [javadoc/org/xins/client/XINSServiceCaller.html].

XINS types

the section called “Defining Input, Output.” shows you an example on how to use the types when defining the function.

Predefined types.

XINS has some integrated types which facilitate the use of input and output parameters with the implementation.

Table 5. XINS integrated types

type	Java representation	Example
_text	java.lang.String	hello world
_boolean	boolean	true
_int8	byte	25
_int16	short	2004
_int32	int	15
_int64	long	654654132135544566
_float32	float	25.6
_float64	double	3.14159
_date	org.xins.common.types.standard.Date.Value	20040514
_timestamp	org.xins.common.types.standard.Timestamp.Value	20040514115930
_property	Map<String, String>	upgrade%3Dtrue%26surname %3Dde%2BHaan

_list	org.xins.common.types.standard.ListValue	item1%26item2%26item1
_set	org.xins.common.types.standard.SetValue	item3%26item1%26item2
_base64	byte[]	aGVsbG8=
_hex	byte[]	546573746f
_url	String	http://www.google.com

_descriptor	org.xins.common.service.Descriptor	descriptor=group, random, target1, target2
		
		descriptor.target1=service, http://127.0.0.1:8080/my-project/, 8000
		
		descriptor.target2=service, http://192.168.0.1:8080/my-project/, 8000

_xml	org.xins.common.xml.Element	<firstName>Alain</firstName>

You can find an example using all these types in the function `demo\xins-project\apis\allinone\spec\SimpleTypes.fnc`.

Defined types

As seen in the section called “Defining Input, Output.”, it's also possible to define your own type. Here is a list of the possible types to define in XINS:

- `<pattern>`: This type accepts a PERL5 regular expression and allows you to define a text that has some constraints in its format.
- `<enum>`: This type must contain a list of `<item>` elements, each item element must have a `value` attribute that contains the value of the item and can have a `name` attribute that contains the description of the item. If the name is not defined it's set by default to the value of the item.
- `<int8>`, `<int16>`, `<int32>`, `<int64>`: These types define a number. They accept the optional attributes `min` and `max` that can be used to set a minimum and a maximum value for the parameter.
- `<float32>`, `<float64>`: These types define a floating point number. They accept the optional attributes `min` and `max` that can be used to set a minimum and a maximum value for the parameter.
- `<properties>`: This type is similar to the `_property` type but it allows you to specify a type for the keys and the values of the property by using the optional attributes `nameType` and `valueType`. If not specified the type `_text` will be used.
- `<list>` and `<set>`: These types are used to define a list of values in a parameter. The type of the value can be restricted by using the optional attribute `type`. If not specified the default value for the `type` attribute is `_text`.
- <xml>: This type is used to specify an XML paramter. It accept the attribute *`xsdLocation`* that can be a URL or a local file in the `spec` directory.
- `<base64>`, `<hex>`: This type defines a binary. They accept the optional attributes `min` and `max` that can be used to set a minimum and a maximum size in bytes for the binary.

You can find an example using some of these types in the function `demo\xins-project\apis\allinone\spec\DefinedTypes.fnc`.

Shared types

If you have more than one API, you may want to reuse types defined in other APIs (for example EMail.typ, Account.typ, LastName.typ, ...). Instead of copying the file to the new API, you can also point to this type. This is done by using `<api name>/<type name>` in api.xml and in the function(s) specification.

For example:

```
<type name="petstore/EMail"/>
```

```
<param name="outputEMail" required="false" type="petstore/EMail">
  <description>An example of output shared type.</description>
</param>
```

Grouping types

Param combos

It's also possible in XINS to defined a group of `input` or `output` parameters. There are 4 kinds of group:

- `inclusive-or`

  If a group is `inclusive-or` then at least one of the parameters defined in the `param-ref` should have a value.

- `exclusive-or`

  If a group is `exclusive-or` then one and only one of the parameters defined in the `param-ref` should have a value.

- `all-or-none`

  If a group is `all-or-none` then either none of the of the parameters defined in the `param-ref` should have a value or all.

- `not-all`

  If a group is `not-all` then at least one of the parameters defined in the `param-ref` should not have a value.

Example:

```
<input>
  <param name="birthYear" required="false" type="_int32">
    <description>The birth date's year.</description>
  </param>
  <param name="birthMonth" required="false" type="_int32">
    <description>The birth date's month.</description>
  </param>
  <param name="birthDay" required="false" type="_int32">
    <description>The birth date's day.</description>
  </param>
  <param name="birthCountry" required="false" type="_text">
    <description>The country where the person is born.</description>
  </param>
  <param name="birthCity" required="false" type="_text">
    <description>The city where the person is born.</description>
  </param>
  <param name="age" required="false" type="Age">
    <description>An example of input for a int8 type with a minimum and maximum.</
  </param>
  <!-- One of the two parameters must be filled but not both -->
  <param-combo type="exclusive-or">
    <param-ref name="birthYear" />
    <param-ref name="age"       />
  </param-combo>
  <!-- At least one of the two parameters must be filled -->
  <param-combo type="inclusive-or">
    <param-ref name="birthCountry" />
    <param-ref name="birthCity"    />
  </param-combo>
  <!-- These parameters must be filled together or not filled at all -->
  <param-combo type="all-or-none">
    <param-ref name="birthYear"  />
    <param-ref name="birthMonth" />
    <param-ref name="birthDay"   />
  </param-combo>
</input>
```

If the condition is not met, XINS will return a result with an `_InvalidRequest` error code if the request does not match the input `param-combo` or an `_InvalidReponse` error code if the result does not match the output `param-combo`.

You can find an example using the groups in the function `demo\xins-project\apis\allinone\spec\ParamCombo.fnc`.

Attribute combos

It is also possible to defined similar constraint on attributes of elements defined in the data section (See next chapter). This is done using the `attribute-combo` and the `attribute-ref` elements.

Example:

```
<data>
  <contains>
    <contained element="person" />
  </contains>
  <element name="person">
    <description>A person</description>
    <attribute name="birthCountry" required="false" type="_text">
      <description>The country where the person is borned.</description>
    </attribute>
    <attribute name="birth-city" required="false" type="_text">
      <description>The city where the person is borned.</description>
    </attribute>
    <attribute name="age" required="false" type="Age">
```

```
      <description>An example of input for a int8 type with a minimum and maximum.
    </attribute>
    <!-- At least one of the two attributes must be filled -->
    <attribute-combo type="inclusive-or">
      <attribute-ref name="birthCountry" />
      <attribute-ref name="birth-city" />
    </attribute-combo>
  </element>
</data>
```

Combos based on values

It is also possible to define constraints based on the value of one parameter (and not only on whether the parameter is set or not). This is done by added the value attribute to the param-ref element or attribute-ref element for attribute-combo.

Example:

```
<input>
  <param name="salutation" required="true" type="Salutation">
    <description>The gender of the person.</description>
  </param>
  <param name="maidenName" required="false" type="_text">
    <description>The maiden name.</description>
  </param>
  <!-- If the salutation is Madam, the maiden name is required -->
  <param-combo type="inclusive-or">
    <param-ref name="salutation" value="Madam" />
    <param-ref name="maidenName" />
  </param-combo>
</input>
```

The description of the type associated with the param-combo is then a bit different:

- `inclusive-or`

  If the parameter has the defined value then the second parameter is required.

- `exclusive-or`

  If the parameter has the value then the other parameter should not be filled, otherwise the second parameter is required.

- `all-or-none`

  If the parameter has the value then the others parameters are required, otherwise the other parameters should not be set.

- `not-all`

  If the parameter has the value then the other parameters should not be set.

For a better understanding, look at the `ParamComboValue.fnc` function provided in the `allinone` API.

It is also possible to have more than one `param-ref` with a value in the same `param-combo`.

The data section.

The data section defines a tree structure in the input or output section.

The specification

Example:

```
<data>
  <contains>
    <contained element="property" />
  </contains>
  <element name="property">
    <description>A property name and value.</description>
    <attribute name="name" required="true" type="PropertyName">
      <description>The property name.</description>
    </attribute>
    <attribute name="value" required="true" type="_text">
      <description>The property value.</description>
    </attribute>
  </element>
</data>
```

An element can also contain another element or text(pcdata) but not both.

To add an element in another element, add

```
<data>
  <contains>
    <contained element="property" />
  </contains>
  <element name="property">
    <description>A property name and value.</description>
    <contains>
      <contained element="product" />
    </contains>
    <attribute name="name" required="true" type="PropertyName">
      <description>The property name.</description>
    </attribute>
  </element>
  <element name="product">
...
```

If you want your element to contain `PCDATA`, use

```
<data>
  <contains>
    <contained element="property" />
  </contains>
  <element name="property">
    <description>A property name and value.</description>
    <contains>
      <pcdata />
    </contains>
    <attribute name="name" required="true" type="PropertyName">
      <description>The property name.</description>
    </attribute>
...
```

Since XINS 1.1.0, it's also possible to include more than one element in the data section:

```
<data>
  <contains>
    <contained element="packet" />
    <contained element="letter" />
  </contains>
  <element name="packet">
    <description>The packet.</description>
    <attribute name="destination" required="true" type="_text">
      <description>The destination of the packet.</description>
    </attribute>
  </element>
  <element name="letter">
    <description>The letter.</description>
    <contains>
      <pcdata />
    </contains>
    <attribute name="destination" required="true" type="_text">
      <description>The destination of the letter.</description>
    </attribute>
  </element>
</data>
```

Attributes can also have a default value using the `default` attribute.

The implementation

The data section is translated into a Java object with the appropriate set methods. Once the object is created, you can add them to the `SuccessfulResult` using provided add methods.

Example:

```
SuccessfulResult result = new SuccessfulResult();
Property myProperty = new Property();
myProperty.setName(myName);
myProperty.setValue(myValue);
result.addProperty(myProperty);
return result;
```

If you had specified that your element could contain another element, the `Property` class would also have a `addProduct(Product)` method.

If you had specified that your element could contain a text, the `Property` class would also contain a `pcdata(String)` method.

The result

To view what kind of result to expect from a data section, we will define an example in the specification of the function.

```
<output-data-example>
  <element-example name="property">
    <attribute-example name="name">upgrade</attribute-example>
    <attribute-example name="value">true</attribute-example>
  </element-example>
  <element-example name="property">
    <attribute-example name="name">surname</attribute-example>
    <attribute-example name="value">Doe</attribute-example>
  </element-example>
</output-data-example>
```

The resulting XML from XINS would then be

```
<result>
  <data>
    <property name="upgrade" value="true" />
    <property name="surname" value="Doe" />
  </data>
</result>
```

If your element accepted `PCDATA`, you can set a value by adding `<pcdata-example>My value</pcdata-example>` before the `<attribute-example>`.

You can find some examples of the output data section in the functions `demo\xins-project\apis\allinone\spec\DataSection.fnc` and `demo\xins-project\apis\allinone\spec\DataSection2.fnc`.

Data section for the input

Since XINS 1.1.0 it's also possible to send a data section for the request. The definition of the data section is similar to the definition for the output section except that it must be done in the input section. The definition of the example is done using the `<input-data-example>` element.

This allows you to send complex structures with XINS in the request.

The framework will generate some extra objects and methods to get the values of the input data section. Here is an example:

```
import java.util.Iterator;
...
Iterator itAddresses = request.listAddress().iterator();
while (itAddresses.hasNext()) {
  Request.Address nextAddress = (Request.Address) itAddresses.next();
  System.out.println(nextAddress.getPostcode());
}
```

An example is provided with XINS in the function `demo\xins-project\apis\allinone\spec\DataSection3.fnc`.

Since XINS 1.5.0, if you are using Java 1.5 and deploying on a Java 1.5 application server, you can also benefit from the Java generics feature as demontrated at the end of the next section.

Data section on the client side

On the client side the data section is retreived as an `org.xins.common.xml.Element` object (javadoc [javadoc/org/xins/common/xml/Element.html]). To get the values in the returned data section, use the methods provided in the `Element` class such as `getAttribute(String localName)` or `getChildElements(String name)`.

If you want to send a data section as input, you will need to create an `org.xins.common.xml.Element` (javadoc [javadoc/org/xins/common/xml/Element.html]) for example by using the `org.xins.common.xml.ElementBuilder` (javadoc [javadoc/org/xins/common/xml/ElementBuilder.html]) class. Then you can pass this object to the CAPI call method or to the generated Request object.

Since XINS 1.3.0, the methods and objects are also generated on the client side for the data section. They are generated the same way as they are on the client side. For example, if you want to send a list of address and receive a list of properties, this will look like this:

- For the request:

  ```
  import com.mycompany.myapi.capi.MyFunctionRequest;
  ...
  MyFunctionRequest request = new MyFunctionRequest();
  MyFunctionRequest.Address address1 = new MyFunctionRequest.Address();
  address1.setCompany("McDo");
  address1.setPostcode("12345");
  request.addAddress(address1);
  ```

- For the result:

  ```
  import java.util.Iterator;
  import com.mycompany.myapi.capi.MyFunctionResult;
  ...
  MyFunctionResult result = capi.callMyFunction(...);
  Iterator itProperties = result.listProperty().iterator();
  while (itProperties.hasNext()) {
    MyFunctionResult.Property nextProperty = (MyFunctionResult.Property) itPropert
    String propertyName = nextProperty.getName();
    String propertyValue = nextProperty.getValue();
    System.out.println(propertyName + ": " + propertyValue);
  }
  ```

  If you are using Java 1.5 or higher and did not set the build.java.version property to a lower version, the generated classes will use the generics feature added to the Java language since Java 1.5. This will simplified your code of using the code:

  ```
  import com.mycompany.myapi.capi.MyFunctionResult;
  ...
  MyFunctionResult result = capi.callMyFunction(...);
  for (MyFunctionResult.Property nextProperty : result.listProperty()) {
    String propertyName = nextProperty.getName();
    String propertyValue = nextProperty.getValue();
    System.out.println(propertyName + ": " + propertyValue);
  }
  ```

Function accesses

XINS includes a way to set some permissions for the functions using ACLs and also a way to disable/enable a function.

ACLs

The ACLs are used to restrict the access of a function based on the IP address from where the request comes.

The ACLs are defined in the `xins.properties` file with the org.xins.server.acl property

The value is a dot comma separated list of the keywords `allow` or `deny`, the IP addresses specified as ACL allowed or denied to access the function and the name of the function or * used for all functions.

An ACL is an IP address followed by / and the number of bits that should remains the same. For example `192.168.0.0/24` defines all IP addresses starting with `192.168.0.`

Example:

```
org.xins.server.acl=allow 127.0.0.1 *; \
                    allow 192.168.0.0/24 MyFunction
```

Per default, if an IP address is not specified in the list then the access is denied. If an IP address is specified twice then the first rule will apply. If you specify `/0` after an IP address then all IP address will match.

Example:

```
org.xins.server.acl=allow 127.0.0.1 *; \
                    deny 192.168.0.21 _GetSettings; \
                    allow 192.168.0.21 _*; \
                    allow 192.168.2.0/24 _*; \
                    allow 0.0.0.0/0 _GetVersion; \
                    allow 192.168.0.0/24 MyFunction
```

In this example, IP addresses starting with 192.168.0. will be able to access `MyFunction`, the IP address 192.168.0.21. will also be able to access the meta functions except the `_GetSettings` meta function. All IP addresses starting with 192.168.2. will be able to access the meta functions. Everybody will be able to access the `_GetVersion` meta function.

Since XINS 1.1.0, the keyword `file` is also accepted with as second argument the location of the file containing the permissions. The specified file should be of a special format. The lines should start with allow, deny or file. If the line start with allow or deny it should be followed by the ACL and the function as shown in the previous example. If the line starts with file, it should be followed by the location of another acl premission file. Empty lines, lines containing only spaces and lines starting with # are ignored. The ACL files will be monitored for changes every `org.xins.server.config.reload` seconds and will be reloaded when the meta function `_ReloadProperties` is invoked.

Example:

```
org.xins.server.acl=allow 194.134.168.0/24 _*;\
                    file /usr/conf/myApp.acl
```

`myApp.acl`:

```
allow 194.134.168.0/24 *
deny 194.134.32.0/24 _*
allow 194.134.32.0/24 *

# comment...
```

```
allow 212.129.129.120 GetKey
```

Since XINS 2.1, it is possible to allow or deny a call based on the calling convention used. To do it add after the name or the pattern of the function, the name or the regular expression pattern of the calling convention you want to allow or deny.

For example:

```
org.xins.server.acl=allow 194.134.168.0/24 _* _xins-std|_xins-xslt;\
                    deny 0.0.0.0/0 _*;\
                    deny 0.0.0.0/0 * _xins-soap
```

allows the meta functions to be called only using the _xins-std or _xins-xslt calling conventions for the given IP range and denies any call using the _xins-soap calling convention.

Enable/Disable a function

It's also possible to enable or disable a function. By default all functions are enabled.

To disable a function, request the following URL: `http://API_PATH?_function=_DisableFunction&functionName=MyFunction`

To re-enable the function, request the URL: `http://API_PATH?_function=_EnableFunction&functionName=MyFunction`

The links to enable or disable a function are provided on the test form generated with the specification documentation.

HTTPS

It's also possible to use HTTPS as communication layer to call a XINS API. To do it, you just need to configure the HTTP server (such as Apache) or the servlet container (such as Tomcat) with the correct settings.

For more information on setting up the server, read the following articles:

- Configuring Tomcat 4.0 to use HTTPS [http://www.dga.co.uk/customer/publicdo.nsf/0/2B4063F90912CC5D85256CB00007888B?OpenDocument]
- Tomcat 4.0 SSL configuration How-To [http://jakarta.apache.org/tomcat/tomcat-4.0-doc/ssl-howto.html]
- How can I support HTTPS (SSL) in a servlet? [http://www.jguru.com/faq/view.jsp?EID=53931]

Note that HTTPS is a supported protocol on the client side only since XINS 1.3.0.

The meta functions

When you're starting a XINS server, a set of meta functions are already available. We just saw in the Enable/Diable section the first two `_DisableFunction` and `_EnableFunction`. Note that all meta functions start with an underscore.

Table 6. XINS Meta functions

Name	Description

_NoOp	No operation: This meta function does nothing but maybe useful to monitor if the server is up or down or the response time of the server.
_GetVersion	Get the running version of xml-enc, XINS, Java and of the API if defined.
_GetSettings	Returns the properties as set in the runtime property file and the Java properties as returned by `System.getProperties()`.
_GetStatistics	Returns the amount of memory used and for each function, the number of successful call, and unsuccessful call with the response time (min, max, last, average). This function may have a parameter `reset=true` that performs a reset of the statistics at the same time. This function may also have a parameter `detailed=true` that displays the statistics per error code for the unsuccessful results.
_ResetStatistics	Resets the statistics.
_ReloadProperties	Reloads the XINS properties file.
_CheckLinks	Checks that the API can access the other API's or URL it's using. This meta function will test the URLs set in the runtime properties file where the property is defined in the `impl.xml` file with the type `_url` or `_descriptor`.
_GetFunctionList	Returns the list of the functions.
_DisableFunction	Disables the function passed in the functionName input parameter.
_EnableFunction	Enables the function passed in the functionName input parameter.
_DisableAPI	Disables the API by returning HTTP 503 to all requests.
_EnableAPI	Re-enables the API that was disabled.
_WSDL	Returns the WSDL [http://www.w3.org/TR/wsdl] of the API.
_SMD	Returns the Simple Method Description [http://dojo.jot.com/SMD] of the API.

Management

As XINS APIs are standard Servlets, most of the management can be done by the Servlet container or the J2EE application server. Most of the management specific to the API is done by editing the runtime property file. See .

JMX

JMX MBeans have been added since XINS 1.5.0 to have the possibility to manager the API using standard management tools sush as HP Openview or the JConsole included in the JDK.

The following information are available through the JMX interface:

- Statistics
- Bootstrap properties
- Runtime properties
- XINS version running
- API version
- Startup time
- The list of the functions

You can also perform the following actions:

- noOp(): Function that does nothing but can be used to check if the API is up.
- reloadProperties(): Reload the runtime properties.

Log4J `HierarchyDynamicMBean` is also registered as MBean.

The object names used for both MBeans starts with `org.xins.server.<api name>`.

By default, JMX is disabled. To enable it set the runtime property org.xins.server.jmx=true.

To have an overview execute **xins run-myproject** with Java 1.5 or later and run the `jconsole.exe` located in `jdk\bin`.

Versioning

You can also set a version to your API.

To set the API version create a `.version.properties` file in the project directory with the 2 following properties defined: `version.major` and `version.minor`.

The version will be apply to all APIs. If you want a specific version for one of the API, create a `.version.properties` in the `apis\<api name>` directory.

Shared instance

In XINS the functions are initialised when you start the server, the initialisation also passes the properties defined in `xins.properties` to the function. XINS has the notion of shared instance which is an object created and initialized in a similar way as functions are. Furthermore this object is passed as an instance to all the defined functions of the API.

You can think about the following situations:

- You want to access a database or an access to LDAP with the access properties set in `xins.properties`.

- You want to share data between functions.

To add a shared instance to the API add in the `impl.xml` file `<instance name="_sharedObject" getter="getSharedObject" class="SharedObject" />`.

Now you need to create a class `SharedObject` in the implementation package that extends the `org.xins.common.manageable.Manageable` class (javadoc [javadoc/org/xins/common/manageable/Manageable.html]).

Now you could use this object by calling in the call(Request request) method

```
// Get some data from the shared object
String firstName = _sharedObject.getFirstName(request.getLastName());
```

An example is provided in the `allinone` project.

Calling Convention

Since version 1.0.1, XINS has the notion of calling convention.

A calling convention specified the format of the request and the format of the result. You can see it as a communication protocol.

XINS 1.1.0 includes the following calling conventions: `_xins-std`, `_xins-old` and `_xins-xml`.

XINS 1.2.0 includes the `_xins-xslt` calling convention and allows to define it's own custom calling convention.

XINS 1.3.0 includes the `_xins-soap` and `_xins-xmlrpc` calling conventions.

XINS 1.4.0 includes the automatic detection of the calling convention meaning that if no _convention parameter is sent and no custom calling convention is defined, XINS will try to detect the calling convention to call based on the format of the request.

XINS 1.5.0 includes the XINS front-end calling convention: xinsff. A front-end framework user guide [frontend/index.html] is also included in the release. It is also possible since this version possible to define the supported HTTP methods for a custom calling convention. This can be useful if you want to define a REST [http://en.wikipedia.org/wiki/REST] calling convention. A new example was also added that uses a `MultipartCallingConvention` to receive binaries files.

XINS 2.0 includes the `_xins-json` and `_xins-jsonrpc` calling conventions. The `_xins-old` calling convention has been removed in this release.

XINS 2.1 includes the `_xins-soap-map` calling convention.

How to define the calling convention

If nothing is specified the standard calling convention (as in XINS 1.0.0) is used.

If the calling convention can be set, by adding the `_convention` parameter to the URL with the value of the calling convention as argument.

The default calling convention for an API can be specified in the `impl.xml` file by adding the `calling-convention` element. For example:

```
<impl>
```

```
  <calling-convention name="_xins-xml" />
</impl>
```

Standard calling convention

The standard calling convention or also named POX-RPC calling convention is the default calling convention used by xins. If no calling-convention is included in the `impl.xml` file and no *`_convention`* parameter is passed in the request, XINS will expect an request using this convention and return a result using this convention. The property value for the POX-RPC calling convention is `_xins-std`. This calling convention is also known as the standard calling convention.

This examples provided in this document are using the POX-RPC calling convention. The examples generated with the specdocs and the test forms are also using this calling convention.

A document (link [protocol/index.html]) is also provided in the `docs\protocol` directory that contains the specifications of this calling convention.

XSLT calling convention

The property value for the XSLT calling convention is `_xins-xslt`. The request is similar to the standard calling convention (using URL). The result returned is the result of the processing of the XML normally returned with the standard calling convention with a specified XSLT file. This means that the result could be a HTML page or XML or plain text or binary depending on the XSLT transformation.

The location of the XSLT stylesheets is set using the runtime property `templates.<api name>.xins-xslt.source`. This property refers to the directory where the XSLT stylesheets can be found. The stylesheets then must have the name of the function with the `.xslt` extension.

You can also pass the location of the template in the request with the *`_template`* parameter. This will override the previous runtime property. The location should be relative to the path or URL specified in the runtime property `templates.<api name>.xins-xslt.parameter.prefix`. If this property is not set the *`_template`* parameter is not allowed.

The XSLT calling convention caches the templates for better performances. You can clear the cache by passing the *`_cleartemplatecache=true`* parameter. You can also disable the cache by setting the `templates.cache` runtime property to `false`.

SOAP calling convention

The property value for he SOAP calling convention is `_xins-soap`. The SOAP calling convention allows you to call the API using SOAP. SOAP is a W3C [http://www.w3.org/TR/soap/] standard communication protocol for Web Services. The SOAP calling convention allows to provide to external software and companies a way to communicate using a standard protocol. The SOAP calling convention allows also to use XINS with WS-BPEL [http://www.oasis-open.org/committees/wsbpel/charter.php] or JBI [http://java.sun.com/integration/].

Example of an input request:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" xmlns:ns0="u
  <soap:Body>
    <ns0:DataSection3Request>
      <inputText>Hello</inputText>
      <data>
```

```
        <address company="McDo" postcode="1234" />
        <address company="Drill" postcode="4567" />
      </data>
    </ns0:DataSection3Request>
  </soap:Body>
</soap:Envelope>
```

SOAP is often associated with a WSDL file that describes the Web Service. XINS can generate this WSDL file by executing **xins wsdl-<api name>**. This generated file is located in the directory `build\webapps\<api>`. Note that the generated WSDL file also contain the XML Schema for the standard types, the defined types, the required fields or the optional fields. By default the endpoint defined in the WSDL is the first environment defined in the environment XML file or if there is no file defined `http://localhost/<api name>/?_convention=_xins-soap`. You can set the endpoint by passing the command line argument `-Dwsdl.enpoint` or by setting the wsdl.endpoint property in `build.properties` file. You can then use this generated file in the Web Services client sush as C#, Visual Basic, PHP, Perl.

The SOAP calling convention and the generation of the WSDL have been developed based the WS-I Basic Profile Version 1.1 [http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html]to ensure better compatibility with other frameworks.

XML calling convention

The property value for the XML calling convention is `_xins-xml`. In this case only HTTP POST are accepted. The returned output is the same as the standard calling convention. Example of an input request:

```
<request function="functionName">
  <param name="paramName">paramValue</param>
  <data>
    <product name="something1" price="12.2" />
    <product name="something2" price="23.5" />
  </data>
</request>
```

XML-RPC calling convention

The property value for the XML-RPC calling convention is `_xins-xmlrpc`. XML-RPC is a specification used to remote procedure call using XML over HTTP. XML-RPC has client frameworks in a lot of languages including AppleScript, J2ME, Ruby.

Example of an input request:

```
<?xml version="1.0"?>
<methodCall>
  <methodName>SimpleTypes</methodName>
  <params>
    <param><value><struct><member>
    <name>inputBoolean</name>
    <value><boolean>0</boolean></value>
    </member></struct></value></param>
  </params>
</methodCall>
```

For more information on XML-RPC, visit the web site at http://www.xmlrpc.com/.

JSON-RPC calling convention

The property value for the JSON-RPC calling convention is `_xins-jsonrpc`. JSON-RPC is a specification used to remote procedure call in different Ajax frameworks such as the DOJO toolkit. The specification of JSON-RPC are located at http://json-rpc.org/wiki/specification. Both specification version (1.0 and 1.1) are supported by the JSON-RPC calling convention.

Example of an input request for JSON-RPC 1.0:

```
{"method": "ResultCode", "params": ["Hallo"], "id": null}
```

Examples of ouput result for JSON-RPC 1.0:

```
{"result": {"outputTest": "Hallo"}, "error": null, "id": null}
```

```
{"result": null, "error": "AlreadySet", "id": null}
```

Example of an input request for JSON-RPC 1.1:

```
{"version": "1.1", "method": "ResultCode", "params": {"inputText": "Hallo"}}
```

```
http://www.myserver.com/path/ResultCode?inputText=Hallo
```

Examples of ouput result for JSON-RPC 1.1:

```
{"version": "1.1", "result": {"outputTest": "Hallo"}}
```

```
{"version": "1.1", "error": {"name": "AlreadySet", "code": 123, "message": "The pa
```

For more examples, visit the web site at http://json-rpc.org/.

The automatic detection of the matching calling convention for a request will only find request using the 1.1 specifications.

XINS also support JSON-RPC 2.0. The property value for the JSON-RPC calling convention is `_xins-jsonrpc2`. The specification of JSON-RPC 2.0 are located at http://www.jsonrpc.org/specification.

Example of an input request for JSON-RPC 2.0:

```
{"jsonrpc": "2.0", "method": "Echo", "params": { "in": "test 123" }, "id": 4 }
```

Examples of ouput result for JSON-RPC 2.0:

```
{"jsonrpc":"2.0", "result":{"out":"test 123"}, "id":4}
```

```
{"jsonrpc":"2.0", "error":{"code":-32000, "message":"The parameter has already bee
```

JSON calling convention

The property value for the JSON calling convention is `_xins-json`. This calling convention is based on the Yahoo! JSON protocol specified at http://developer.yahoo.com/common/json.html. This includes the support for the callback parameter.

Example of an input request:

```
http://www.myserver.com/path/ResultCode?inputText=Hallo&output=json
```

Examples of ouput result:

```
{"outputTest": "Hallo"}
```

Example of an input request with callback:

```
http://www.myserver.com/path/ResultCode?inputText=Hallo&output=json&callback=proce
```

Examples of ouput result with callback:

```
process({"outputTest": "Hallo"})
```

This calling convention could be useful if you want to call the API from GWT [http://code.google.com/webtoolkit/] (Google Web Toolkit).

SOAP-MAP calling convention

The property value for the SOAP-Map calling convention is `_xins-soap-map`. This calling convention accepts SOAP requests and applies the same mapping for the request or the response as what is done in the **xins wsdl-to-api** command.

This means that if you need to write a web service using a pre-defined WSDL, you can do it by executing `xins wsdl-to-api` command and pass *`_convention`*=*`_xins-soap-map`* in the URL query parameters.

The calling convention will copy the Envelop and Body XML elements as what received in the request, try to keep the same namespaces prefix and map data element attributes to sub-elements. If after the transformation not all parameters are detected and mapped correctly, you can also extend the org.xins.server.SOAPMapCallingConvention and implement you own parsing on specific parts or create your specific response.

Custom calling convention

To create your own calling convention, you need to extend the `org.xins.server.CustomCallingConvention` class (javadoc [javadoc/org/xins/server/CustomCallingConvention.html]).

This class has two abstract methods, `convertRequestImpl` and `convertResultImpl`, that you must implement.

Note that the `CustomCallingConvention` extends the `Manageable` class, this means that you can read the bootstrap settings in the `bootstrapImpl` method and the runtime properties in the `initImpl` method.

To define the calling convention in the `impl.xml`, you need to add an extra argument `class` to the `calling-convention` element. For example:

```
<impl>
  <calling-convention name="my-convention" class="com.mycompany.myproject.MyConven
</impl>
```

You can also define more than one calling convention, in this case one calling convention must have the attribute `default="true"`.

Example:

```
<impl>
  <calling-convention name="_xins-std" default="true" />
  <calling-convention name="my-convention" class="com.mycompany.myproject.MyConven
</impl>
```

More information on how to create your own calling convention is provided in the calling convention primer [http://xins.sourceforge.net/ccprimer.html].

You can also look at the `fileupload` API provided in the examples that defines a `CustomCallingConvention` in order to upload binaries files.

Examples calling convention

Several custom calling conventions are provided in the examples. Some of them are not included in the core libraries so if you want to use them, you will need to copy the file to your project.

Here is a list of the custom calling convention used in the examples:

- `MultipartCallingConvention` is defined in the `fileupload` API. This calling convention allows to receive files according to the HTTP file upload protocol. This calling convention requires the `commons-fileupload.jar` included in `apis\fileupload\lib` directory.
- `RESTCallingConvention` is defined in the `rest` API. This calling convention only accepts strict REST [http://en.wikipedia.org/wiki/Representational_State_Transfer] requests.
- The `petstore` example uses the `FrontendCallingConvention` which is a custom calling convention added to the core. More information about this calling convention can be found in the XINS Front-end Framework manual [frontend/index.html].

Utility classes

XINS also contains in the org.xins.common package, a set of classes that could be useful for the implementation of your API.

Note also that the following APIs are distributed with all XINS applications: HTTPCore [http://hc.apache.org/httpcomponents-core-ga/index.html], HTTPClient [http://hc.apache.org/httpcomponents-client-ga/index.html], commons logging [http://jakarta.apache.org/commons/logging/], commons codec [http://jakarta.apache.org/commons/codec/], Log4j API [http://logging.apache.org/log4j/docs/index.html], json [http://www.json.org/java/index.html], logdoc [https://github.com/znerd/logdoc], and xmlenc [http://xmlenc.sourceforge.net/]. You can then use these APIs without adding any `dependency` element in the `impl.xml` file.

- Spec package

  The spec package is located in org.xins.common.spec (javadoc [javadoc/org/xins/common/spec/index.html]). This package contains classes that provide information on the specification of the API. You can get the specification from the client side by using `capi.getAPISpecification()` and from the server side using `api.getAPISpecification()`. The method will return a `APISpec` object from which you will be able to get all the specification of the API. You can get information sush as the list of the functions, the type of a parameter in a function, the error code returned by a function.

- HTTPServiceCaller

  The org.xins.common.http.HTTPServiceCaller (javadoc [javadoc/org/xins/common/http/HTTPServiceCaller.html]) is a class that is used to get information from a URL. This class also supports load-balancing and fail-over via the GroupDescriptor such as the XINSServiceCaller does. This class has a call method that requires a HTTPCallRequest as input where you can specify the method to call the URL, the parameters of the URL and the behaviour for the fail-over, and this call returns a HTTPCallResult where you can get the returned text as String, byte[] or as InputStream as well as the HTTP code returned.

- ExpiryFolder

  The org.xins.common.collections.expiry.ExpiryFolder (javadoc [javadoc/org/xins/common/collections/expiry/ExpiryFolder.html]) class allows you to store some properties (key & value pairs) for a limited amount of time. When a property is added, it has the maximum time to live. If the property is not requested during this time then the property is removed. If the property is requested using the get() method then time to live of this property is set again to the maximum.

  The ExpiryFolder is particularly interesting if you want to implement caching of the result. More for look at the online article [http://xins.sf.net/resultcaching.html] explaining how to do result caching for a method.

- BeanUtils

  The org.xins.common.BeanUtils (javadoc [javadoc/org/xins/common/BeanUtils.html]) class allows to copy the properties of a POJO to another POJO. It can useful to fill a generated Request object on the client side or a SuccesfulResult object on the server side. Especially if you use libraries such as JPA (Java Persistence API) or the Spring Framework. It is also useful when you want to forward a request or when the result of a function call should be used as input to call another function.

  If the names of the properties are different, you can also provide a mapping of the properties name as parameter, element name or attribute name. The method will also convert the types whenever possible.

- FileWatcher

  The org.xins.common.io.FileWatcher (javadoc [javadoc/org/xins/common/io/FileWatcher.html]) class allows you to monitor a file for changes. You just need to specify the file to monitor, the interval between two checks of the file and the listener that will be notified when the file has changed. Don't forget to invoke the start() method to start the monitoring of the file.

- ElementList

  The org.xins.common.xml.ElementList (javadoc [javadoc/org/xins/common/xml/ElementList.html]) class allows you to list and get sub DOM elements of a DOM element. As described in the Javadoc, this class has several advantages over NodeList.

- Optimized classes

  The org.xins.common.text (javadoc [javadoc/org/xins/common/text/URLEncoding.html]) package contains classes for URL encoding, URL decoding or creation of regular expression patterns.

Ant tasks

Before using the tasks, you need to define them using the following lines:

```
<property environment="env" />
<taskdef resource="org/xins/common/ant/antlib.xml" classpath="${env.XINS_HOME}/b
```

callxins task

The `callxins` task is used to call a xins API. The result of the call is then stored in properties.

Example:

```
<callxins function="MyFunction"
          apiLocation="http://localhost:8080/myproject/"
          prefix="myapi">
  <param name="gender" value="m" />
  <param name="personLastName" value="Lee" />
</callxins>
```

This task will call the `MyFunction` function of the `myproject` API. The result will be store in the myapi.message Ant property as `MyFunction` just returns one output parameter named *`message`*.

The `prefix` attribute is optional.

If the function returns a data section, the properties will be set as described in the Ant XML property task [http://ant.apache.org/manual/CoreTasks/xmlproperty.html].

xins task

The xins task is used to execute a xins target. If the specifications have changed this target will take care of recreating the build.xml and calling the requested target.

If you execute the script from another directory, you will need to define the `projectdir` optional attribute.

Example:

```
<xins api="myproject" target="specdocs" />
```

Note that the api attribute is optional:

```
<xins target="create-api" />
```

XINS targets

Here is a description of the most useful targets.

Table 7. XINS targets

Target name	Description
run-<api>	Runs the WAR for the API.
war-<api>	Creates the WAR for the API.
specdocs-<api>	Generates all specification documentation for the API.
javadoc-api-<api>	Generates Javadoc API documentation for the API.
server-<api>	Generates the war file, the Javadoc API documentation for the server side and the specdocs for the API.
jar-<api>	Generates and compiles the Java classes for the client-side API.
javadoc-capi-<api>	Generates Javadoc API docs for the client-side API.
client-<api>	Generates the specdocs, the Javadoc API docs for the client side, the CAPI jar file and a Zip file containing all of this.
clean-<api>	Cleans everything for the API.

rebuild-<api>	Regenerates everything for the API.
all-<api>	Generates everything for the API.
wsdl-<api>	Generates the WSDL of the API.
stub-<api>	Generates the API stub.
test-<api>	Generates the unit tests if needed and tests the API.
javadoc-test-<api>	Generates Javadoc for the unit test of the API.
opendoc-<api>	Generates the specification of the API in open document format.
all	Generates everything.
clean	Removes all generated files.
specdocs	Generates all specification docs.
clients	Generates client-<api> for all APIs.
javadoc-capis	Generates the Javadoc for all CAPIs.
javadoc-apis	Generates the Javadoc for all APIs.
wars	Creates the WARs for all APIs.
tests	Runs the unit tests of the APIs that have tests.
create-api	Generates a new api specification file.
create-function	Generates a new function specification file.
create-rcd	Generates a new error code specification file.
create-type	Generates a new type specification file.
create-example	Generates a new example for a function.
create-logdoc	Generates the basic logdoc files for an API.
version or -version	Prints the version of XINS.
help	Prints the possible targets along with the possibles APIs.

XINS tools

XINS can also execute several tools that can improve the quality of your API.

To use it, execute **xins <tool target>**.

The target will ask you upon which API you want to execute the tool. You can also pass the name of the API as a parameter name using *`-Dapi.name=<api name>`* or add an api.name property to the `build.properties` file.

You need to have the required libraries in the `XINS_HOME\lib` directory, unless specified otherwise.

- **xins help-tools**

  Description: Prints the list of the tool targets with its description and its required or optional build properties.

- **xins download-tools**

  Description: Downloads and installs the required libraries needed to execute the tools.

- **xins java2html**

  Description: Generates HTML pages which contain the source code of the API.

  Required library: j2h.jar [http://www.ibiblio.org/maven2/java2html/j2h/1.3.1/j2h-1.3.1.jar]

  Web site: http://www.java2html.de/

  Result location: `build\j2h\<api name>\index.html`

- **xins pmd**

  Description: Analyses the source code of the API. The default rules set is `rulesset/basic.xml,rulesset/unusedcode.xml`. You can override it by settting the pmd.rules property.

  Required library: pmd.jar [http://www.ibiblio.org/maven2/pmd/pmd/3.7/pmd-3.7.jar] and jaxen [http://www.ibiblio.org/maven2/jaxen/jaxen/1.1-beta-11/jaxen-1.1-beta-11.jar]

  Web site: http://pmd.sourceforge.net/

  Result location: `build\pmd\<api name>\index.html`

- **xins checkstyle**

  Description: Checks the source code for compliance to the Java coding convention [http://xins.sourceforge.net/XINS%20Java%20Coding%20Conventions%20-%20v1.0%20-%20July%202006.pdf]. It also perform some analisys of the code.

  Required library: checkstyle.jar [http://www.ibiblio.org/maven/checkstyle/jars/checkstyle-4.1.jar] commons-beanutils.jar [http://www.ibiblio.org/maven/commons-beanutils/jars/commons-beanutils-1.7.0.jar] antlr.jar [http://www.ibiblio.org/maven/antlr/jars/antlr-2.7.6.jar]

  Web site: http://checkstyle.sourceforge.net/

  Result location: `build\checkstyle\<api name>\index.html`

- **xins coverage**

  Description: Reports on the coverage of the API code by the unit tests.

  Required libraries: cobertura.jar [http://www.ibiblio.org/maven2/cobertura/cobertura/1.8/cobertura-1.8.jar], asm.jar [http://www.ibiblio.org/maven2/asm/asm/2.2.1/asm-2.2.1.jar]

  Web site: http://cobertura.sourceforge.net/

  Result location: `build\coverage\<api name>\index.html`

- **xins emma**

  Description: Reports on the coverage of the API code by the unit tests using the EMMA code coverage library.

  Required libraries: emma.jar [http://mirrors.ibiblio.org/pub/mirrors/maven2/emma/emma/2.0.5312/emma-2.0.5312.jar], emma_ant.jar [http://mirrors.ibiblio.org/pub/mirrors/maven2/emma/emma_ant/2.0.5312/emma_ant-2.0.5312.jar]

  Web site: http://emma.sourceforge.net/

Result location: `build\coverage\<api name>\index.html`

- **xins findbugs**

  Description: Analyses the code in order to indentify possible bugs. You will need to set the findbugs.home=<findbugs directory> build property.

  Required library: Install findbugs.zip [http://surfnet.dl.sourceforge.net/sourceforge/findbugs/findbugs-1.1.1.zip].

  Web site: http://findbugs.sourceforge.net/

  Result location: `build\findbugs\<api name>\index.html`

- **xins lint4j**

  Description: Analyses the code in order to indentify possible bugs.

  Required library: lint4j.jar [http://www.jutils.com/maven/lint4j/jars/lint4j-0.9.1.jar]

  Web site: http://www.jutils.com/

  Result location: `build\lint4j\<api name>\lint4j-report.txt` and `build\lint4j\<api name>\lint4j-report.xml`

- **xins jdepend**

  Description: Analyses the package dependency and generates design quality metrics.

  Required library: jdepend.jar [http://www.ibiblio.org/maven/jdepend/jars/jdepend-2.9.1.jar] in the `ANT_HOME\lib` directory.

  Web site: http://clarkware.com/software/JDepend.html

  Result location: `build\jdepend\<api name>\index.html`

- **xins cvschangelog**

  Description: Generates the change log report of the API.

  Required library: None but cvs should be installed.

  Result location: `build\cvschangelog\index.html`

- **xins glean**

  Description: Generates a Glean report for the API. Glean is a tool agregator that produce a feedback repost about the source code. You will need to set the glean.home=<Glean directory> build property. If not already provided, a `apis\<api name>\glean.properties` file is created that you can customize to your wishes.

  Required library: Install glean.zip [http://jbrugge.com/glean/glean-1.2.zip].

  Web site: http://jbrugge.com/glean/

  Result location: `build\glean\index.html`

- **xins jmeter**

Description: Generates JMeter [http://jakarta.apache.org/jmeter/] tests from the examples defined in the functions.

Required library: None.

Web site: http://jakarta.apache.org/jmeter/

Result location: `build\jmeter\<api name>\<api name>.jmx`

- **xins run-jmeter**

  Description: Runs the JMeter tests. You will need to the set jmeter.home=<jmeter directory> property. You can set the jmeter.test property for the location of the tests without the extension. The default is the location of the generated tests.

  Required library: Install JMeter [http://jakarta.apache.org/jmeter/].

  Web site: http://jakarta.apache.org/jmeter/

  Result location: `build\jmeter\<api name>\index.html`

- **xins maven**

  Description: Generates the Maven [http://maven.apache.org/] `pom.xml` file for the specified API.

  Required library: None.

  Web site: http://maven.apache.org/

  Result location: `apis\<api name>\pom.xml`

- **xins smd**

  Description: Generates the SMD [http://maven.apache.org/] (Simple Method Description) of the API. You can set the destination of the API with the smd.endpoint property. The default is the first address in the `environment.xml` file with `?_convention=_xins-jsonrpc`.

  Required library: None.

  Result location: `build\smd\<api name>.smd`

- **xins xsd-to-types**

  Description: Generates .typ files with the types defined in the given XML Schema files for the specified API.

  Parameters: xsd.dir - Directory containing the XML Schema files (*.xsd)

  Required library: None.

  Result location: `apis\<api name>\*.typ`

- **xins wsdl-to-api**

  Description: Generates an API specification based on a WSDL [http://www.w3.org/TR/wsdl] file or URL.

  Parameters: wsd.location - The location of the WSDL (local file or URL)

Required library: None.

Result location: `apis\<api name>\api.xml` `apis\<api name>\*.fnc` `apis\<api name>\*.typ` `apis\<api name>\*.rcd`

- **xins webstart**

  Description: Generates a Java WebStart file (also called JNLP file) to run the API.

  Required library: None.

  Result location: `build\webstart\<api name>.jnlp`

  Note you will need to sign the WAR file if you want to make the API available online.

- **xins appengine**

  Description: Generates a Google App Engine of the Web Service.

  Required library: Google App Engine SDK.

  Result location: `build\webapps\<api name>`

Performances

XINS performances are regularly measured. An old article about XINS performance [http://xins.sourceforge.net/articles/performance.html] shows that XINS could be even faster than the competition.

The number of logs written per call is one of the important factor that can slow down or increase the performance of an API. You are advice to only log necessary messages.

Compressed data

Since XINS 3.1, GZip compressed data are also accepted. If not already done, you are adviced to configure the HTTP Server or the Servlet container to return compressed data.

If the Servlet container runs in a HTTP Server, it's better to configure the compression on the HTTP Server. For example for Apache server, use mod_deflate [http://httpd.apache.org/docs/2.2/mod/mod_deflate.html].

For Tomcat add to the `server.xml` `<Connector>` element the following attributes: `compression="on" compressionMinSize="2048" noCompressionUserAgents="gozilla, traviata" compressableMimeType="text/html,text/xml,application/xhtml+xml,text/javascript,application/json"`.

For Jetty, you need to add a Servlet Filter [http://blog.max.berger.name/2010/01/jetty-7-gzip-filter.html]. This is done in `impl.xml` with the following code:

```
<web-app element="filter">
  <web-app element="filter-name">GzipFilter</web-app>
  <web-app element="filter-class">org.eclipse.jetty.servlets.GzipFilter</web-app>
  <!-- <web-app element="filter-class">org.mortbay.servlet.GzipFilte</web-app> for
  <web-app element="init-param">
```

```
    <web-app element="param-name">mimeTypes</web-app>
    <web-app element="param-value">text/html,text/xml,application/xhtml+xml,applic
  </web-app>
</web-app>
<web-app element="filter-ampping">
  <web-app element="filter-name">GzipFilter</web-app>
  <web-app element="url-pattern">/*</web-app>
</web-app>
```

For other servlet containers, you can add in a the Jetty filter by adding jetty library as dependencies and the code above or use the pjl-comp-filter [http://sourceforge.net/projects/pjl-comp-filter/] project.

Cached data

There are a lot for cases in web services where getting data is much often called than creating or updating data. For example, getting the data of the top 100 products of the catalog of getting the data of the currently logged users.

XINS 3.1 adds the possibility to specify how long the data for a function could be cached on the client side. This is done by adding a cache attribute to the function element in the `fnc` file. The value being the number of seconds that the result could be cached.

```
<?xml version="1.0" encoding="US-ASCII"?>
<!DOCTYPE function PUBLIC "-//XINS//DTD Function 3.1//EN" "http://xins.sourceforge
<function name="FastData" cache="60">
   ...
</function>
```

If the client is a browser the data will be cached automatically in the browser cache.

To use the cache system for the XINS client API of with the XINSServiceCaller, you will need to set the CachingHttpClient [http://hc.apache.org/httpcomponents-client-ga/httpclient-cache/index.html] in XINSCallConfig:

```
XINSCallConfig config = new XINSCallConfig();
config.setHttpClient(new CachingHttpClient(config.getHttpClient()));
capi.setXINSCallConfig(config);
// or in the XINSCallRequest: request.setXINSCallConfig(config);
```

Note that HttpClient cache is not included in XINS, so to use it you need to download it and add it to the dependencies.

Note

HttpClient cache has support for advanced cache such as ehcache and memcache. You can then for example use a shared or distributed cache. If the same software is updating the data, you can remove the cache entry when after the update function is called.

Not modified data

There are also cases where data should be accurate but is not often updated. For example the price or availability of a product. In this case the server is returning all the time the same data, using unneeded resources such as CPU and bandwidth.

XINS 3.1 add the notion of returning not modified to the client. In this case the HTTP status code 304 is returned and no data is including in the body of the message.

To return not modified, just add `return new NotModifiedResult();` in the implementation class when you know that the result is identical to the last request of the same client.

On the client side, a `isNotModified()` method has been added to `XINSCallResult`.

For the CAPI a `NotModifiedException` (which extends `XINSCallException`) will be thrown.

Content of the XINS package

This chapter gives a description of the files and directories included in XINS package, so that you can find the wanted information quicker.

Documentation

In the top directory of XINS, the following documents are available:

- `README.html` contains a quick description of XINS, links to the documentation and quick start to run a small project.
- `CHANGES` contains the change logs between each releases.
- `NOTES` contains the release notes with the known bugs and the OS and Java version with which XINS has been tested.

The docs directory contains the following documents:

- The user guide (it's this document)
- The Javadoc of the XINS.
- The XINS primer. The primer is a description step by step on how to create your first API.
- HTML Logdoc. This document contains a description of the message logged by XINS.
- The XINS protocol. This document explains the communication protocol for the REST calling convention.

On the website http://www.xins.org/documentation.html, the following extra documentation is available

- A document to help you migrate API [http://xins.sourceforge.net/articles/migration2-3to3-0.html] written for XINS 2.3 or lower to XINS 3.0.
- Articles over Asynchronious calls [http://xins.sourceforge.net/articles/asynchronous.html], Result caching [http://xins.sourceforge.net/articles/resultcaching.html], API implementation with scripting languages [http://xins.sourceforge.net/articles/scripting.html] and XINS performance [http://xins.sourceforge.net/articles/performance.html].
- Presentations of XINS for managements [http://xins.sourceforge.net/presentations/xins_intro.pdf] and developers [http://xins.sourceforge.net/presentations/frameworks_and_xins.pdf].

Examples

Three API examples are distributed with XINS. The API's are located in the directory `demo\xins-project`.

Here is a description of the API's:

- `myproject`: This example is a very basic example, much like a "Hello World" example.

- `allinone`: This example uses most of the features including in XINS. The API contains 1 new feature per function.
- `filteredproject`: This example is an API that uses the generated CAPI to call another API (the myproject API).

The examples also include client examples. This shows how to call an API using different programming languages.

The most interesting examples are the Ajax [http://en.wikipedia.org/wiki/Ajax_%28programming%29] examples located in the directory `demo\capis\javascript`. These examples show how to call a XINS API synchroniously or asynchroniously with Javascript and insert the result in the current HTML page. XINS works particularly well with Ajax thanks to the REST calling convention and the XSLT calling convention.

Here is a description of the Ajax examples:

- `callMyFunction.html` calls the XINS API using the REST calling convention and insert the result of the call in the web page.
- `callMetaFunction.html` calls the XINS API using the XSLT calling convention and insert the HTML returned by the call in the web page.
- `callMyFunction2.html` calls the XINS API using the REST calling convention and tranform the returned result using XSLT on the client side. It then inserts the HTML in the web page.

Note

Because of security issues, when these examples are executed locally using Netscape as browser, `callMyFunction.html` will ask for authorisation, `callMetaFunction.html` will fail and `callMyFunction2.html` will fail.

XINS also contain examples on how to call an API using PHP version 4 and 5, Perl and Java. If you want to call a XINS API using another language, just call the API using the URL and parse the returned XML. You can also use the SOAP calling convention or the XML-RPC calling convention.

Program

The package not only contains the documentation and the examples, it contains also the program :-).

The program is located in different directories:

- `bin` contains the shell scripts. This directory should be in you `PATH` environment variable.
- `build` contains the XINS library.
- `lib` contains the third party libaries used by XINS with their license.
- `src` contains XINS source code as well as the XSLT, XML and css files used for the code generation.

A Ant `build.xml` script is provided in the root directory in the case you want to recompile XINS.

Integration with IDE

This chapter will help you to configure a development environment so that you can create, develop, test and debug a project faster.

If you have found some more settings that helped you to develop the XINS API faster using Eclipse, send the procedure to anthony.goubard@japplis.com [mailto:anthony.goubard@japplis.com].

Integration with Eclipse.

The Eclipse version used for this manual is 3.2 and can be found at http://www.eclipse.org/platform.

Note

As described in the install section, if you're using the Ant distribution included with Eclipse, you may need to copy xercesImpl-2.6.2.jar [http://mirrors.ibiblio.org/pub/mirrors/maven2/xerces/xercesImpl/2.6.2/xercesImpl-2.6.2.jar] to the `plugins\org.apache.ant_1.6.5\lib` directory to avoid putDocumentInCache error messages. If you still have the error, install Ant and go to Window -> Preferences -> Ant -> Runtime -> Ant Home... and choose the Ant directory.

As Eclipse locks the opened files, you choose close them before calling the **create-xxx** targets. For example `xins-projects.xml` should be closed before calling **create-api**.

xins eclipse

- If the xins-project.xml is not already created, create it in a new directory with the content specified in the section called “xins-project.xml”.
- In Run -> External Tools -> External Tools, create a new program with the name `xins`, the location set to the `xins\bin\xins.bat` file and the working directory to the directory containing `xins-projects.xml`. Then you need to add in the arguments the `${string_prompt}` variable and add in environment the variable `XINS_HOME` with the value of the XINS directory.
- If the API is not created, click on the run button and enter **`create-api`**. This target will ask a series of question in order to create the API including the first function of the API.
- Click on the run button and enter **`eclipse`**. Enter the name of the API.
- The command will create a `xins-eclipse.userlibraries` in the `xins\src\tools\eclipse` directory. Then go to Window -> Preferences -> Java -> Build Path -> User Libraries -> Import... -> Browse and select the `xins-eclipse.userlibraries` file. This step only needs to be done once.
- After the **eclipse** command, depending on the directory location of the api:
  - If your project is outside your workspace, choose File -> New -> Project... -> Java -> Java Project -> fill the project name -> Create project from existing source and choose the apis\<api name> directory -> Finish.
  - If your project is in your workspace, choose File -> Import... -> General -> Existing Projects into Workspace -> Next -> choose the apis\<api name> directory -> Finnish.

Run it with Tomcat plug-in.

In order to be able to run and debug your web application, you'll need to install Tomcat (http://jakarta.apache.org/tomcat/) and the Tomcat plug-in for Eclipse (http://www.sysdeo.com/eclipse/tomcatplugin).

Configure the Tomcat plug-in in the Preferences:

- Set the Tomcat home to the location where you have installed Tomcat

- Add, in the JVM settings, the JVM parameter `-Dorg.xins.server.config=c:\java\tomcat\conf\xins.properties` (the `xins.properties` could also be in another directory). For more details about the `xins.properties`, read the section called “The runtime properties”.
- Change the Configuration file to `apis\<api name>\tomcat-server.xml`.
- Start Tomcat.

Run it and debug it with Jetty plug-in.

You can also use the Jetty plug-in for Eclipse. To install it, go to Help -> Software Updates -> Find and Install... -> Search for new features to install -> Add Update Site. Then enter the name JettyLauncher and the URL http://jettylauncher.sourceforge.net/updates/.

- Create an Jetty XML file to link a directory to the `build/webapps/<api>/<api>.war` file.
- Go to Run -> Run...
- Create a new Jetty Web project
- Set a name for the project, set the Jetty home
- Click on Use a Jetty configuration file and set the value to the Jetty XML file you've created
- Go to the argument tab and add the parameter `-Dorg.xins.server.config=<jetty_home>\etc\xins.properties`.
- Now you can run, add breakpoints and debug your web application using the buttons on the toolbar

Other improvements.

With Eclipse, by changing the code and saving it you can run your modification without recompiling if the modification is minimal. If you modified a lot of code, you will need to regenerate the war file.

If you have also generated the specific documentation you may want to use the test forms or the example to test your project. To do so, create a new External tool named specdocs with the main program linked to your HTML Browser and the arguments set to `build\specdocs\<api>\index.html` file.

In order to be able to edit the specifications that are in XML, we also advise that you install a XML plug-in for eclipse such as XMLBuddy (http://www.xmlbuddy.com/) or XML Author (http://www.svcdelivery.com/xmlauthor/). Then in Preferences -> Workbench -> File Associations, add `*.fnc`, `*.typ` and `*.rcd` associated with the XML tool.

If you execute the `clean` target, you may have afterwards a window popping up asking you to choose between `build.xml` and `build.xml (1)`. To remove this window go to External Tools... menu and remove the `build.xml (1)` configuration.

Integration with NetBeans

The version of NetBeans used for this manual is 5.0 and can be downloaded at http://www.netbeans.org/.

Setting up the project.

An Ant script for NetBeans 5.0 or higher is provided with XINS. It requires a correctly set `XINS_HOME` environment variable.

- Add the DTD catalog by clicking on the runtime tab, right click on the DTD and XML Schema Catalogs item and add the OASIS catalog located at `src\dtd\xinsCatalog.xml`.
- Register some extension as XML files in `Tools -> Options -> Advanced Options -> IDE Configuration -> System -> Object Types -> XMLObjects -> Extensions and MIME Types` by adding the items `fnc`, `typ`, `rcd` and `cat`. Check also at the same time the XML Indentation Engine settings. From NetBeans 6.5 `Tools -> Options -> Miscellaneous -> Files`
- Open the project in `demo\xins-project\apis\petstore`.
- If the API already exists, execute the `copy-nb-files` target of the `nbbuild.xml`.
- If you want to create a new API, execute the `create-api` target. This target will ask a series of question in order to create the API including the first function of the API.
- Open the project by selecting the directory of the API: `<project dir>\apis\<api name>`
- You can now open your function implementation file and write it's implementation in the `Result call(Request request)` method.
- Click on the Compile button to compile, on the Run button to run your API, on the Debug button to debug it or on the Apply Code Changes if you have modified the code with debugging.
- To profile the API, install NetBeans profiler (http://profiler.netbeans.org/) and execute the profile target.

In NetBeans, the targets can be executed by right-clicking on the `nbbuild.xml` and `Run target` or by right-clicking on the project icon or by using the toolbar if your project is the main project.

In NetBeans you can set conditional breakpoints or exception breakpoints that stop the debugger whenever an exception is thrown.

Targets to deploy your API in Tomcat are provided. You can then use the Netbeans HTTP Monitor with it.

If NetBeans 6 has problems to recognize the `Request` and `Result` objects, you can add the option *`-J-DCacheClassPath.keepJars=true`* in the Netbeans `etc\netbeans.conf` file.

Integration with other frameworks

XINS is based on several standard such as POJO, Servlet API, SOAP, REST or Ant which eases the integration with other frameworks.

Spring framework

XINS can easily be integrated with the Spring Framework [http://www.springframework.org/].

XINS generates POJO's for requests and result on the client and server side. The classes and the methods are public so that they can be called from another package. The generated POJO's try to follow the Java Beans specification as much as possible.

For example, if you want to use Spring AOP transaction for a specific API, you can define as point cut all methods starting with call in the generated CAPI class.

On the server side, XINS is a Servlet so you can use it with org.springframework.web.servlet.* [http://www.springframework.org/docs/api/org/springframework/web/servlet/package-summary.html].

XINS also includes a Spring framework package. The package is located in `org.xins.common.spring` (javadoc [javadoc/org/xins/common/spring/index.html]). This package is adding a few convinient classes for validation, factory bean and client interceptor.

If you want to load and access an Application Context in the API, you need to add Spring, and a listener to `impl.xml`:

```
<web-app element="context-param">
  <web-app element="param-name">contextConfigLocation</web-app>
  <web-app element="param-value">/WEB-INF/applicationContext*.xml</web-app>
</web-app>
<web-app element="listener">
  <web-app element="listener-class">
    org.springframework.web.context.ContextLoaderListener
  </web-app>
</web-app>
<content dir="api/myproject" includes="applicationContext*.xml" />
<dependency dir="spring" includes="spring.jar" />
```

Then to get access to the bean defined in the application context use annotations such as `@Component` or `@Autowired`.

Dojo toolkit

The Dojo toolkit [http://dojotoolkit.org/] has several ways to communicate with the server.

One of it is JSON-RPC (dojo.rpc.JsonService) which is also supported by XINS on the server side with the `_xins-jsonrpc` calling convention. The easiest way to use this JsonService is in combinaison with a SMD (Simple Method Description). SMD is a file which describes the name of the methods of the API with the expected parameters. There are two ways to get this file from XINS. Either execute the **xins smd** command that will generate the file `build\smd\<api name>.smd` or by calling the meta function `_SMD` to the API you want to invoke. An example is provided in `demo\capis\javascript\callMyFunctionDojo.html`.

You can also use dojo.rpc.YahooService where calls will be received by the `_xins-json` calling convention. This service is also able to read and interpret the SMD file.

Google Web Toolkit (GWT)

GWT [http://code.google.com/webtoolkit/] has mostly two ways to communicate with a server. They first have their own RPC system [http://code.google.com/webtoolkit/documentation/com.google.gwt.doc.DeveloperGuide.RemoteProcedureCalls.html]. GWT also facilitates the call to APIs using the Yahoo JSON protocol which matches the `_xins-json` calling convention.

An example is provided at http://gwt.google.com/samples/JSON/JSON.html.

Note that XINS also supports the callback parameter which is use to indicate to the call which method should be called with the output parameter when the response returns.

AJAX

The others (than Dojo and GWT) AJAX frameworks are also supported.

They have different ways to call XINS. They can use the _xins-json or _xins-jsonrpc calling conventions but also the _xins-std and parse the returned XML themselve or use the _xins-xslt calling convention in combination with an XSLT stylesheet that transform the XML in the HTML to insert dynamiccally in the page.

Several examples of the later are provided in the `demo\capis\javascript` directory.

Mule

Mule [http://mule.codehaus.org/display/MULE/Home] is the most popular open source Entreprise Service Bus (ESB).

As XINS is a Servlet, you can register a handler on the URI (embedded Jetty), or bind through an existing web container (servlet endpoint).

Documentation can be found in the org.mule.providers.http.jetty [http://mule.codehaus.org/docs/apidocs/org/mule/providers/http/jetty/package-summary.html] package or in the org.mule.providers.http.servlet [http://mule.codehaus.org/docs/apidocs/org/mule/providers/http/servlet/package-summary.html] package.

ESB and JavaEE

Most of modern ESB support the possibility to plug JavaEE servers to it and as part of the JavaEE specification there is the possibility to deploy and run Servlet, there shouldn't be any problem to install XINS APIs in an ESB and/or JavaEE server.

The main problem can only be to indicate where find the runtime properties file, as it is normally done using the org.xins.server.config system property. If you cannot set this property, you can also include the runtime property file to use in the WAR file as `WEB-INF/xins.properties`. Use the `<content dir="<path>" includes="xins.properties" web-path="WEB-INF" />` element in `impl.xml` file.

Orchestration is also an important feature in ESB. It can be done using WS-BPEL as XINS APIs can communicate using SOAP with the `_xins-soap` calling convention. The API can also give the WSDL file to indicate the SOAP message format with the `_WSDL` meta function.

Ant

XINS uses Ant to generate the different files. It is also possible to run the tests or the API with Ant targets. The easiest way to do it is to use the xins task as described in the the section called “xins task”.

Another possibility is to look at the nbbuild.xml file included with the petstore demo and can be used for other existing APIs or new APIs. Note that some of the targets are NetBeans specific but most of them not.

Maven

XINS also has support for Maven 2.

Each stable releases are added in the Maven 2 repository (http://www.ibiblio.org/maven/org.xins/). Note that not only the jar files are included, the source code and the Javadoc are also included.

If you want to build your API using API using Maven, you can execute **xins maven** command which will create the `<api name>.pom` in the `apis\<api name>` directory. Note that the external dependencies (other than XINS) are not included in the generated POM and you should add them manually.

Script languages (Groovy, Ruby, PHP, Perl...)

On the client side it is possible to either call the standard calling convention or use a standard protocol such as SOAP or XML-RPC.

As the standard calling convention is simple, it should be easy to write a script that creates a URL, send it and parses the XML response returned.

Some of the script also have libraries to call a server by using SOAP or XML-RPC. XML-RPC clients are listed at http://www.xmlrpc.com/directory/1568/implementations.

Examples with different script are provided in the `demo\capis` directory.

You can also implement an API using a script language. In this case you need to use a library to pass the input parameters to the script and get the output parameters from the script. An example is provided in XINS using Groovy as script language in the `demo\xins-project\apis\toolbox` directory.

Google App Engine (work in progress)

Since XINS 2.3, a new command **xins appengine** allow to package the API and deploy it on the Google App Engine [http://appengine.google.com].

You will need to specify the location of the appengine SDK with the `appengine.sdk.dir` property.

The Google App Engine requires a `appengine-web.xml` in the `WEB-INF` directory. If you don't have one XINS will create one for you (in `build/webapps/<api name>/war/WEB-INF`). You may need to change the application name and version (. and _ are not accepted in version). Another possibility is to set `appengine.api.name` and `appengine.api.version` build properties.

Also as there is no runtime properties, the file specified in the `org.xins.server.config` build property will be used.