



July 13

Sage **ERP**X3

---

## TEC302-SAFE X3 Web Services

---

From SAFE X3 Version 6.2



# Objectives

- To be more comfortable with the Web Services concept.
- To understand, configure and use Web services.
- To be able to implement Web Services in X3.

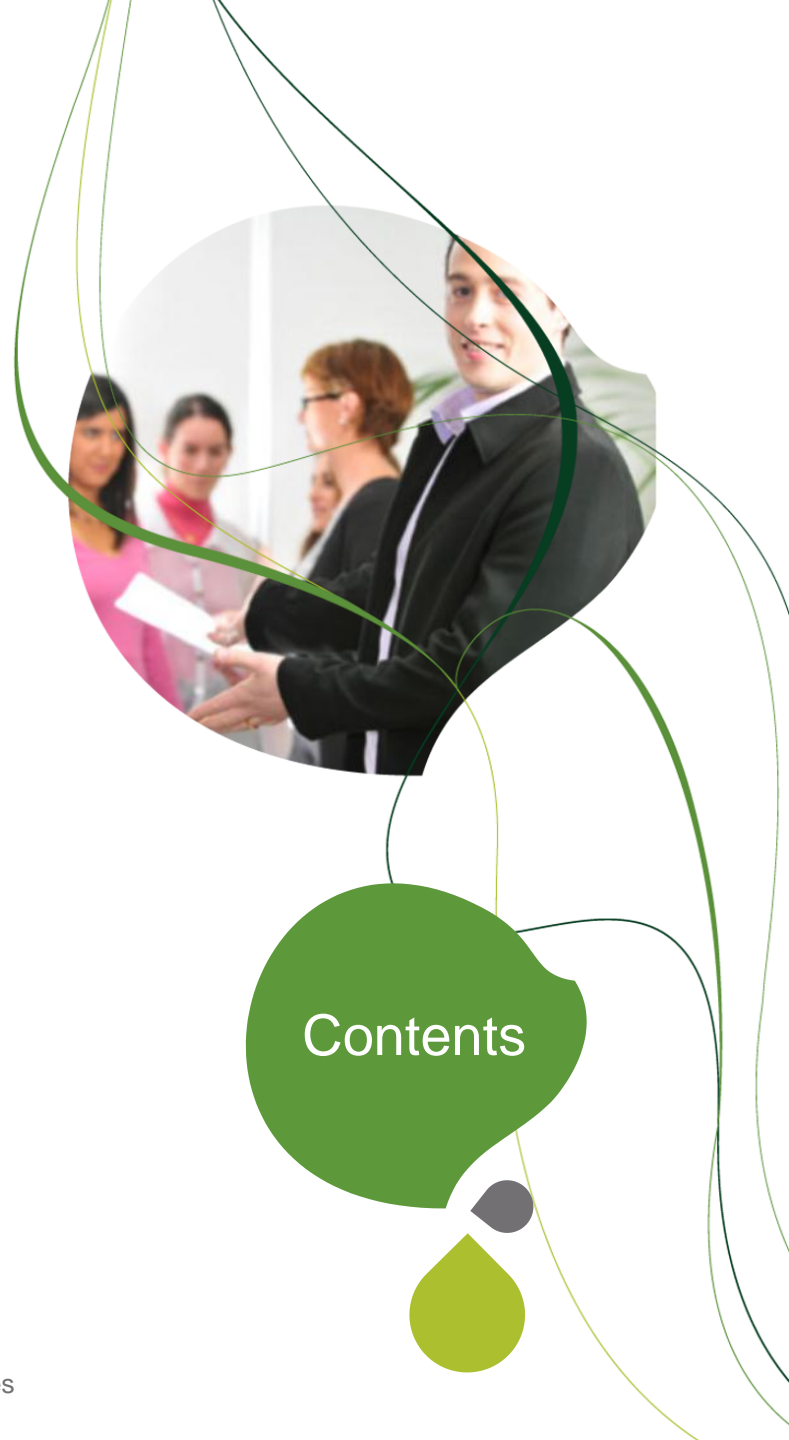


# Course Contents

1. Introduction: The Web Service Protocol
  - 1.1 – What Is a Web Service
  - 1.2 – The X3 Web Services
  - 1.3 – X3 Web Service Methods
2. The X3 Web Services Architecture
  - 2.1 – The SAFE X3 Solution
  - 2.2 – The X3 Web Server
  - 2.3 – Web Service Request Processing
  - 2.4 – The Connexion Pool
  - 2.5 – Publishing Web Services In X3
3. Setting Up X3 Web Services
  - 3.1 – Installing The Web Server
  - 3.2 – Setting Up The Web Services Pool
  - 3.3 – The Web Server Technical Page
4. Publishing Subprograms
  - 4.1 – The Subprogram Dictionary
  - 4.2 – Publishing A Subprogram
  - 4.3 – The XML Description File
  - 4.4 – The Wrapper Program
  - 4.5 – Connection Types
5. Publishing Objects
  - 5.1 – Publishing An Object
  - 5.2 – The XML Description File
  - 5.3 – The Wrapper Program
6. Testing Web Services
  - 6.1 – The Web Service Tester (Web Server Technical Page)
  - 6.2 – XML Data Streams
  - 6.3 – A Java Tester For X3 Web Services
7. Integrating Web Services Into Your Application
  - 7.1 – The WSDL Description
  - 7.2 – The Stub Classes
  - 7.3 – .NET Development
  - 7.4 – Java Development
  - 7.5 – PHP Development
  - 7.6 – Debugging Web Services
8. Best Practices & Performances
  - 8.1 – Publication
  - 8.2 – Web Services Pool
  - 8.3 – Miscellaneous



# 1 - Introduction



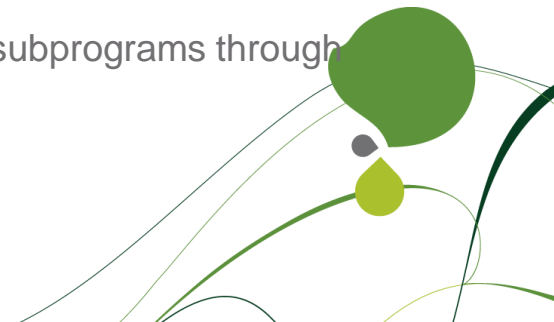
# 1.1 – What is A Web Service?

- A **Web Service** is a functionality proposed by an application as a *service* to other applications, through the internet, independently from underlying platforms or development languages used by the client applications.
- The set of network protocols Web Services are based on is called **SOAP** (**S**imple **O**bject **A**ccess **P**rotocol). It defines XML message transportation over a network, usually through http. Web Services are hence mainly based on http and XML.
- A Web Service has a set of **methods** that may be invoked or called by the client application. Each method represents a functionality of the Web Service. It is called by sending the parameters to it in XML format, and returns a set of results, also in XML format.
- Web Services are the ideal tool for integration of business applications and **B2B/B2C** commerce.



## 1.2 – The X3 Web Services

- With SAFE X3, you may publish **Objects**, **Object Lists** and **Subprograms** through Web Services that will make them accessible to external applications through the internet:
  - Public commercial web site that creates orders in X3
  - Enterprise application other than X3
  - Commercial sales application on a PDA
  - ...
- The common links between those applications are:
  - The need to access X3 in a fully synchronous way
  - Maximum security (no data/text files transiting on the servers)
- SAFE X3 Web Services fully comply with the public Web Service protocol. They are no longer proprietary as were the ADAPI tools in the previous version.
- X3 publishes a **single** web service that gives access to all objects and subprograms through the 3 different methods.



# 1.3 – X3 Web Service Methods

**Objects**, **Object Lists** and **Subprograms** are the three method types

X3 publishes a unique Web Service that contains three types of methods:

- ◆ **Objects:** A set of methods to create, modify or delete an X3 object record and also to execute buttons or menus linked to those objects, with certain conditions:  
Sales Orders  
Customers  
Products  
...
- ◆ **Object Lists:** A method that queries the main object left list, i.e. provides a list of the records with the same format as the X3 left browser.
- ◆ **Subprograms:** A method for calling an X3 subprogram (X3 code) declared in the Subprogram Dictionary, for example:  
Get a price list for a product/customer combination  
Get a product picture  
Get the available stock for a product  
Allocate customer reservations  
(Any X3 subprogram that does not require user interaction or an X3 window)



# 1.3 – X3 Web Service Methods

## Object methods

### *read(context, object, key)*

*context*: Connection parameters and other options (trace, debug etc.)  
*object*: The object Web Service name  
*key*: Key(s) element(s) for the record to be read

- This reads an object record and returns all the field values. It is equivalent to accessing the record in X3, but the record is not locked.

### *save(context, object, data)*

*data*: All field data necessary for creating the object (XML format).

- This creates an object record based on the data transmitted to the method, at least all the values for mandatory screen fields. It is equivalent to populating the screens and pressing the *Create* button in X3. All application controls are performed.
- X3 responds by returning all the screen data (All fields) and eventual X3 error or warning messages.

### *modify(context, object, key, data)*

*data*: Field values to be modified (XML format)

- This modifies an existing record in the object. The key is specified, as well as field values to be modified.
- This is equivalent to populating the fields and pressing the *Save* button in X3.





# 1.3 – X3 Web Service Methods

## *Object* methods

### `delete(context, object, key)`

*key*: Key(s) element(s) for the record to be deleted

- This deletes an object record after performing necessary data integrity checks. It is equivalent to pressing the *Cancel* button in X3.

### `actionObject(context, object, option, data)`

*option*: The object option code attached to the button or menu item.

*data*: Object field data, same as in `create`

- This triggers the action linked to an object button or menu item, for example the **V** action attached to Sales Orders that will calculate the SO Value without creating the order.
- X3 responds by returning all the screen data (All fields) and eventual X3 error or warning messages.



# 1.3 – X3 Web Service Methods

## *Subprogram* method

*run(context, subprogram, data)*

*subprogram:* Key(s) element(s) for the record to be deleted

*data:* Subprogram parameters (input) in XML format

- ◆ This runs an X3 subprogram identified by its Web Service name, using the parameters in the input XML data.
- ◆ Return values are in XML format.



- ◆ To be published as a Web Service, a subprogram must **not require any user interaction (X3 window)**, the only data it will run with being in the input parameters.



## 1.3 – X3 Web Service Methods

### ***Object List*** method

***query(context, object, key, nbLines)***

***object:*** The object Web Service name

***key:*** Key(s) element(s) of the record

***nbLines:*** The maximum number of lines requested from the left list

- This queries the object's main left browser and returns the column values for a maximum of *nbLines* records, in XML format.



# 1.3 – X3 Web Service Methods

## *Web Service Description* method

### `getDescription(context, webService)`

*webService*: A Web Service public name (Object or Subprogram)

- This method allows you to:

Verify that an object, subprogram or object list has been published in X3 as a Web Service.

Load the latest version of that Web Service in the X3 web server cache.

Obtain a description of the data required or returned by the Web Service, i.e. the following information:

- For an **object list**, the names and types of the columns defined in the object
- For an **object**, the names and types of the screen fields for that object
- For a **subprogram**, the name and types of the subprogram arguments.





## 2 – X3 Web Service Architecture

Contents

## 2.1 – The Sage X3 Solution

Web Services are linked to an X3 solution

A solution is a unique, identified Sage X3 installation that contains

- A technology platform (Runtime, Database Server, Web Server etc.)
- A number of folders arranged in a hierarchy (Reference folder → Child folders)
- Database elements that belong to each folder
- Directories on the application server

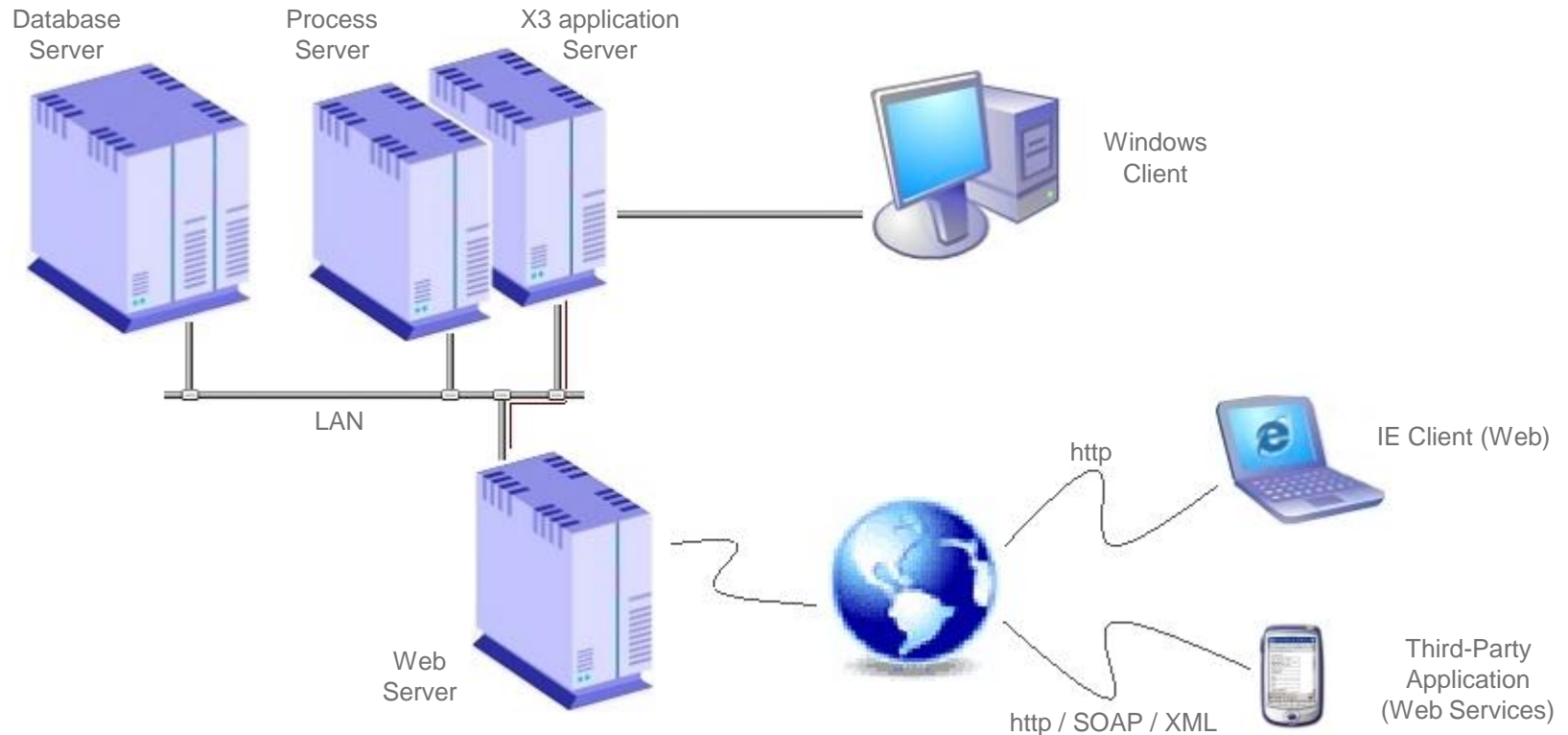
An X3 installation (Solution) is identified by a name and a code, defined in the Console and stored in the *solution.xml* file in the parent directory of the reference folder directory (Typically the *.../Folders* directory.)

Different X3 solutions, on different machines or on the same machine, may have folders with the same name published through the same Web Servers. The solution name acts as the differentiator in that case.



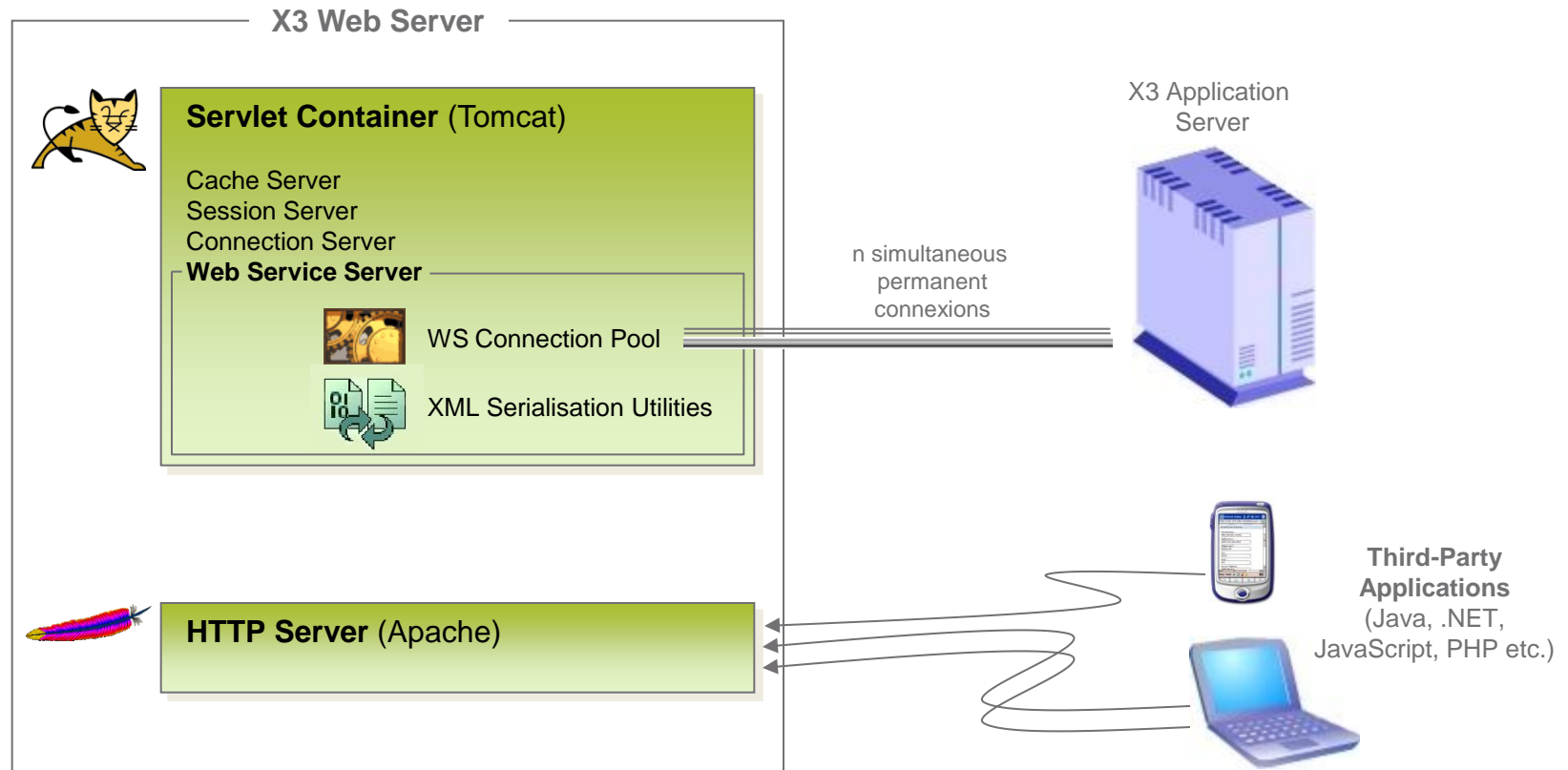
## 2.1 – The Sage X3 Solution

### X3 solution architecture



## 2.2 – The X3 Web Server

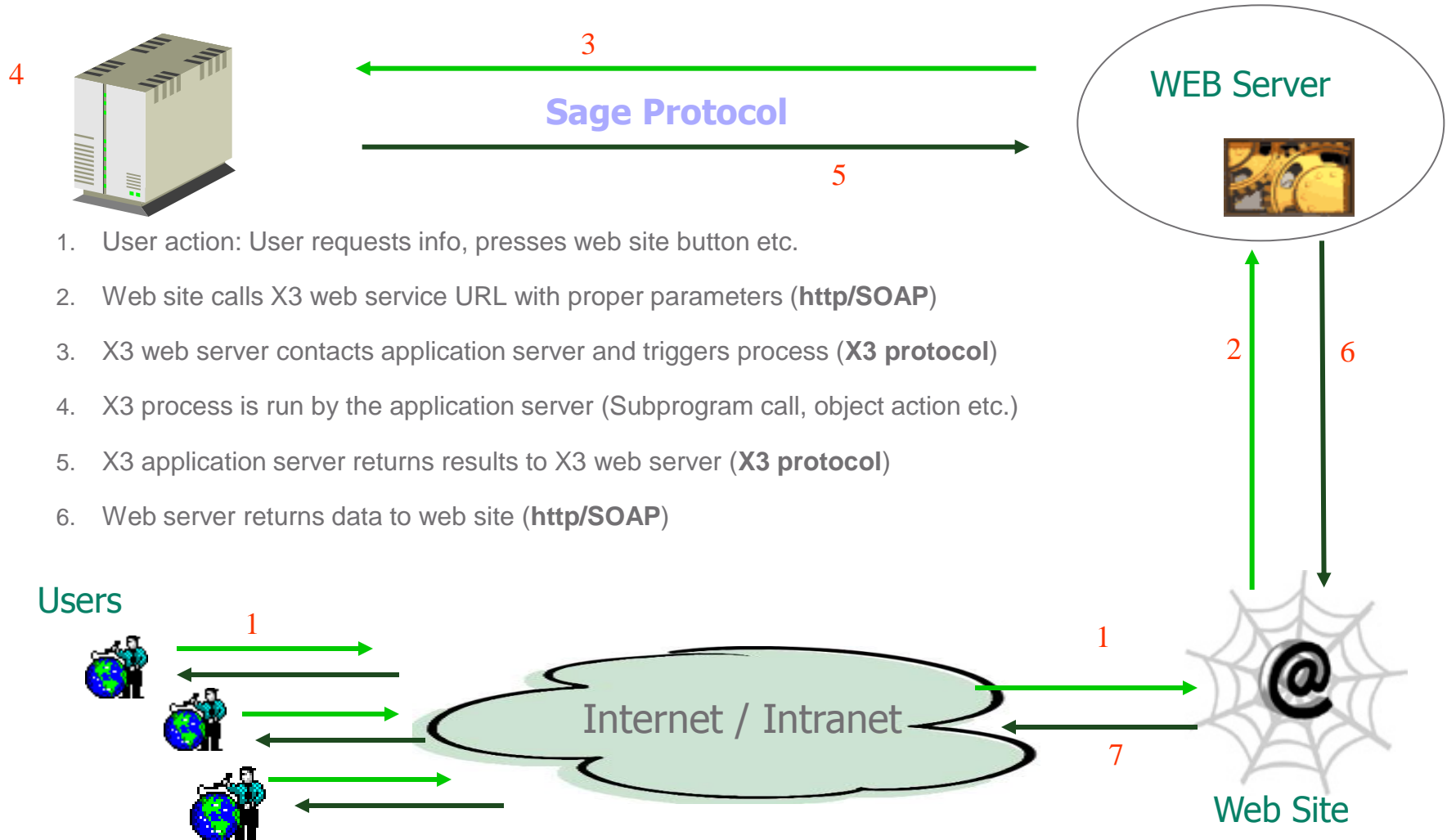
The X3 web server publishes Web Services and processes requests





## 2.3 – Web Service Request Processing

### X3 Application Server



## 2.4 – The Connection Pool

### What is the connection pool?

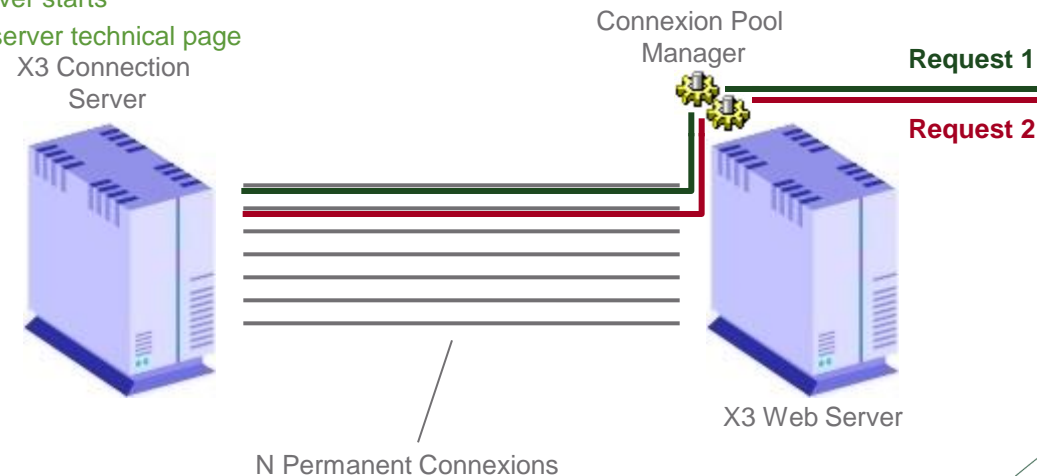
A set of **permanent** X3 connexions with the application server, used when needed by the web server.

The tools and mechanisms to manage waiting queues for Web Service requests and the way available connections are used to process the requests.

### The connection pool is started with a predefined number of connections

Automatically when the X3 web server starts

Manually from the console or web server technical page



## 2.4 – The Connection Pool

### How does the connection pool work?

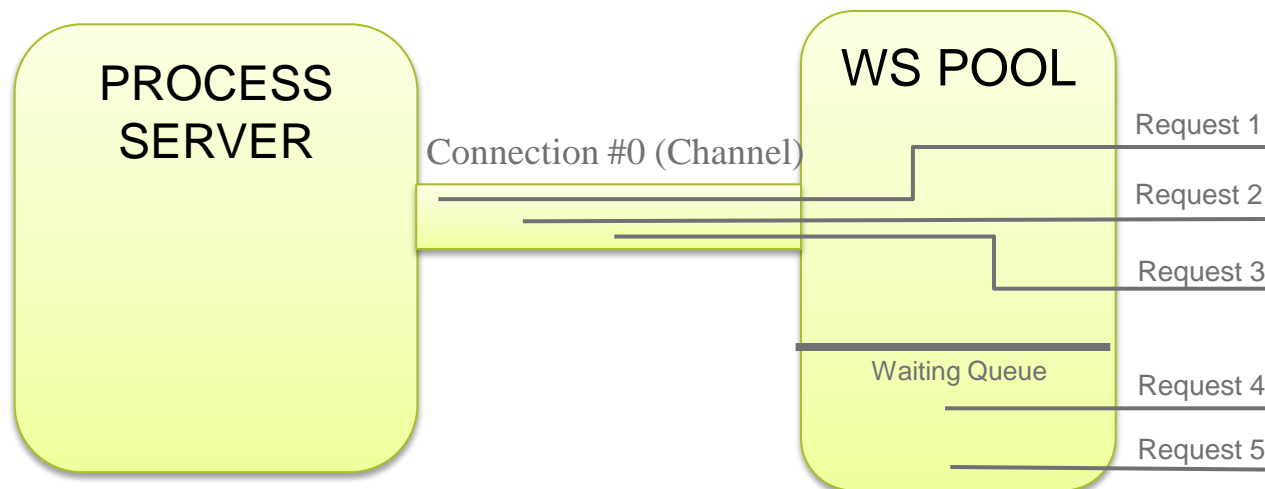
Requests may arrive simultaneously on the web site.

Requests are piled in a waiting queue until a connection is available.

When a connection is available, the web server uses that connection to process the top request in the waiting queue (in FIFO order)

- The pool distribution algorithm optimises the use of available connexions depending on the user, language, folder etc. specified.

The total number of connections is specified in the Web Server parameters (Console) and is limited by the maximum number of Web Service connections allowed by the X3 licence.

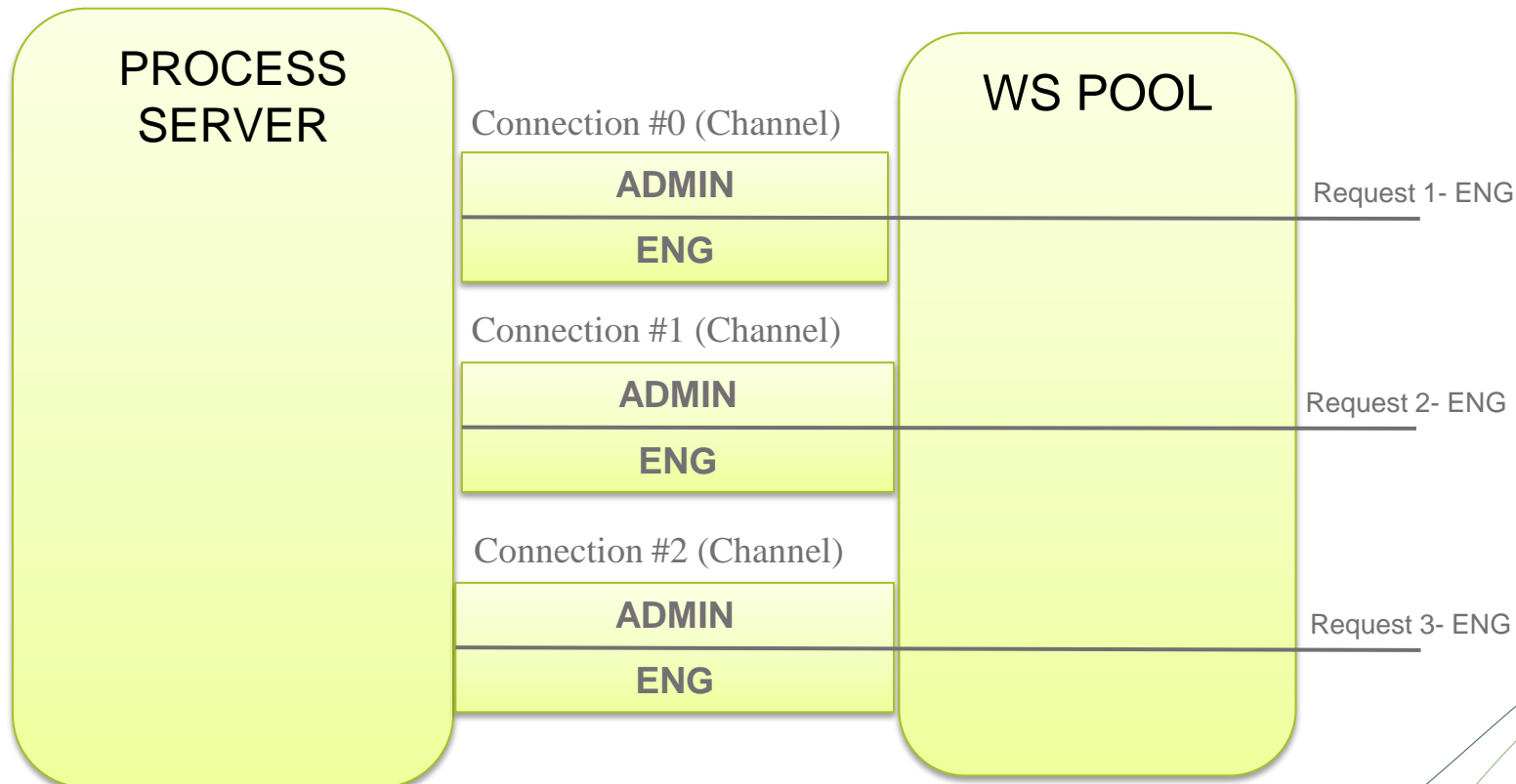


## 2.4 – The Connection Pool

How does the connection pool work?

E.G: requests with same user and 3 languages.

- The channels will adapt to the requests in order to remain open without switching context.

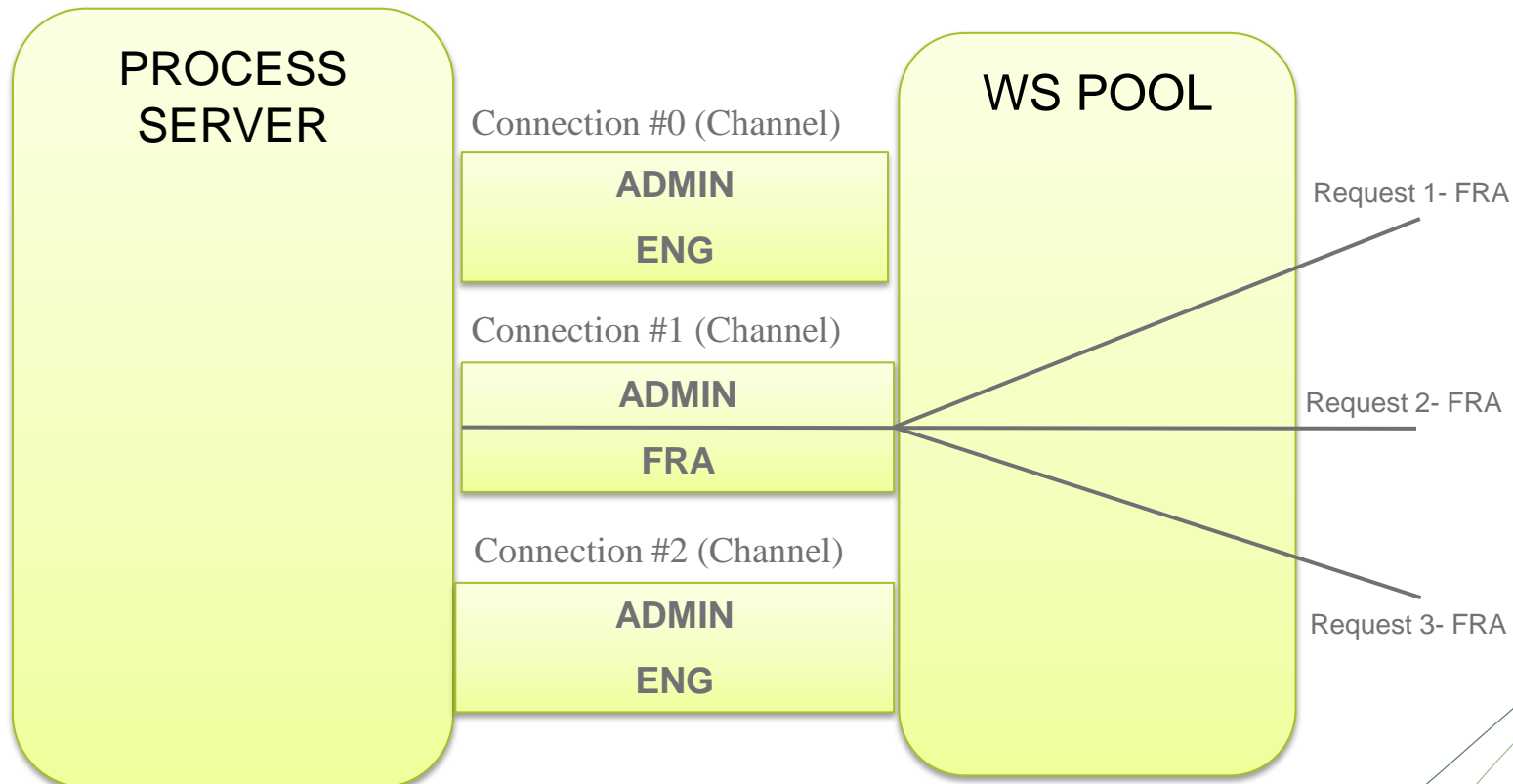


## 2.4 – The Connection Pool

How does the connection pool work?

E.G: requests with same user and 3 languages.

- The channels will adapt to the requests in order to remain open without switching context.

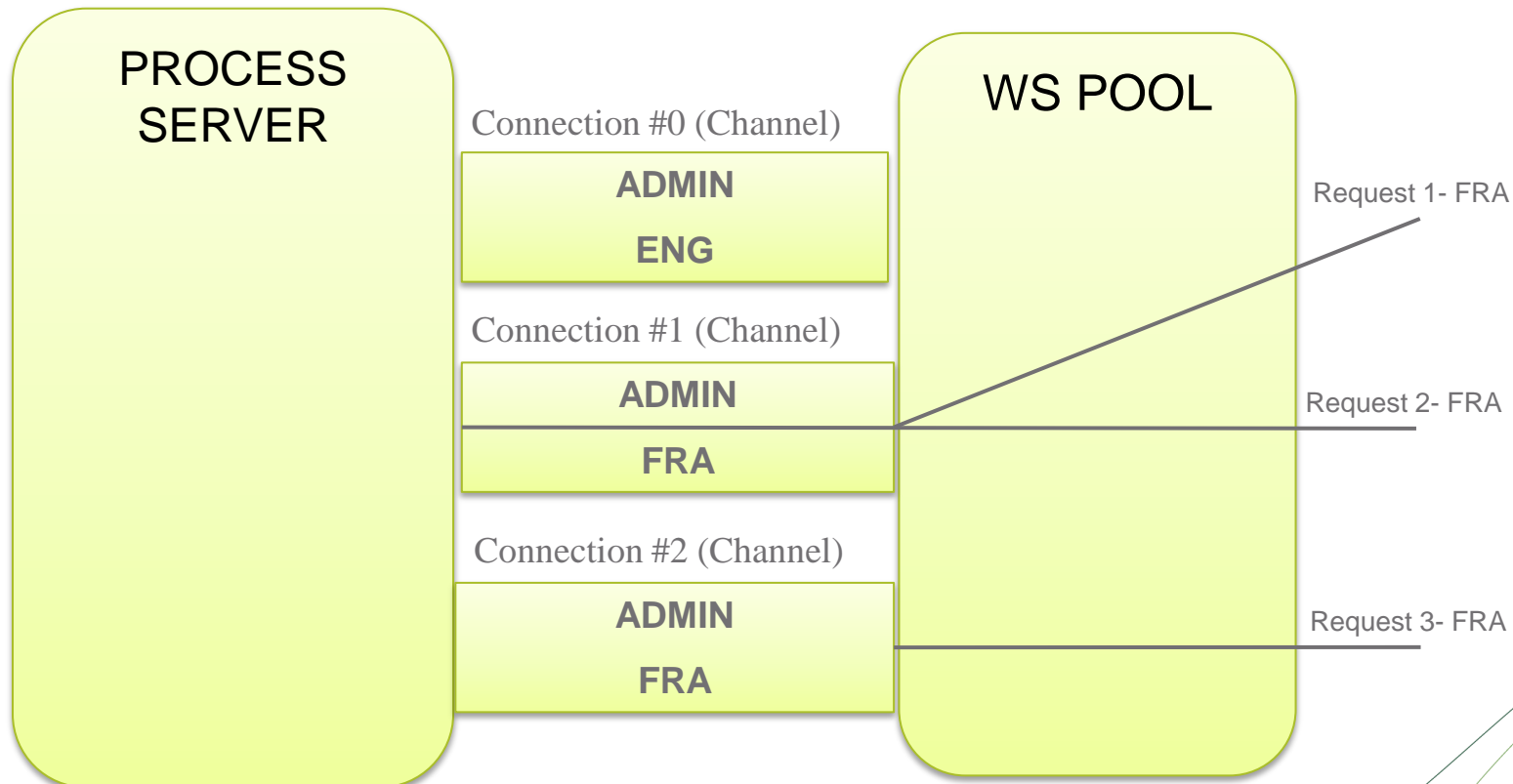


## 2.4 – The Connection Pool

How does the connection pool work?

E.G: requests with same user and 3 languages.

- The channels will adapt to the requests in order to remain open without switching context.

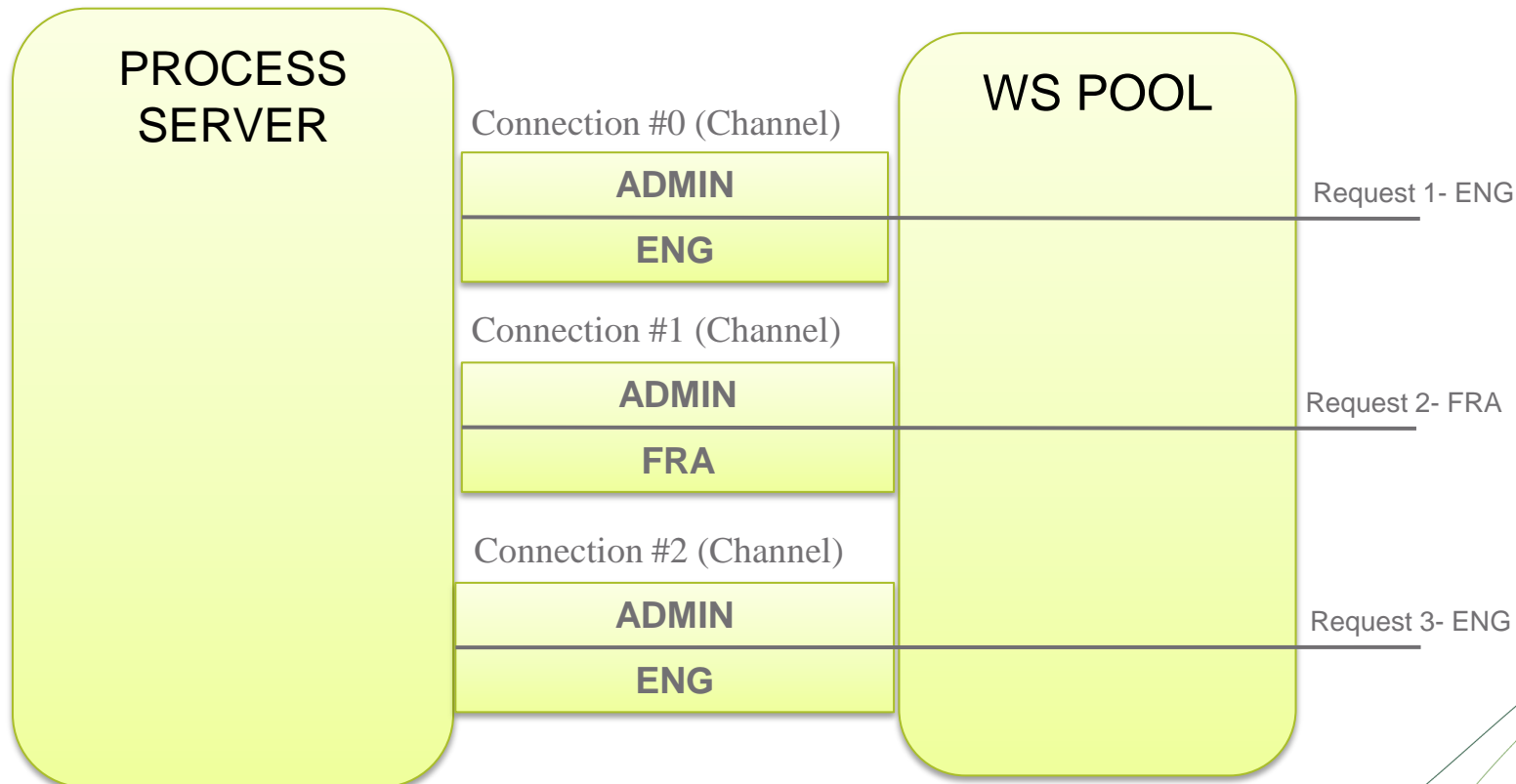


## 2.4 – The Connection Pool

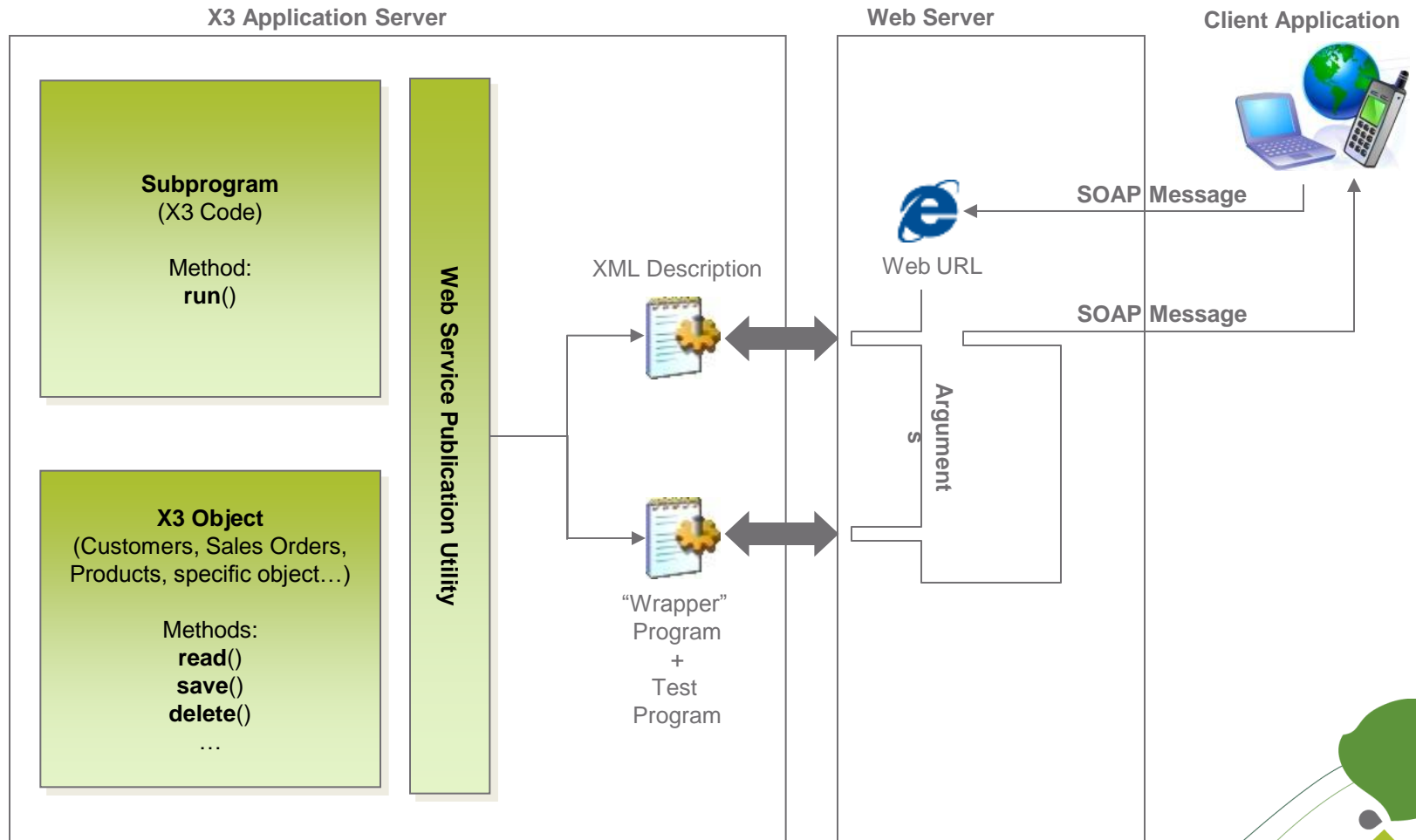
How does the connection pool work?

E.G: requests with same user and 3 languages.

- The channels will adapt to the requests in order to remain open without switching context.



## 2.5 – Publishing Web Services In X3





## 2.5 – Publishing Web Services In X3

The publication utility (*Development / Processes / Processes/ Web Services*)

It is a very simple function that transforms a subprogram or object into a web service with a public name, and generates its XML description, stored in the *X3\_PUB/<FolderName>/GEN/ALL/WEBS* directory on the application server.

It is also possible to list published web services (for the folder), to unpublish existing web services, or to generate all the web services defined for the folder (Global Publication).

An object or subprogram may not be published twice with different names. To modify the public name of an existing object or subprogram, it must be unpublished and re-published with the new name.

Generated web services are time-stamped, and the web server will always check if the version in its cache is the latest version. If not, it will reload the latest version from the application server.



## 2.5 – Publishing Web Services In X3

### The wrapper program

The publication utility also generates a wrapper program in X3 code, which is the interface between X3 and the web server. The wrapper program is called:

- **V5:**  
**WJ[OBJ][TRS]** for objects, where [OBJ] is the object code and [TRS] the entry transaction code selected (If any).  
**WR[WS\_NAME]** for subprograms, where [WS\_NAME] is the subprogram public name (Web Service name).
- **V6:**  
**WJ[WS\_NAME]** for objects, where [WS\_NAME] is the object public name (Web Services).  
**WJ[WS\_NAME]** for subprograms, where [WS\_NAME] is the subprogram public name (Web Service name).

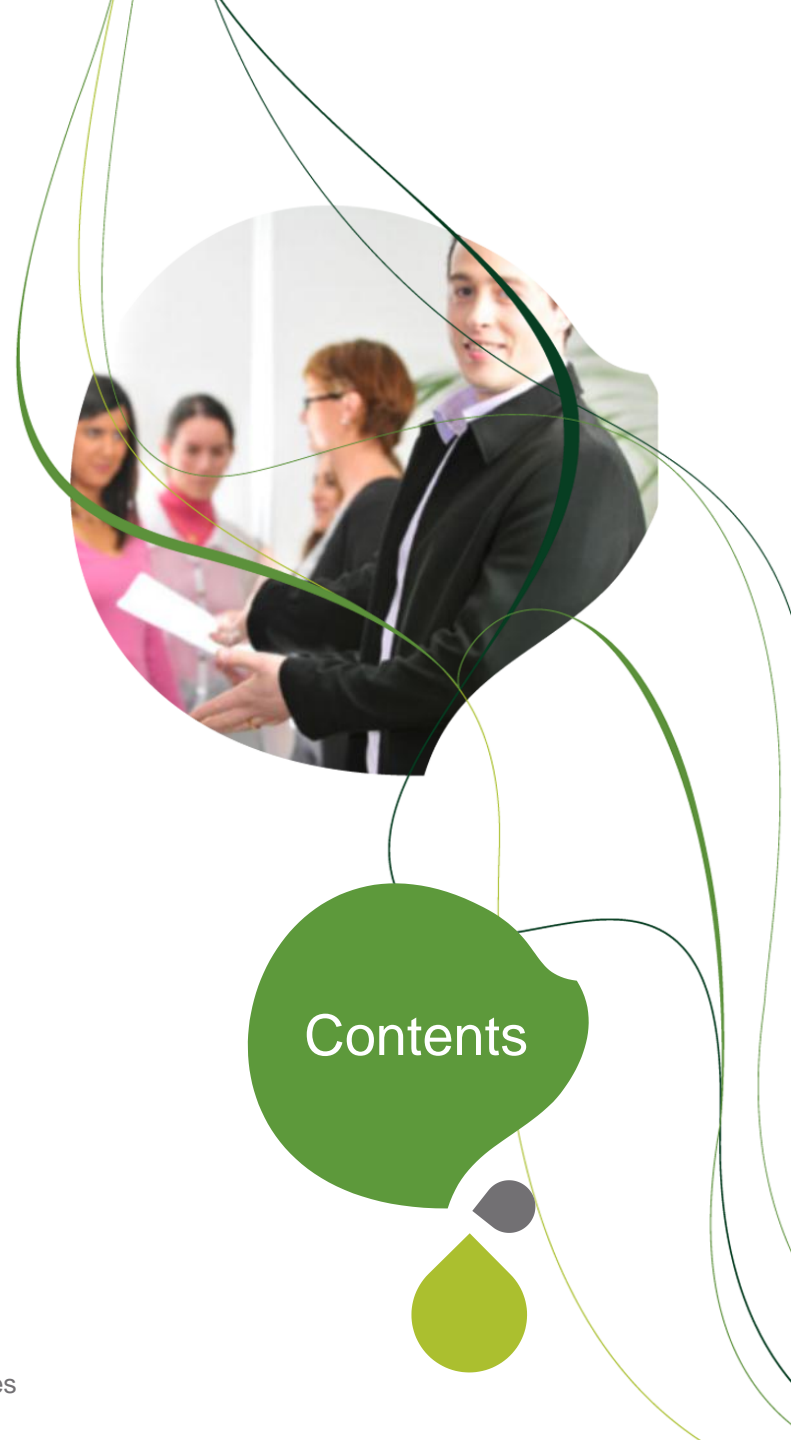
The wrapper program manages:

- Header arguments (Technical data, timestamps etc.)
- Messages
- Data streams, both incoming and outgoing.

The wrapper program contains an automatic test subprogram that allows the developer to “run” the web service in X3 before using it in a third-party application.



## 3 – Setting Up Web Services



## 3.1 – Installing The Web Server

- Web Services are part of the X3 web server

You must install the web server to be able to use X3 Web Services.

Once the web server is installed, you must publish the folder you will access, using the normal procedure (Console – c.f. Installation course).

- Web Services are subject to licence restrictions

Your licence must allow Web Service connections.

Web Service information may be viewed in the *Licence Enquiry* function.

*Development > Utilities > Verifications > Visualize licence*



## 3.2 – Setting Up The Web Services Pool

The Web Service connection pool is set up using the console (*Web Servers* panel)

A pool is attached to a single X3 folder, via a connection server (runtime).

A pool is identified by a unique name in the X3 solution, and by the following parameters:

- Default startup language (X3)
- Default startup X3 user code and password
- System account / password
- MaxSize (maximum number of connections)
- Initial size (number of connections at startup)

**REMARK : The maxSize and InitSize MUST NOT be greater than the number of WS licenses. If so, the pool will be automatically stopped, and will not be started unless you put proper values.**

When configured, the pool may be started manually from the Console.



## 3.3 – The Web Server Technical Page

The web server technical page shows detailed information on the system's Web Service pools

It is accessible through the Tomcat URL

V1.4.X [http://\[WebServer\]:1898/root](http://[WebServer]:1898/root),

V5.X and V6.X [http://\[WebServer\]:28880/root](http://[WebServer]:28880/root)

**Login: sage**

**Password: sage**

See *Web Services Server* section.

It contains the following elements:

- **Connection Pool administration:** Contains configuration information for all the connection pools managed by this Web Server. May be used to start/stop the pools.
- **Cache administration:** Shows cached Web Service descriptions.
- **Axis server:** Shows the status of the Web Server *Axis* component, as well as published Web Services and their methods.
- **Axis Stubs:** Download section where different stub classes may be found.
- **Trace:** Server activity log and associated configuration utilities.

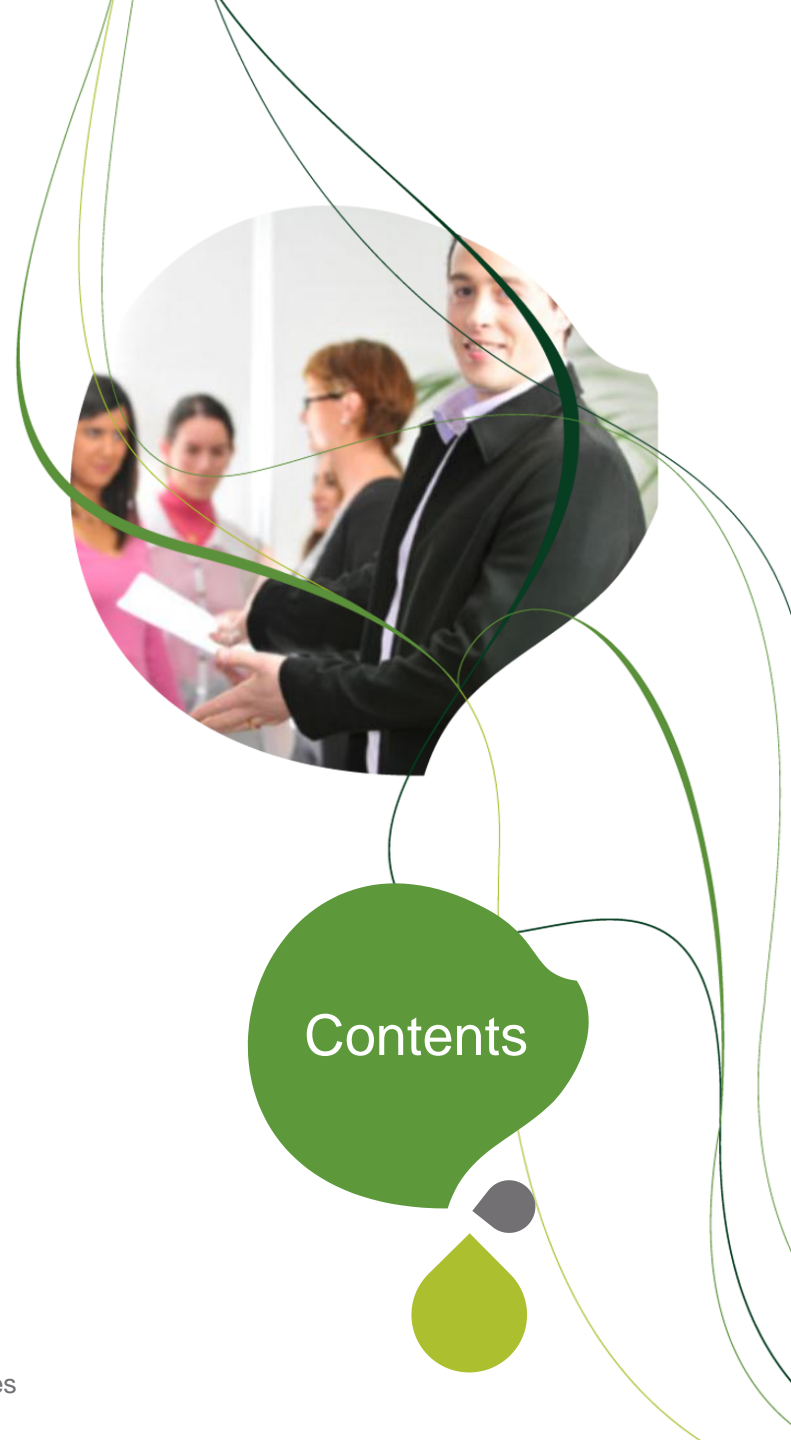
The web server technical page also contains a Web Service tester.



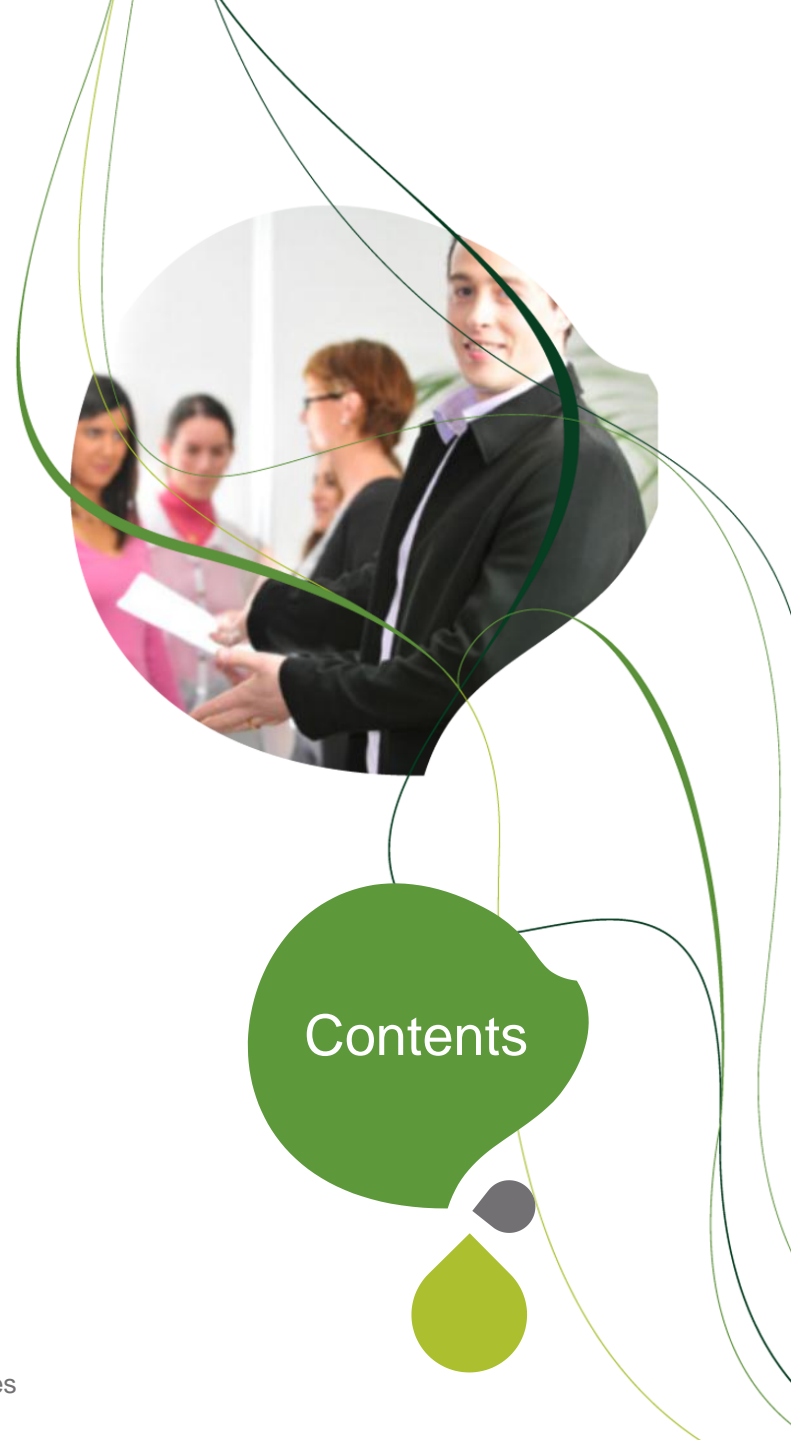
# Exercise

## Exercise 1:

### SETTING UP WEB SERVICES



## 4 – Publishing Subprograms





# 4.1 – The Subprogram Dictionary

## Introduction

A subprogram must be declared in the *Subprogram Dictionary (Development / Processes / Subprograms)* in order to be used as a Web Service.

The dictionary lists the subprogram's arguments (Detected automatically from the code).

A subprogram called as a Web Service may return values to the calling entity via some of its arguments, passed by address rather than by value.



# 4.1 – The Subprogram Dictionary

## Dictionary information

Subprograms that may be called as Web Services have the **Web Service** flag enabled.

Subprogram arguments are detected and populated automatically when the subprogram is declared

- Arguments must be declared line by line in the code to guarantee automatic detection
- Arguments with the same dimension are arranged in groups that will be reflected in the Web Service XML description.

The *Length* column defines a maximum size for character, clob and blob variables. This allows the client application to implement additional controls.



## 4.1 – The Subprogram Dictionary

### Message generation in a subprogram

Messages may be generated and sent to the Web Server during the execution of a subprogram. To make sure messages are correctly generated, you should always use the following calls:

- Information message: `Call MESSAGE(MESSAGE) From GESECRAN`
- Warning message: `Call ADDMESSWARN(MESSAGE) From AWEB`
- Error message: `Call ERREUR(MESSAGE) From GESECRAN`

Messages are transmitted in the XML response data and may be used by the client application.

Up to 50 messages may be generated by a single subprogram.



# 4.1 – The Subprogram Dictionary

## Possibilities and limitations

50 arguments maximum for a subprogram (But all arguments may have dimensions).

The subprogram dictionary is a system table, i.e. it is the same for all the folders in the solution.

50 messages maximum in a subprogram.

Subprograms must not open any X3 window for user interaction. They must run independently using their arguments only.

All display instructions (`Affzo`, `Diszo` etc.) are forbidden. All database instructions are allowed, of course.

The `infbox` instruction does not stop the web service, but will not send any message to the client application.

Emails may be sent via the `Send` instruction, or using the `meladx` utility if server email is wanted.

Reports may be generated using the `Report` instruction. Always use an X3 Print Server.



## 4.2 – Publishing A Subprogram

### The publication utility

In order to publish a subprogram, you must simply define the web service public name.

The publication utility will generate the XML description file in the *X3\_PUB/<FolderName>/GEN/ALL/WEBS* directory on the application server.

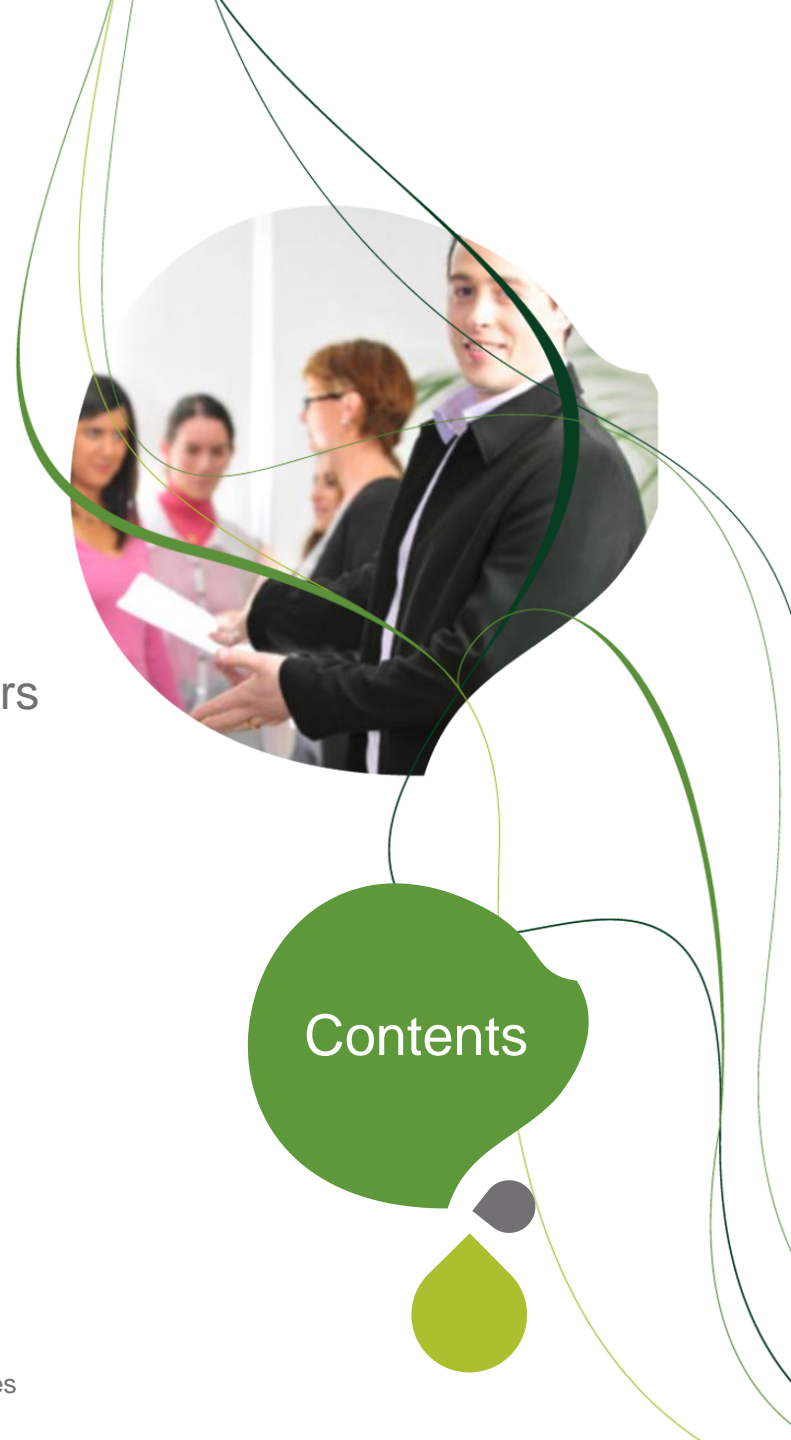
The wrapper program is also created, with the following name: *WJ[WS]* where [WS] is the web service public name.



# Exercise

## Exercise 2.1.1:

Publishing a subprogram : List of customers



## 4.3 – The XML Description File

### The XML description

The XML description file is generated in UTF-8 encoding.

It may be retrieved by the client application, in order to generate forms or interfaces, for example.

It contains the following nodes:

<b>&lt;ADXDOC&gt;</b>	Header node
<b>&lt;GRP&gt;</b>	Argument group
<b>&lt;FLD&gt;</b>	Individual argument



## 4.3 – The XML Description File

### The <ADXDOC> header

<code>&lt;ADXDOC PNA="STOCKSITE"</code>	-----	Subprogram public name
<code>TIM="20060208121856"</code>	-----	Time stamp
<code>PRG="ZSTOCK"</code>	-----	Process name
<code>SPG="CHECK_STOCK"</code>	-----	Subprogram name
<code>FOL="DEMOBRI"</code>	-----	Folder name
<code>SOL="SAZX3"</code>	-----	Solution name
<code>USER="ADMIN"</code>	-----	X3 user who generated the web service
<code>VER="3.20060728"</code>	-----	Generator version
<code>HEAD="1"</code>	-----	Version of the header protocol (SOAP)
<code>WRP="WRSTOCKSITE"</code>	-----	Wrapper program name
<code>C_BRI="Available Stock By Site"&gt;</code>	-----	Designation(s)





## 4.3 – The XML Description File

### The <GRP> and <FLD> nodes

```
<GRP NAM="G1" DIM="1">
```

```
  <FLD NAM="STOCK"
```

```
    TYP="Decimal"
```

```
    PAR="Adr"
```

```
    MOD="Input"
```

```
    C_BRI="Avail. Stock"/>
```

```
</GRP>
```

Argument group code (From the Subprogram Dictionary)

Number of occurrences of that group

Argument group code (From the Subprogram Dictionary)

Argument Type (Integer, Char, Blob, Clob, Date, Decimal)

Addressing mode: "Adr" = By address, "Value" = By value

(Always set to *Input*)

Parameter name (Title)

## 4.4 – The Wrapper Program

### Wrapper program header arguments

The subprogram web service wrapper program acts as an interface between the web server and X3code. It manages:

- Header arguments
- Information / Error / Warning messages
- The subprogram arguments and return values.

The following header arguments are included in the wrapper program:

● <b>WW_OK</b>	Integer	Return status: 1 (OK), 0 (Error)
● <b>WW_ZONE</b>	Character	Not used in this case
● <b>WW_STAT</b>	Integer	Number of messages
● <b>WW_GRAVE</b>	Integer(1..50)	Message level ( 1 (Info), 2 (Warning), 3 (Error) )
● <b>WW_MESS</b>	Character(1..50)	Message text
● <b>WW_ACTION</b>	Character	Not used in this case
● <b>WW_IDENT</b>	Character	Not used in this case
● <b>WW_NB</b>	Integer	Not used in this case
● <b>WW_HORDAT</b>	Character	Web service timestamp
● <b>WW_TAB</b>	Character deletion actions)	Not used (For future line insertion and line
● <b>WW_PAR</b>	Character deletion actions)	Not used (For future line insertion and line
● <b>WW_TRACE</b>	Clobfile	Trace parameters for activity logging

## 4.4 – The Wrapper Program

### Using the wrapper program

The wrapper program contains test code that may be used to make sure the subprogram behaves as wanted, simply by running the wrapper program.

The test code generates a generic trace file with the results.

Two sections, \$INITWS and \$RESULTWS, may be inserted into the original subprogram, in which you can initialise argument values and print additional statements in the trace file. They are called if they exist in the original subprogram.

It is recommended to implement your testing code and modifications in a separate process, since the wrapper program is re-generated each time the Web Service is published again.



## 4.5 – Connection Types

- The **adxtyp** system variable value indicates the session type:

Primary session:	<b>adxtyp = 1</b> (Windows client) – <b>adxtyp = 9</b> (Web client)
Secondary session:	<b>adxtyp = 2</b> (Windows client) – <b>adxtyp = 10</b> (Web client)
Batch connexion	<b>adxtyp = 3</b>
VT Terminal	<b>adxtyp = 5</b>
Web Service	<b>adxtyp = 12</b>

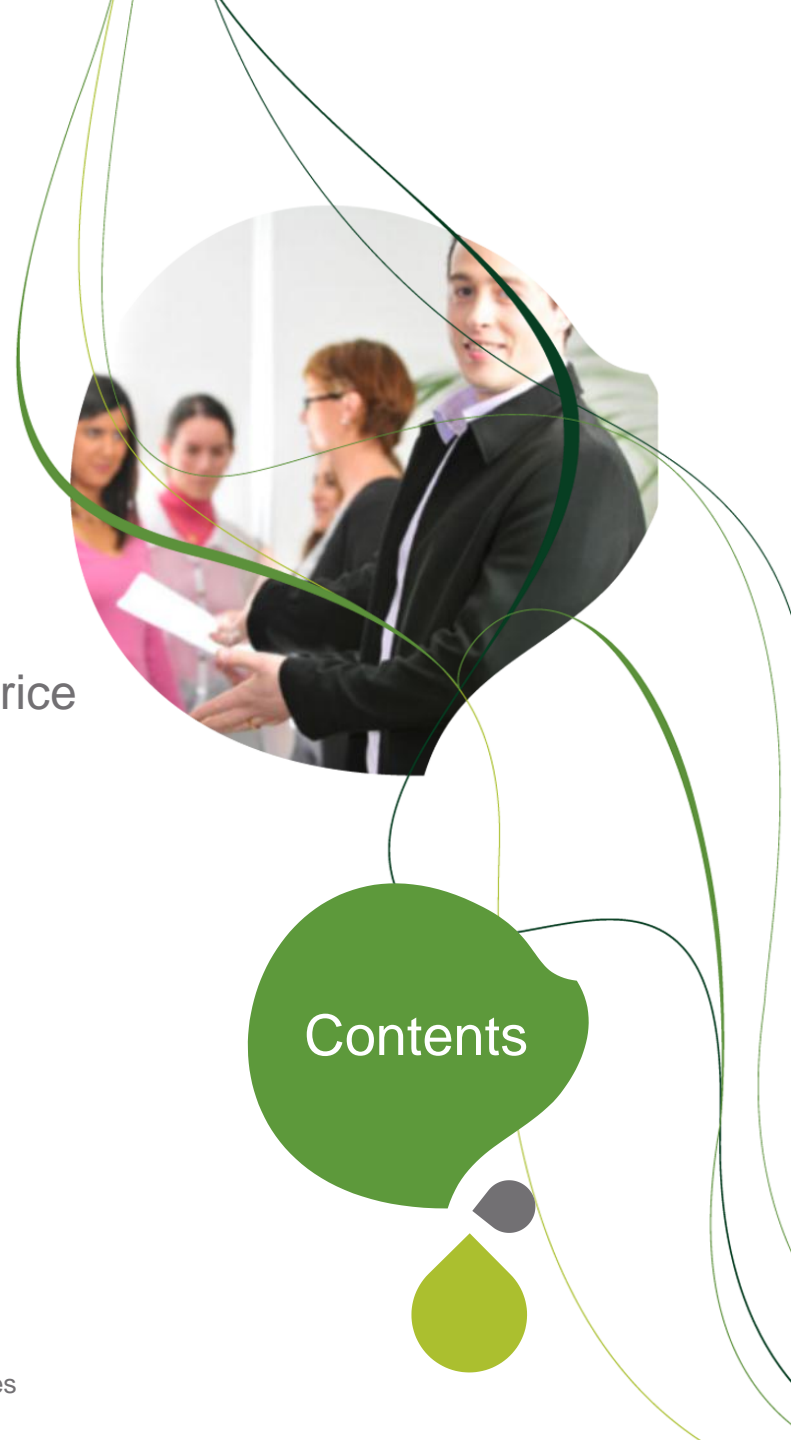
- In web service mode the numeric global variable **GWEBSERV** is set to 1.



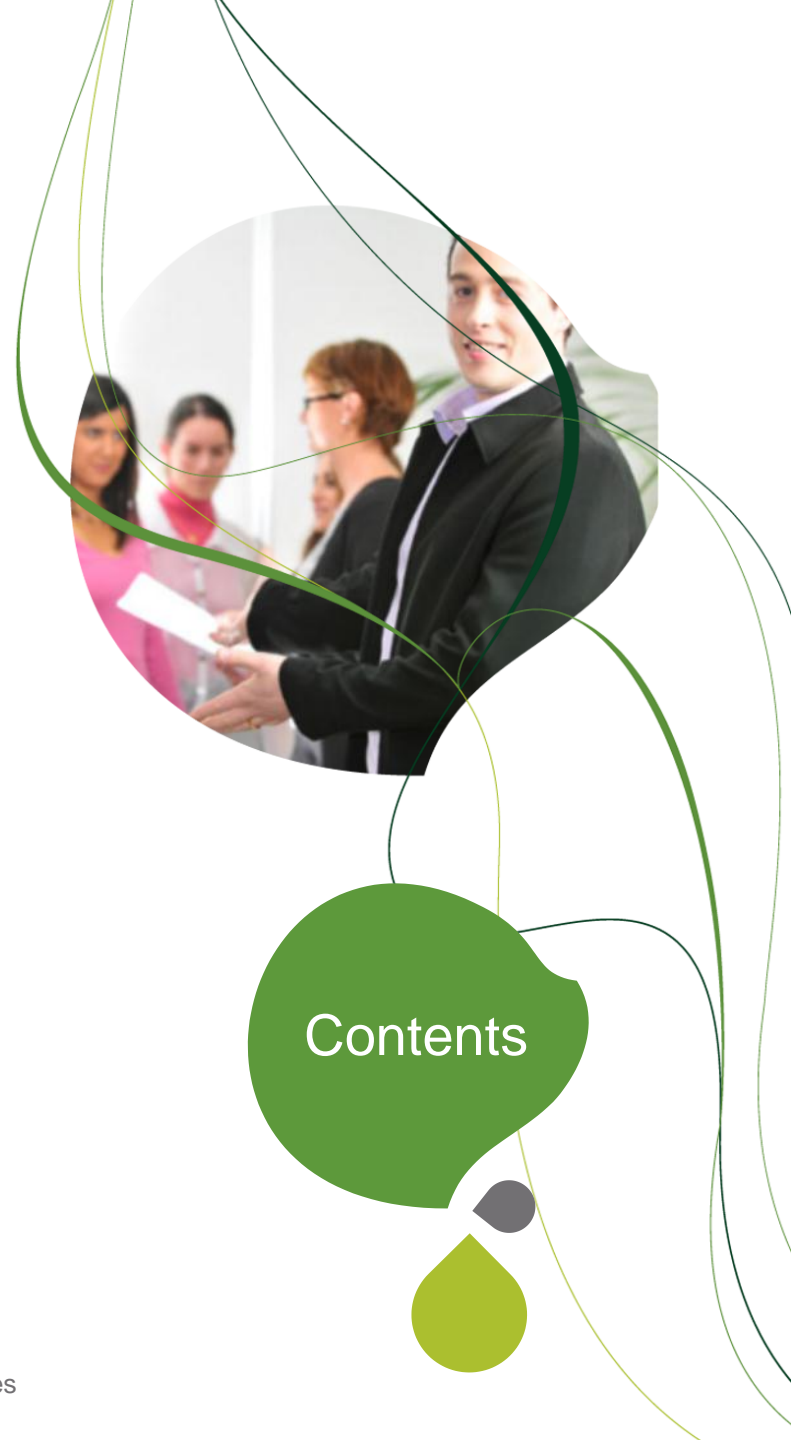
# EXERCISE

## Exercise 2.1.2:

Publishing a subprogram : Get the sales price



## 5 – Publishing Objects



# 5.1 – Publishing An Object

## The object dictionary

X3 objects (Products, Sales Orders, Suppliers etc.) are stored in a dictionary that describes:

- The left browser (Selection parameters)
- The object's screens (Fields, blocks etc.)
- Special options / buttons / menus
- The main object table
- X3 processes linked to the object

To publish an object, no dictionary change is usually necessary. If the object is available in Windows and browser mode, it should also be available through Web Services.

However, slight modifications may be necessary to access additional entry screens, such as addresses screens in Sales Orders. These are done via the *Additional Screens* option in the object dictionary.

The object template is the most widely used function type in X3. Other functions that do not belong to that template (e.g. std process functions such as Automatic Invoice Validation) may usually be encapsulated into subprograms that may be called via Web Services.



# 5.1 – Publishing An Object

## Field types

Object fields may be:

- **Entered:** The field is used and transmitted in both Web Services and traditional clients (Windows or web).
- **Displayed:** The field is transmitted in both Web Services and traditional clients.
- **Invisible:** Invisible fields are never transmitted to the client for traditional clients. For Web Services you have the choice of publishing those fields or not.
- **Technical:** Those fields are never transmitted to either traditional clients or Web Service applications.

Invisible and technical fields are available in the [M] classes. Not transmitting them to the clients simply reduces network traffic.





# 5.1 – Publishing An Object

## Field Actions

*An execution type is attached to each field action in the screen dictionary (Execution field):*

- **Interactive:** Action is executed only in traditional clients (Windows or web).
- **Always:** Action is executed in traditional clients and in web service mode.
- **Import / Web Service:** Only executed in import and web service mode (non-interactive).

*The following actions are never executed in web service mode:*

- **Init Button**
- **ButtonN**
- **Selection**
- **Click** (Icons)



## 5.1 – Publishing An Object

### The web service generator

The web service publication utility published X3 objects, with the choice of:

- **Transaction entry code** (if available). Usually simple entry transactions are built for web services.
- **Including invisible fields:** Invisible fields may or may not be included in the object web service (Technical fields are never published).

The publication utility will generate the XML description file in the *X3\_PUB/<FolderName>/GEN/ALL/WEBS* directory on the application server.

The wrapper program is also created, with the following name:

**V5:** *WJ[OBJ][TRS]* where [OBJ] is the object code and [TRS] is the transaction code.

**V6:** *WJ[WS\_NAME]* where [WS\_NAME] is the publication name of the web service.



# 5.1 – Publishing An Object

## Tuning and limitations

Once an object has been published, it must be tuned to make sure it is compatible with web services.

In standard, the only objects that have been verified are Sale Orders (**SOH**), Customers (**BPC**) and Products (**ITM**).

Other objects must be verified: Depending on parameters and user actions, additional windows may be displayed, which is not compatible with web services.

Main limitations are linked to display problems (`Infbox` instruction, opening additional windows etc.)



## 5.1 – Publishing An Object

### Tuning and limitations

Only 65000 fields are publishable for both objects and subprograms  
(Nb of fields x Dimension)

➔ Important to use a dedicated, customized entry transaction (if available).





Exercise 2.2:

Publishing an object

Exercise 2.3.1:

Using Object's left list

Exercise 2.3.2:

Using Read Method

Exercise 2.3.3:

Using Create method

Exercise 2.3.4:

Using Modify method

Exercise 2.3.5:

Using Delete Line method

Exercise 2.3.6:

Using Delete method

## 5.2 – The XML Description File

### The XML description

The XML description file (*<WebServiceName>.xml*) is generated in UTF-8 encoding.

It may be retrieved by the client application, in order to generate forms or interfaces, for example.

It contains the following nodes:

<b>&lt;ADDOC&gt;</b>	Header node
<b>&lt;GRP&gt;</b>	Screen block (List or Table)
<b>&lt;FLD&gt;</b>	Individual screen field
<b>&lt;ADXKEY&gt;</b>	Left browser
<b>&lt;ADXMEN&gt;</b>	Local menu
<b>&lt;ADXSER&gt;</b>	Methods that may be called for that object (Generic methods and object actions)
<b>&lt;ADXREAD&gt;</b>	Object key field(s) (Index field(s))



## 5.2 – The XML Description File

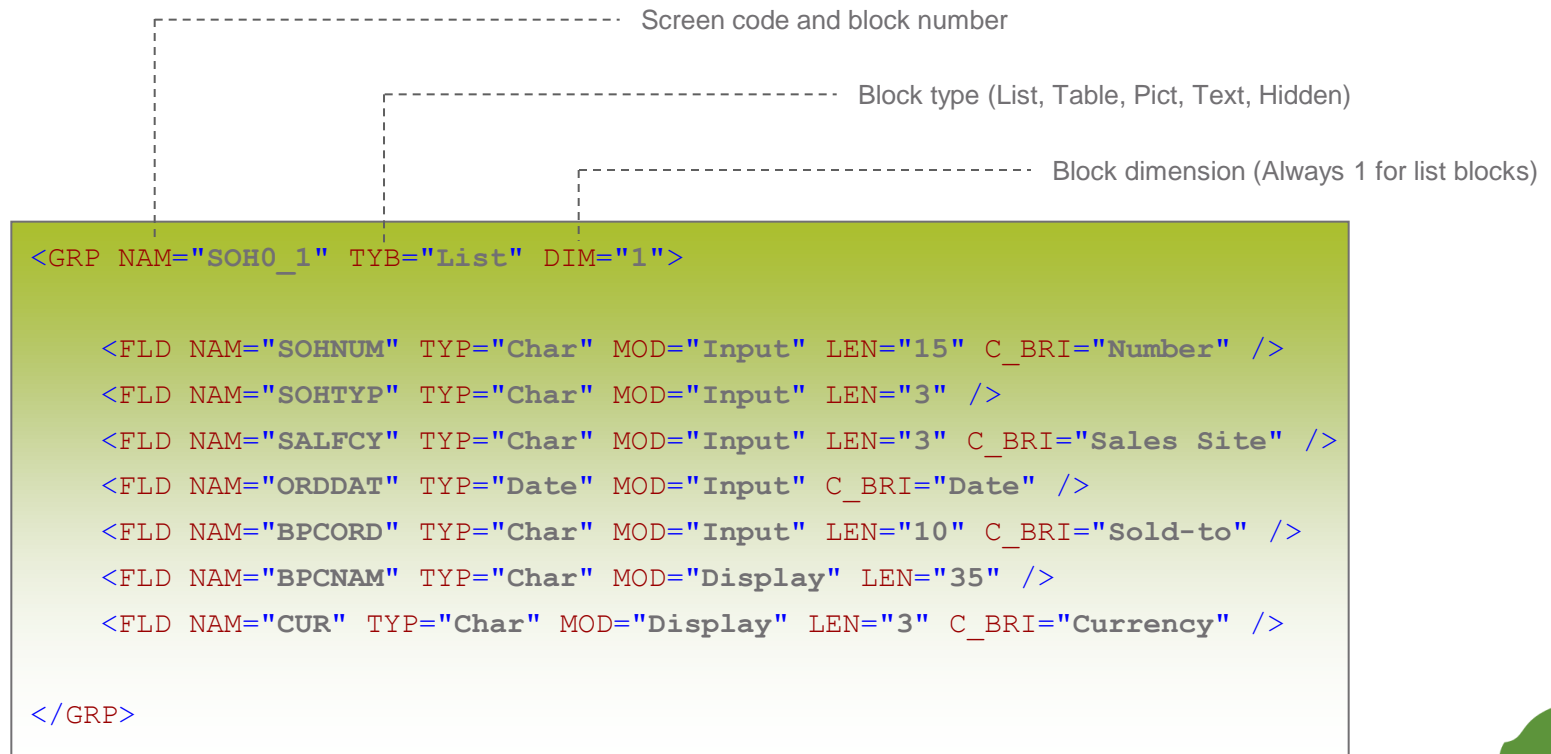
### The <ADXDOC> header

<code>&lt;ADXDOC PNA="WS_SOH"</code>	-----	Subprogram public name
<code>NAM="WGOSOH"</code>	-----	Object window code
<code>TIM="20060208121856"</code>	-----	Time stamp
<code>OBJ="SOH"</code>	-----	Object code
<code>TRA="WEB"</code>	-----	Entry transaction code
<code>FOL="DEMOBRI"</code>	-----	Folder name
<code>SOL="SAZX3"</code>	-----	Solution name
<code>USER="ADMIN"</code>	-----	X3 user who generated the web service
<code>VER="3.20060728"</code>	-----	Generator version
<code>HEAD="1"</code>	-----	Version of the header protocol (SOAP)
<code>WRP="WJWSORDER"</code>	-----	Wrapper program name



## 5.2 – The XML Description File

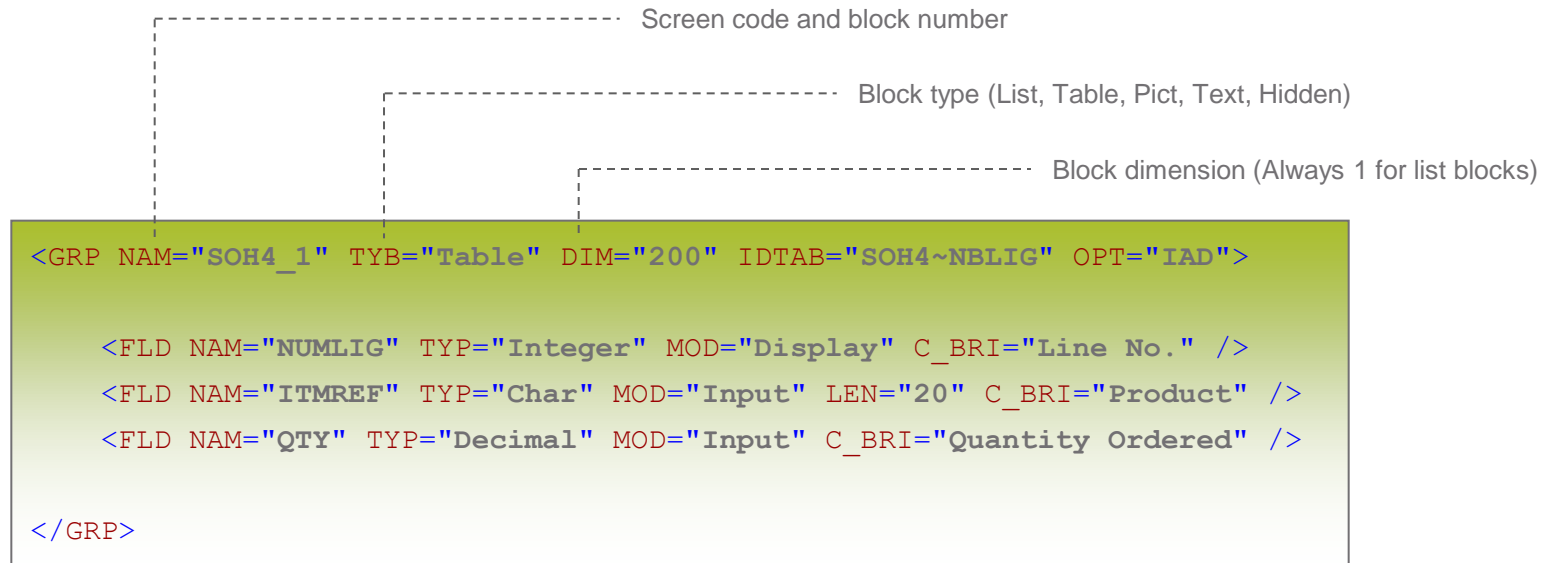
### The <GRP> node for a **list** block





## 5.2 – The XML Description File

The <GRP> node for a **table** block



## 5.2 – The XML Description File

### The FLD node (Screen field)

```
<FLD NAM="MYFIELD"
```

Field code

```
TYP="Integer"
```

Type (Integer, Char, Blob, Clob, Date, Decimal)

```
MOD="Display"
```

Enter type (Input, Display, Hidden)

```
LEN="3"
```

Field length. If the field is a local menu, this attribute is not present

```
DIM="2"
```

Dimension. If the dimension is 1, that attribute is not present

```
MEN="415"
```

Local menu number (Chapter)

```
C_BRI="My Field" />
```

Field name

## 5.2 – The XML Description File

The <ADXKEY> node: Left browsers

```
<ADXKEY> ----- Left browser  
  
  <GRP NAM="LEFTLIST" DIM="10000">  
    <FLD NAM="SOHNUM" ----- Column field code  
      MOD="Display"  
      TYP="Char"  
      LEN="30"  
      C_BRI="Order Num." /> ----- Column title  
  </GRP>  
  
</ADXKEY>
```



## 5.2 – The XML Description File

### The <ADXMEN> node: Local menus

```
<ADXMEN> ----- Local menu group

  <MNU NO="1"> ----- Local menu number (Chapter)
    <VAL IND="1" C_BRI="No" /> ----- Item (Text) number and label
    <VAL IND="2" C_BRI="Yes" />
  </MNU>

</ADXMEN>
```

## 5.2 – The XML Description File

### The <ADXSER> node: Object methods

```
<ADXSER>-----  
  
  <MET ID="READ" C_BRI="Read" />-----  
  <MET ID="CREATE" C_BRI="Create" />  
  <MET ID="MODIFY" C_BRI="Modify" />  
  <MET ID="DELETE" C_BRI="Delete" />  
  <MET ID="SUPLIG" C_BRI="Line deletion" />  
  <MET ID="LIST" C_BRI="List" /> -----  
  <MET ID="V" C_BRI="Valuation" />  
  
</ADXSER>
```

Local menus group

Method code and title

Object action (Button code in object dictionary)



## 5.2 – The XML Description File

### The <ADXREAD> node: Key fields

```
<ADXREAD TAB="SORDER"> ----- Object index fields  
  
  <FLD NAM="SOHNUM" ----- Key field code  
    TYP="Char" ----- Field type  
    LEN="15" ----- Field length  
    C_BRI="Order Num." /> ----- Field title  
  
</ADXREAD>
```



## 5.3 – The Wrapper Program

### Wrapper program header arguments

The object wrapper program acts as an interface between the web server and the X3 object.

It manages:

- Header arguments
- Information / Error / Warning messages
- The object fields as arguments and return values.

The following header arguments are included in the wrapper program:

● <b>WW_OK</b>	Integer	Return status: 1 (OK), 0 (Error)
● <b>WW_ZONE</b>	Character	Error field if there was an error
● <b>WW_STAT</b>	Integer	Number of messages
● <b>WW_GRAVE</b>	Integer(1..50)	Message level ( 1 (Info), 2 (Warning), 3 (Error) )
● <b>WW_MESS</b>	Character(1..50)	Message text
● <b>WW_ACTION</b>	Character	Action/Method code (Action to execute): READ / CREATE / MODIFY / DELETE / SUPLIG
● <b>WW_IDENT</b>	Character	Identifier (Key1~Key2~Key3... or start value or left list quick select criteria)
● <b>WW_NB</b>	Integer	Number of records to read (Left list) or number of records returned
● <b>WW_HORDAT</b>	Character	Web service timestamp
● <b>WW_TAB</b>	Character	For line deletion actions
● <b>WW_PAR</b>	Character	For line deletion actions
● <b>WW_TRACE</b>	Clbfile	Trace parameters for activity logging



## 5.3 – The Wrapper Program

### Using the wrapper program

The wrapper program contains test code that may be used to make sure the object behaves as wanted, simply by running the wrapper program with the proper parameters for each object action/method.

The test code generates a generic trace file with the results.

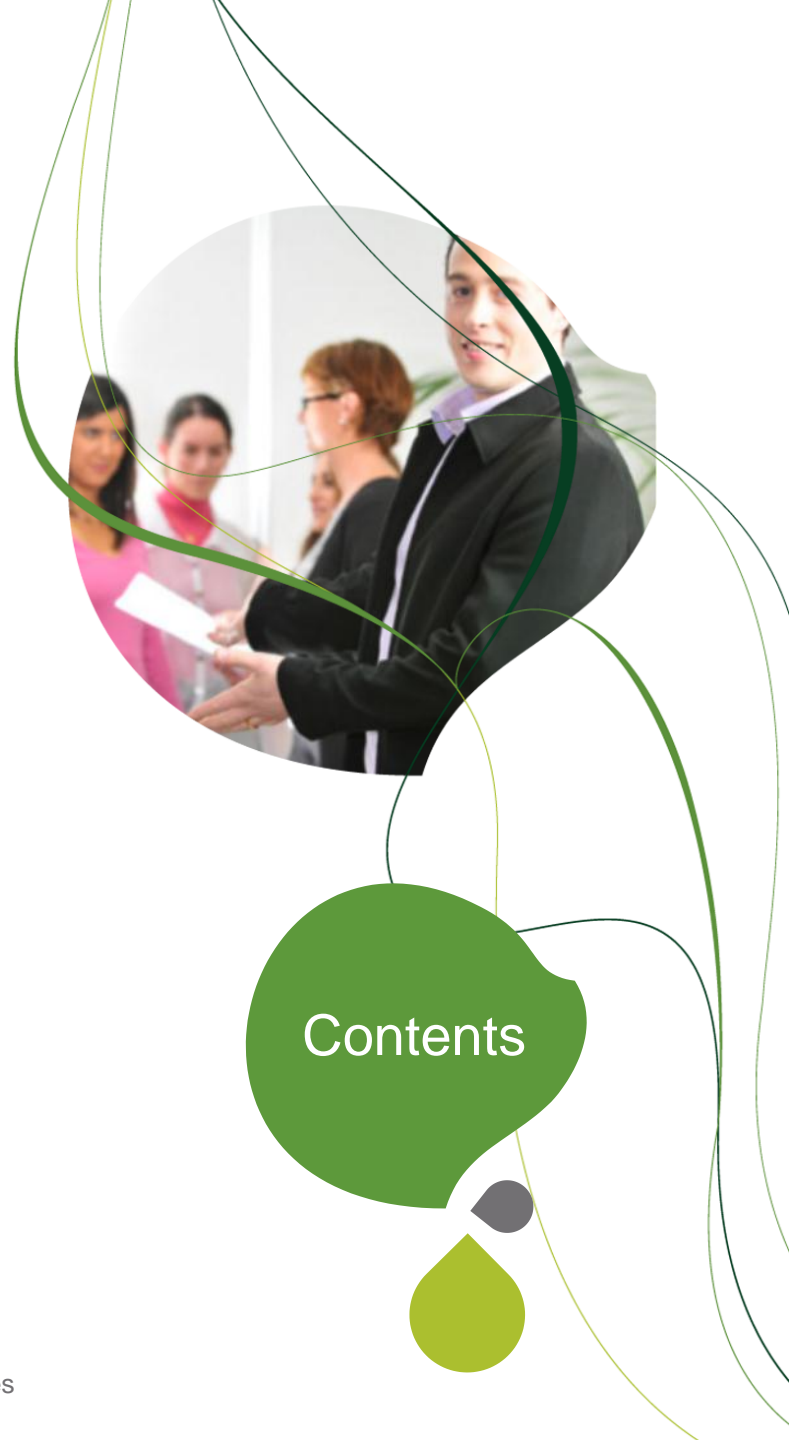
A number of sections may be inserted into a separate program called **ZZWS[OBJ]** where [OBJ] is the object code. They will be called if they exist:

- **\$INITMAJ**: Initialisation of parameters for object actions/methods (Read, Modify, Create, Delete and special object actions).
- **\$RESMAJ**: Analysis of results and additional trace file statements for object actions/methods.
- **\$INITLG**: Initialisation of parameters for left list method.
- **\$RESLG**: Analysis of results and additional trace file statements for left list method.





## 6 – Testing Web Services



Contents

# 6.1 – The Web Service Tester

## The Web Service Tester

The Web Server includes a Web Service Tester, accessible in the web server technical page.

It is a simple web service client that implements the three types of methods X3 may publish: Subprograms, object methods and the object list.

It works by sending the raw XML data provided by the user to the web service and simply displaying the results as raw XML data in corresponding text fields.



## 6.1 – The Web Service Tester

### Setting up the pool

***Pool Entry Group:*** Public name of the web service pool to use (As defined in the console).

***Adonix User / Password:*** X3 user name and password to use for connection.

***Language Code:*** Language to use for the X3 connection.

The ***Save Context*** button will save the above parameters for future use.



# 6.1 – The Web Service Tester

Testing subprograms, objects and lists

The **Web Service** section holds the public name and generic parameters for executing the web service.

The **Action** section contains the methods for the web service type to be tested:

- **Execute** for subprograms
- **Read, Create, Modify** etc. for objects
- **List** for object lists
- A common **Description** action for all web service types that gets the web service XML description.

A **Subprogram Data** section allows users to enter raw XML input data (see subsequent sections in this course)

Four output sections will display web service result data (all available within the XML output stream)

- **Reply Description** is a verbose description of the final processing status.
- **XML Reply** shows the response part where actual X3 data is contained.
- **Data Process Log** is a detailed account of how the input XML data was processed.
- **Request Trace** is the detailed log of the whole process.



## 6.2 – XML Data Streams

### The XML protocol

An XML data stream starts with a **header** or **declaration** in the following format:

```
<?xml version="1.0" encoding="UTF-8"?>
```

XML data is enclosed into **elements** with the following format:

**<TAG Property="Value">Content</TAG>**

Start and end delimiters are called **tags**. The element's content (Or value) is anything that appears between the tags.

Elements may be nested into other elements, in which case they are called **child elements**.

```
<PARAM>
  <FLD NAME="SOHNUM">SO000052</FLD>
  <FLD NAME="BPCORD">C001</FLD>
  <FLD NAME="ORDDAT">20061124</FLD>
  . . .
</PARAM>
```

Elements have **properties** that are identified inside the start tag (Property="Value").

For X3 web services, the body of the input data is enclosed in a **<PARAM>** element (Input parameters), and individual fields or arguments are in a **<FLD>** child element.



## 6.2 – XML Data Streams

### XML input for subprograms

Subprogram arguments are enclosed in the **<PARAM>** section, inside **<FLD>** nodes.

The argument code is indicated by the **NAME** attribute.

Only input arguments (mandatory arguments) are necessary, i.e. subprogram parameters that are required for the subprogram to function.

```
<PARAM>  
  <FLD NAME="ITMREF">CD100</FLD>  
</PARAM>
```



## 6.2 – XML Data Streams

- XML input for objects

An XML data stream must be given to the web service when:

- **Creating** an object: All necessary fields must be included in the <FLD> node. The <GRP> nodes are not necessary and the only attributes that need to be transmitted is the **NAME**.
- **Modifying** an object: Only modified fields need to be transmitted. In addition to the input XML data, the object key values must be specified.

```
<PARAM>
  <FLD NAME="SOHNUM">SO000052</FLD>
  <FLD NAME="ITMREF">CD100</FLD>
</PARAM>
```

For other object actions, no input XML data is necessary. The tester interface provides the necessary fields:

- **Read, Delete:** Only the object's key (Index fields) is required. Corresponding values are populated in the *Key(s)* section.
- **List:** Left list criteria are entered in the *Criteria* section. No input XML stream is required.

The key is also required for the **Modify** action in addition to XML data, since the system needs to know which record is being modified.

## 6.2 – XML Data Streams

### XML input data for table screens

Objects may contain table screens (E.g. SOH4 in Sales Orders).

In that case the screen is represented by a **<TAB>** element with an **ID** property identifying the block.

- Inside the **<TAB>** element, **<LIN>** child elements represent an individual line, with the **NUM** property indicating the line number.

```
<PARAM>
  <FLD NAME="SOHNUM">SO000052</FLD>

  <TAB ID="SOH4_1">
    <LIN NUM="1">
      <FLD NAME="ITMREF">CD100</FLD>
      <FLD NAME="QTY">99</FLD>
      <FLD NAME="GROPRI">25</FLD>
    </LIN>
    <LIN NUM="2">
      <FLD NAME="ITMREF">CD100</FLD>
      <FLD NAME="QTY">99</FLD>
      <FLD NAME="GROPRI">25</FLD>
    </LIN>
  </TAB>
</PARAM>
```

----- Screen SOH4, block 1 (ID="SOH4\_1")

----- Line 1 (The **NUM** attribute is not mandatory)

----- Line 2



## 6.2 – XML Data Streams

### XML input data for arrayed fields inside list blocks

Array fields inside list blocks (E.g. Company Name in BPs) are represented inside a **<LST>** element with the **NAME** property containing the field name.

The **<LST>** element contains nested **<ITM>** elements with individual values for each index.

```
<PARAM>
  <FLD NAME="BPRNUM">C0001</FLD>
  <FLD NAME="BPRSHO">B&J Toys</FLD>

  <LST NAME="BPRNAM">
    <ITM>B&J Toys</ITM> ----- BPRNAM(0)
    <ITM>(Harrods Group)</ITM> ----- BPRNAM(1)
  </LST>

</PARAM>
```



## 6.2 – XML Data Streams

### Return XML data streams

#### Status and messages:

- The returned XML stream contains a section that indicates the status of the request and possible information, warning or error messages.

#### Technical information:

- It also contains a technical section that contains the detailed processing log (If logging has been enabled) and timing information.

#### Data:

- For all web service methods (Object, list, subprogram), the system returns an XML stream that corresponds to the **<ADXDATA>** element in the XML web service description with values populated (See chapter 5.2 – **The XML Description File** for more details).
- The data is contained in the **<RESULT>** element in the returned XML stream.



## 6.2 – XML Data Streams

### Data integrity checks

In X3, data integrity and contention management is carried out using symbolic locks. Only a single user is allowed to modify an object at any point in time, and the record is locked as soon as the user accesses it.

In Web Services, the **Read** and **Modify** methods are separated. The read method does **not** lock the object (there is no point in locking the object when it is read in web service mode).

To check that an object has not been modified between the time when the data has been read and the time when it is modified, the following procedure must be followed:

- The **Read** method returns two technical fields enclosed in the **<ADXTEC>** element in the returned XML stream:  
**WW\_MODSTAMP**      Time stamp of latest object update  
**WW\_MODUSER**      Modification user
- If the WW\_MODSTAMP field is included in the input XML for the **Modify** method, the system checks that the object has not been modified since that time.  
If the object has been modified, an error message is returned  
(*"Record modified since last read"*)



## 6.2 – XML Data Streams

### Table/detail screens

Table/detail screens require special care when they are accessed in web service mode. The most frequent example is the **Addresses** screen in BPs, Customers, Suppliers etc.

To be accessed through web services, those screens must comply with the following standards:

- Entered field code (right part) = Constant character + invisible field code (E.g. **XBPAcry** = **X** + **BPAcry**).
- The last invisible numeric (**C**) field in the block is the line number control variable. It is initialised by the wrapper program for each line when the screen is used.

Only the fields in the table block (left part) are transmitted through web services and need to be populated when creating or modifying. All invisible fields are automatically included in the web service description.

When modifying records in those screens, the table variable (**NBLIG**, **NBADR** etc.) must be set to the correct number. If it is not, then the last line in the input XML stream is taken as the last line in the screen.



## 6.2 – XML Data Streams

### Additional screens (Object dictionary)

It is possible to add to an object up to 8 additional screens with their respective abbreviation.

The system will:

- Include those screens in the XML structure generated when the object is published.
- Open the screens in the window's automatic process, and declare them as window tabs.
- Load the screen with the input XML stream (If any).
- Include the screens in the response XML stream.

The application (X3 code) must ensure those screens are used correctly in web service mode (**GWEBSERV=1**).

The standard SOH object has been modified to illustrate the use of additional screens.



## 6.3 – A Java Tester For X3 Web Services

A Java Tester to easily test X3 web services

Sage developers have designed a basic Java web service test application that automatically generates a graphical interface based on the web service description. It allows the user to avoid the work involved in constructing a correct XML stream.

### The Java Tester

- Implements available X3 web service methods (Object, List, Subprogram)
- Automatically constructs XML input streams
- Shows execution trace files
- Shows XML responses
- Includes a load builder for performance testing



## 6.3 – A Java Tester For X3 Web Services

### Load testing

Once a web service is published, it has to be load-tested (Or benchmarked) to assess the global robustness and performance rating of the web service system.

- Important: Creating an object record via web services is *slower* than using the X3 import templates.

Load testing is also important to assess the maximum number of connections that should be available in the web service licence for acceptable response times.

Load testing will show you the total CPU and memory usage on the web server, which may be used as an estimation of future needs for the live system.

The Java Tester provides a load testing section that may be used to simulate important web service activity according to a specified scenario, and for a specified number of simultaneous connexions.



## 7 – Integrating Web Services





# 7.1 – The WSDL Description

## Web services and WSDL

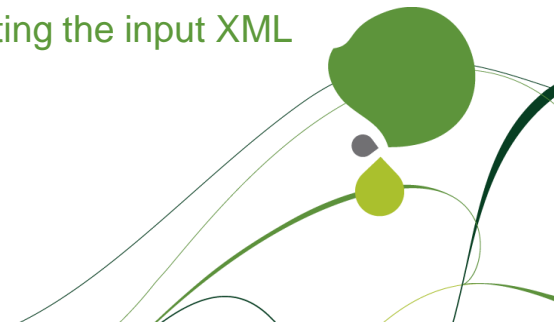
The web service protocols include a description standard (different from the one generated directly by X3) integrated in an XML structure. It is called the **WSDL** (**Web Service Description Language**).

Normally, each field or argument to be transmitted to a web service must be described in the WSDL structure. In X3 however, a single web service is used for all objects.

In order not to have a large number of WSDL descriptions, the "internal" structure (screen fields, subprogram arguments etc.) is described as a variable XML flux included as raw XML in single "generic" WSDL, where only methods (**read**, **create**, **query** etc.) are fixed.

This also avoids having to change your client application each time a field is modified or added in an X3 screen.

As a consequence, the client application has the responsibility of constructing the input XML stream and parsing the response XML stream.



# 7.1 – The WSDL Description

## The WSDL structure and stub classes

The WSDL file includes detailed information about the web service's methods and how to implement them in the form of complex data types and functions.

The X3 web service WSDL file is published by the web server and may be accessed through the following URL:

[http://\[WebServer\]/adxwsvc/services/CAdxWebServiceXmlCC?wsdl](http://[WebServer]/adxwsvc/services/CAdxWebServiceXmlCC?wsdl)

When processed through a web service-enabled development platform such as Visual Studio® .NET or the Java + Axis wsdl2java toolkit, the wsdl file is transformed into a set of **stub classes** (either Java or in a .NET language) with the appropriate methods and properties to quickly access the web service.

Stub classes are also shipped ready to use with the X3 Web Server and may be downloaded from the **Web Service Server** section on the technical page.



## 7.1 – The WSDL Description

Solution 1: Integration of WSDL in the development platform and generation of stub classes



## 7.1 – The WSDL Description

Solution 2: Direct integration in your project stub classes downloaded from X3 Web Server



X3 Web Server

[Download Eclipse 3.1 Workspace](#)  
[Download Visual Studio 2005 Solution](#)



## 7.2 – The Stub Classes

### Main X3 web service stub classes

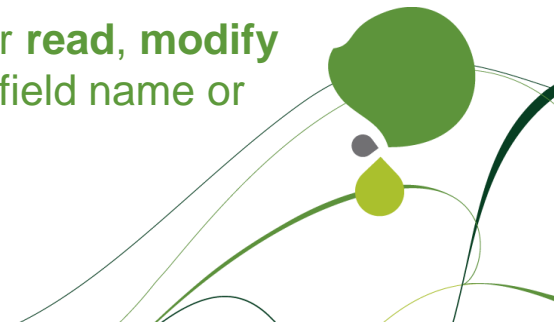
**CAdxWebServiceXmlCCService:** The main web service class with the following methods:

<b>run()</b>	Run subprogram
<b>read()</b>	Read object record
<b>save()</b>	Create object record
<b>modify()</b>	Modify object record
<b>query()</b>	Run object left list
<b>delete()</b>	Delete object record
<b>getDescription()</b>	Get web service description

**CAdxCallContext:** This class manages the connection and processing parameters. An instance of this is passed to practically all CAdxWebServiceXmlCCService methods. It has the following properties:

string <b>codeLang</b>	Language code
string <b>codeUser</b>	X3 user code
string <b>password</b>	X3 user password
string <b>poolAlias</b>	Name of the web service pool to connect to
string <b>requestConfig</b>	Configuration string (Debugger, trace file activation etc.)

**CAdxParamKeyValue:** This class represents an object key (For **read**, **modify** and **delete** methods). It has two public properties, **key** (The X3 field name or rank in the main index) and **Value** (The key value).



## 7.2 – The Stub Classes

### The **CAdxCallContext.requestConfig** property

The **requestConfig** property in the **CAdxCallContext** class contains a string with elements separated by the **&** character. It specifies request processing settings:

<code>adxwss.trace.on=[on/off]</code>	Activates the web server trace option
<code>adxwss.trace.size=16384</code>	Maximum trace file size (Do not change)
<code>adonix.trace.on=[on/off]</code>	Activates the X3 server trace option
<code>adonix.trace.level=(1/2/3)</code>	Trace level (Input, output, Input and output)
<code>adonix.trace.size=8</code>	Maximum trace size (Clob size, do not change)
<code>adonix.debug.on=(on/off)</code>	Activates the X3 debugger
<code>adonix.debug.host=localhost</code>	Debugger client machine (Where the debugger application is run)
<code>adonix.debug.port=1789</code>	Debugger port

Example:

```
requestConfig =  
"adxwss.trace.on=off&adxwss.trace.size=16384  
&adonix.trace.on=off&adonix.trace.level=1&adonix.trace.size=8  
&adonix.debug.on=off&adonix.debug.host=localhost&adonix.debug.port=1789"
```



## 7.2 – The Stub Classes

### X3 web service response classes

**CAdxResultXml:** This class contains the actual results in an XML clob, along with the status, messages and technical information.

int <b>status</b>	Response status (1 = OK, 0 = Not OK)
CAdxMessage[] <b>messages</b>	An array of X3 messages (See below)
string <b>resultXml</b>	The XML output or return string (Raw XML format)
CAdxTechnicalInfos <b>technicalInfos</b>	Technical information regarding request processing (See below).

**CAdxMessage:** This class manages information/warning/error messages returned by X3. It has two public properties:

string <b>type</b>	The message type (1 = information, 2 = warning, 3 = error)
string <b>message</b>	The actual message

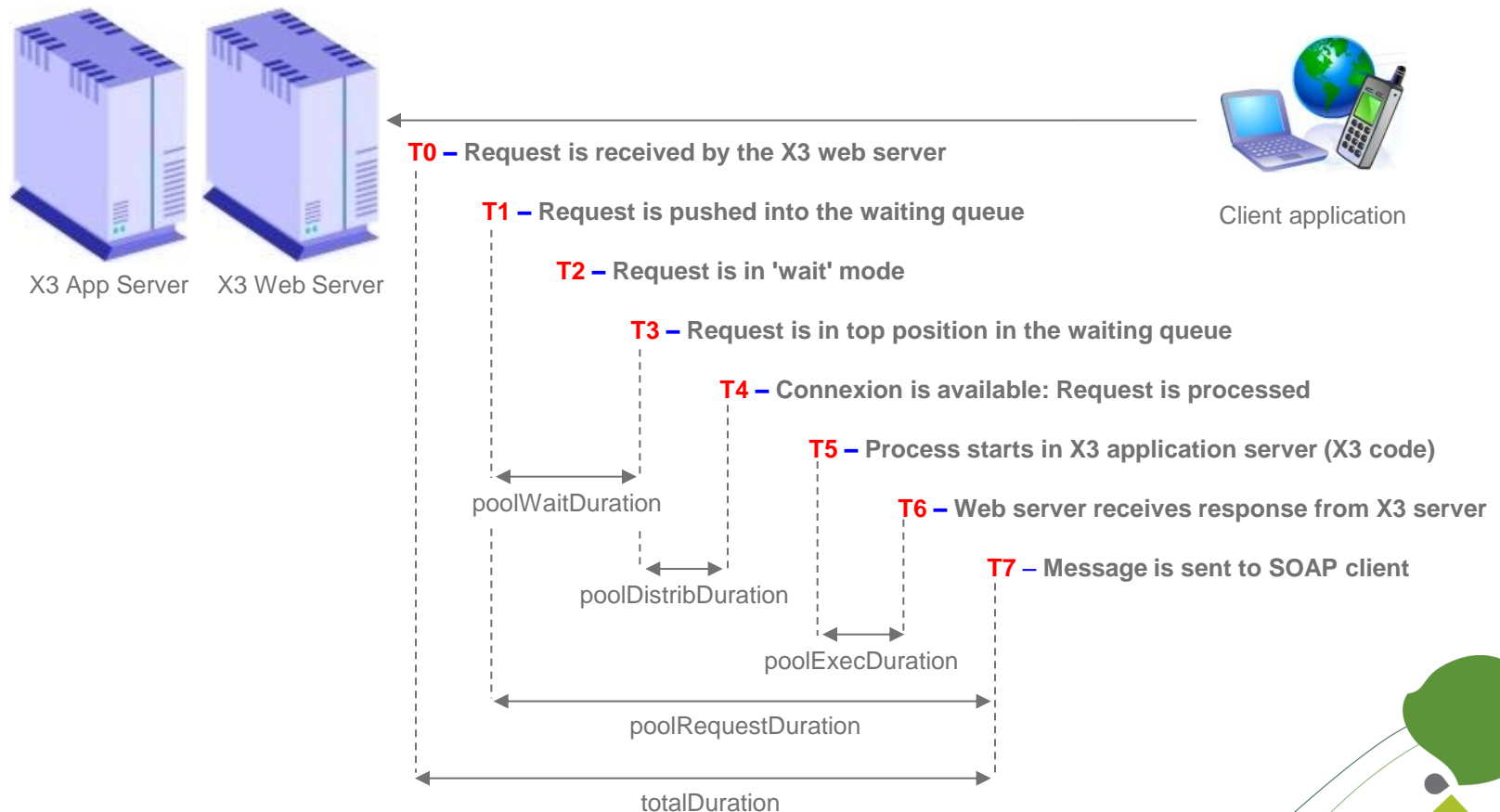
**CAdxTechnicalInfos:** This class manages technical information regarding request execution, performance, logs etc. It has a number of properties that indicate:

- The time it took to process the request at each stage (Connection pool distribution, X3 code execution, wait times etc.)
- The request processing log (Trace file)
- Technical information on user and language switching, number of distribution cycles etc.



## 7.2 – The Stub Classes

### CAdxTechnicalInfos: Duration times





## 7.3 – .NET Development

### Outline

Using X3 web services in an external application amounts to:

- Building an XML header with the context information (Using the CAdxCallContext class)
- Building an XML string containing input parameters (Using basic XML or String tools)
- Calling the service via the stub classes (Using the CAdxWebServiceXmlCC class)
- Parsing/displaying the XML results using the CAdxResultXml class
- Parsing/displaying error messages and tech info (If necessary) using CAdxTechnicalInfo and CAdxMessage.

Most of the code in your application will usually be dedicated to managing the interface and building or parsing Xml streams.



## 7.3 – .NET Development

### Preparing the solution

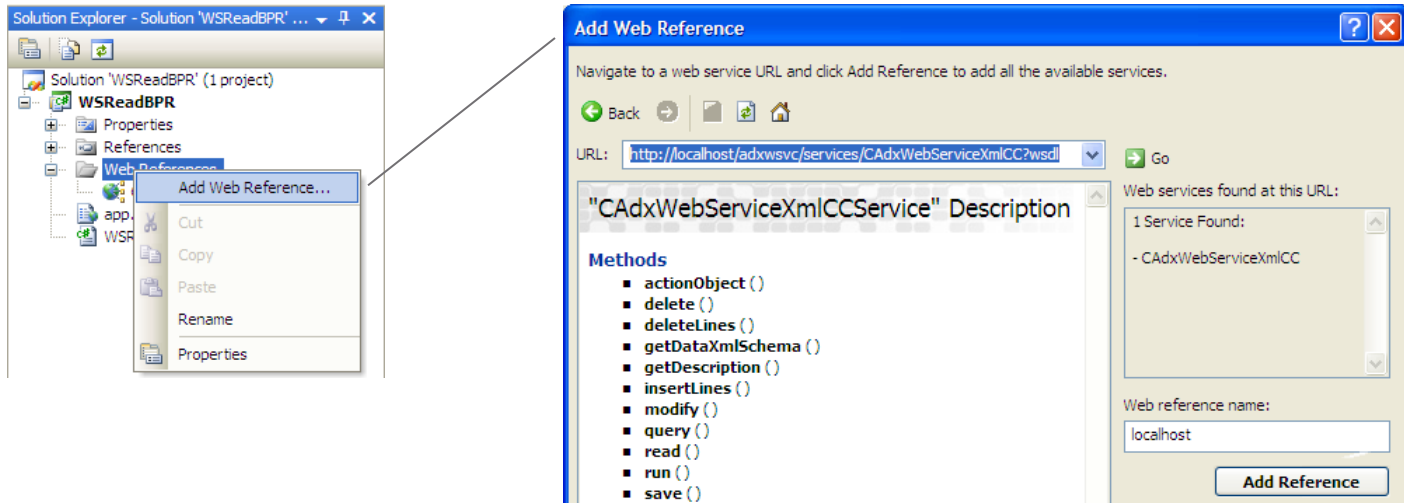
*Examples in this section are developed in the C# language using Microsoft® Visual Studio® 2005.*

In order to prepare a .NET solution for web services, you must add to it a **web reference**, which is the name of a web service WSDL in the .NET environment.

The web reference URL is

[http://\[WebServer\]/adxwsvc/services/CAdxWebServiceXmlCC?wsdl](http://[WebServer]/adxwsvc/services/CAdxWebServiceXmlCC?wsdl) (See section 7.1).

When this is added to the solution, the Visual Studio environment automatically creates the stub classes needed for web service access.



## 7.3 – .NET Development

### System classes to use

Once the web reference is imported, you must use the main class in your project:

```
using CAdxWebServiceXmlCC;
```

You must also use a number of other .NET classes:

```
using System.Text;  
using System.Xml;  
using System.IO;
```



## 7.3 – .NET Development

### The calling context

The first step in calling a web service is preparing the call context structure, which will be passed as a parameter to all the web service's methods (See section 7.2):

```
// Calling context - class CAdxCallContext
CAdxCallContext cc = new CAdxCallContext(); // Instance of CAdxCallContext
cc.codeLang = "BRI";                        // Language code
cc.codeUser = "ADMIN";                      // X3 user
cc.password = "";                           // X3 password
cc.poolAlias = "WS_DEMOBRI";                // Pool name
cc.requestConfig = "adxwss.trace.on=on&adxwss.trace.size=16384
                  &adonix.trace.on=on&adonix.trace.level=3
                  &adonix.trace.size=8"; // Request configuration string
```



## 7.3 – .NET Development

Preparing an instance of the web service and the results class

The next step is to create an instance of the web service, and to prepare a variable of type **CAdxResultXml**, the class that will hold the results:

```
// Web service instance
CAdxWebServiceXmlCCService wsvc = new CAdxWebServiceXmlCCService();

// XmlResult instance
CAdxResultXml result = new CAdxResultXml();
```



## 7.3 – .NET Development

### Preparing the input XML string

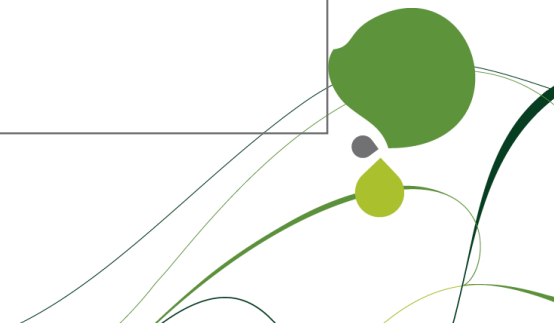
Most of the X3 web service methods take an XML input stream as an argument (amongst others).

The XML input stream may be built as a simple string, or using available .NET XML utilities and builders. The **run** method accepts strings as the third argument.

The **String.format()** method is also convenient for preparing short input XML strings.

```
// Prepare XML
StringBuilder xmlBuilder = new StringBuilder("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
xmlBuilder.Append( "<PARAM><FLD NAME=\"ITM\">" );
xmlBuilder.Append( "CD100" );
xmlBuilder.Append( "</FLD></PARAM>" );

String xmlInput = xmlBuilder.ToString(); // This is the input XML string
```



## 7.3 – .NET Development

### Calling the **run** method (Subprograms)

The **run** method takes 3 arguments:

- The calling context (CAdxCallContext, see chapter 7.2 and previous sections in this chapter)
- The subprogram's public name
- The input XML string

```
// Run subprogram
// run method: run(CAadxCallContext <Context>, String <WebServiceName>, String <XML Input>)
// XML Input:
// XML header + <PARAM><FLD NAME="field1">value1</FLD><FLD NAME="field2">value2...</PARAM>
result = wsvc.run( cc, "GETITMPRI", xmlInput );
```



## 7.3 – .NET Development

### Calling the **create** method (Objects)

Just like **run**, the **create** method takes 3 arguments:

- The calling context (CAdxCallContext, see chapter 7.2 and previous sections in this chapter)
- The subprogram's public name
- The input XML string

```
// Create BPR object  
// Method: save(CAadxCallContext <context>, String <publicname>, String <xmlData>)  
result = wsvc.save( cc, "WS_BPR", xmlData );
```





## 7.3 – .NET Development

### Calling the **read** method (Objects)

**read** does not take any XML input stream. Instead, you must use an array of instances of the `CAdxParamKeyValue` class to indicate which record is to be read. Each instance contains one key element. For simple objects, there will be only one element in the array:

```
// Record key: An array of CAdxParamKeyValue
CAdxParamKeyValue[] objKey = new CAdxParamKeyValue[1]; // Only one element (Simple object)
objKey[0] = new CAdxParamKeyValue();
objKey[0].key = "1"; // First element in the index
objKey[0].value = "C0001"; // Element value

// Method: read(CAdxCallContext <context>, String <publicname>, CAdxParamKeyValue <key>)
result = wsvc.read(cc, "WS_BPR", objKey);
```



## 7.3 – .NET Development

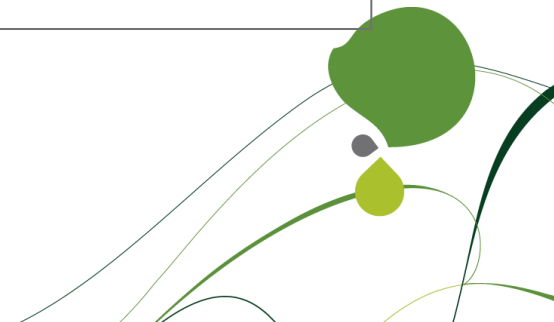
### Calling the **modify** method (Objects)

**modify** is a combination of read and save. It takes both the CAdxParamKeyValue array to know which records to access, and an input XML stream to know which fields to modify:

```
// Record key: An array of CAdxParamKeyValue
CAdxParamKeyValue[] objKey = new CAdxParamKeyValue[1]; // Only one element (Simple object)
objKey[0] = new CAdxParamKeyValue();
objKey[0].key = "1"; // First element in the index
objKey[0].value = "C0001"; // Element value

// Input XML: Field to modify is CRY
String xmlHeader = new String("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
String xmlInput = new String("<PARAM><FLD NAME=\"CRY\">US</FLD></PARAM>");

// Method: modify(<context>, String <publicname>, CAdxParamKeyValue <key>, String <xmlData>)
result = wsvc.modify(cc, "WS_BPR", objKey, String.Concat(XmlHeader,xmlInput));
```



## 7.3 – .NET Development

### Calling the **query** method (Object list)

**query** returns data corresponding to the object's left browser columns. It also takes an array of `CAdxParamKeyValue` instances as an argument, but this time it represents quick select criteria for each of the left browser's columns.

```
// Read object
CAdxParamKeyValue[] objKey = new CAdxParamKeyValue[3];

objKey[0] = new CAdxParamKeyValue();
objKey[0].key = "1";    // First column (BPRNUM)
objKey[0].value = "C*"; // Searching for customer codes that start with "C"
objKey[1] = new CAdxParamKeyValue();
objKey[1].key = "2";
objKey[1].value = "*";
objKey[2] = new CAdxParamKeyValue();
objKey[2].key = "3";
objKey[2].value = "*";

// Method: query(CAdxCallContext <context>, String <public name>,
               CAdxParamKeyValue[] <keys>, int <number of elements>)
result = wsvc.query( cc, "WS_BPR", objKey, 200 );
```

## 7.3 – .NET Development

### The result XML stream

The result XML stream is contained in the **CAdxResultXml** instance. This class contains:

- The actual XML result as a string: **CAdxResultXml.resultXml**
- The response status as an integer: **CAdxResultXml.status** (1 = OK, 0 = not OK)
- X3 messages that may have been generated, in **CAdxResultXml.messages**. This is an array of **CAdxMessage** instances. The **CAdxMessage** class has the following properties:
  - CAdxMessage.type**: The message type (1 = information, 2 = warning, 3 = error)
  - CAdxMessage.message**: The actual message text
- **CAdxResultXml.messages.Length** indicates the number of messages that were generated (Length is a common property of all arrays).



## 7.4 – Java Development

### Preparing your project

*Examples in this section are developed using Eclipse 3.1*

The process of preparing the Java project to generate and use the stub classes is the same as in a .NET environment. The best tool to use in a Java environment is the Axis Ant task and **wsdl2java**, that convert a WSDL files to Java stub classes.

A number of wsdl2java wizards have been developed as Eclipse plug-ins. They behave like the .NET "Add Web Reference" wizard seen in the previous section in this course.

You can also download the stub classes directly from the X3 web server's *Web Service Server / Axis Stubs* section.



## 7.4 – Java Development

### Importing the Java stub classes

Once the stub classes have been generated, they must be included in your project's directory / build path and must be imported into your project:

```
/* X3 Web Service stub classes */
import com.adonix.wsvc.stubs.CAdxCallContext;
import com.adonix.wsvc.stubs.CAdxWebServiceXmlCC;
import com.adonix.wsvc.stubs.CAdxWebServiceXmlCCServiceLocator; // Note this new one
import com.adonix.awss.stubs.CAdxResultXml;
import com.adonix.awss.stubs.CAdxMessage;
```

Other Java SDK classes will also be necessary (A few of them are listed here):

```
import java.io.StringReader; // String reading/building classes
import javax.xml.parsers.DocumentBuilder; // DOM builder
import javax.xml.parsers.DocumentBuilderFactory;
import org.xml.sax.InputSource;

import org.w3c.dom.Document; // May be useful for Xml document parsing
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
```

[...]

## 7.4 – Java Development

### The web service locator

In a Java environment, there is no "Web Reference" included in the project from a defined, fixed URL. Instead, the stub classes are simply added to your project. In order to link the web service to the actual URL, you must use the *web service locator* stub class:

```
CAdxWebServiceXmlCCServiceLocator serviceLocator;    // Service locator class
serviceLocator = new CAdxWebServiceXmlCCServiceLocator()

CAdxWebServiceXmlCCService x3Service;                // The web service instance

// Must give actual URL
serviceLocator.setCAdxWebServiceXmlCCEndpointAddress
    ("http://server/adxwsvc/services/CAdxWebServiceXmlCC");

x3Service = serviceLocator.getCAdxWebServiceXmlCC(); // This is the actual web service
```

This is done by:

- Creating an instance of CAdxWebServiceXmlCCServiceLocator
- Setting its URL endpoint to your web service server URL using setCAdxWebServiceXmlCCEndpointAddress()
- Getting the web service instance using getCAdxWebServiceXmlCC()



## 7.4 – Java Development

### The calling context

As with .NET, the first step in calling a web service is preparing the call context structure (See section 7.2):

```
// Calling context - class CAdxCallContext
CAdxCallContext cc = new CAdxCallContext(); // Instance of CAdxCallContext
cc.codeLang = "BRI";                        // Language code
cc.codeUser = "ADMIN";                      // X3 user
cc.password = "";                          // X3 password
cc.poolAlias = "WS_DEMOBRI";                // Pool name
cc.requestConfig = "adxwss.trace.on=on&adxwss.trace.size=16384
                  &adonix.trace.on=on&adonix.trace.level=3
                  &adonix.trace.size=8"; // Request configuration string
```





## 7.4 – Java Development

### Preparing the input XML string

Most of the X3 web service methods take an XML input stream as an argument (amongst others).

The XML input stream may be built as a simple string, or using available Java StringBuffer or XML utilities and builders. The **run** method accepts strings as the third argument.

The **String.format()** method (Same as in .NET) is also convenient for preparing short input XML strings.

```
// Prepare XML
StringBuffer xmlBuilder = new StringBuffer(1000);

xmlBuilder.append("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
xmlBuilder.append( "<PARAM><FLD NAME=\"ITM\">" );
xmlBuilder.append( "CD100" );
xmlBuilder.append( "</FLD></PARAM>" );

String xmlInput = xmlBuilder.toString(); // This is the input XML string
```



## 7.4 – Java Development

### Calling the **run** method (Subprograms)

The **run** method takes 3 arguments:

- The calling context (CAdxCallContext, see chapter 7.2 and previous sections in this chapter)
- The subprogram's public name
- The input XML string

```
// Run subprogram
// run method: run(CAadxCallContext <Context>, String <WebServiceName>, String <XML Input>)
// XML Input:
// XML header + <PARAM><FLD NAME="field1">value1</FLD><FLD NAME="field2">value2...</PARAM>
result = wsvc.run( cc, "GETITMPRI", xmlInput );
```



## 7.4 – Java Development

### Calling the **create** method (Objects)

Just like **run**, the **create** method takes 3 arguments:

- The calling context (CAdxCallContext, see chapter 7.2 and previous sections in this chapter)
- The subprogram's public name
- The input XML string

```
// Create BPR object
// Method: save(CAadxCallContext <context>, String <publicname>, String <xmlData>)
result = wsvc.save( cc, "WS_BPR", xmlData );
```



## 7.4 – Java Development

### Calling the **read** method (Objects)

**read** does not take any XML input stream. Instead, you must use an array of instances of the `CAdxParamKeyValue` class to indicate which record is to be read. Each instance contains one key element. For simple objects, there will be only one element in the array:

```
// Record key: An array of CAdxParamKeyValue
CAdxParamKeyValue[] objKey = new CAdxParamKeyValue[1]; // Only one element (Simple object)
objKey[0] = new CAdxParamKeyValue("BPRNUM", "C0001");

// Method: read(CAdxCallContext <context>, String <publicname>, CAdxParamKeyValue <key>)
result = wsvc.read(cc, "WS_BPR", objKey);
```

Notice the `CAdxParamKeyValue` element with index 1 has been built in one shot this time, using the `CAdxParamKeyValue` constructor with two arguments (The key field and its value). Also notice the field name has been used rather than its position in the index.

Both methods are valid in .NET and in Java.



## 7.4 – Java Development

### Calling the **modify** method (Objects)

**modify** is a combination of read and save. It takes both the `CAdxParamKeyValue` array to know which records to access, and an input XML stream to know which fields to modify:

```
// Record key: An array of CAdxParamKeyValue
CAdxParamKeyValue[] objKey = new CAdxParamKeyValue[1]; // Only one element (Simple object)
objKey[0] = new CAdxParamKeyValue();
objKey[0].key = "1"; // First element in the index
objKey[0].value = "C0001"; // Element value

// Input XML: Field to modify is CRY
String xmlHeader = new String("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
String xmlInput = new String("<PARAM><FLD NAME=\"CRY\">US</FLD></PARAM>");

// Method: modify(<context>, String <publicname>, CAdxParamKeyValue <key>, String <xmlData>)
result = wsvc.modify(cc, "WS_BPR", objKey, xmlHeader.concat(xmlInput));
```



## 7.4 – Java Development

### Calling the **query** method (Object list)

**query** returns data corresponding to the object's left browser columns. It also takes an array of `CAdxParamKeyValue` instances as an argument, but this time it represents quick select criteria for each of the left browser's columns.

```
// Read object
CAdxParamKeyValue[] objKey = new CAdxParamKeyValue[3];

objKey[0] = new CAdxParamKeyValue("1","C*"); // Searching for cust.codes that start with "C"
objKey[1] = new CAdxParamKeyValue("2","*");
objKey[2] = new CAdxParamKeyValue("3","*");

// Method: query(CAdxCallContext <context>, String <public name>,
               CAdxParamKeyValue[] <keys>, int <number of elements>)
result = wsvc.query( cc, "WS_BPR", objKey, 200 );
```



## 7.5 – PHP Development

### Definition

**PHP** (**PHP: Hypertext Pre-processor**) is a reflective programming language originally designed for producing dynamic Web pages. PHP is used mainly in server-side application software, but can be used from a command line interface or in standalone graphical applications.

In a web page context, PHP runs on the web server, taking PHP code as its input and creating web pages as output. The actual code may be embedded into html pages and vice-versa (PHP pages may include html elements).



## 7.5 – PHP Development

### Connecting to the Web Service

No class import or class generation is needed to connect to a web service in PHP. The PHP processor dynamically analyses the web service wsdl definition and creates needed structures.

The **SoapClient** structure is used to connect to the web service wsdl:

```
<?php  
  
// Get the SOAP structures from the Web Service  
$client = new SoapClient('http://localhost/adxwsvc/services/CAdxWebServiceXmlCC?wsdl');
```





## 7.5 – PHP Development

### The Calling Context

The next step in calling the web service is building the Calling Context structure. It is a string structure that must conform to the wsdl definition published for the web service:

```
// Build Calling Context array for connecting to the Web Service

$CContext["codeLang"] = "BRI";
$CContext["codeUser"] = "ADMIN";
$CContext["password"] = "";
$CContext["poolAlias"] = "WS_DEMOBRI";
$CContext["requestConfig"] = "adxwss.trace.on=on&adxwss.trace.size=16384
                             &adonix.trace.on=on&adonix.trace.level=3&adonix.trace.size=8";
```



## 7.5 – PHP Development

### Calling the **run** method (Subprograms)

Most of the code for running a subprogram will be dedicated to constructing the XML input:

```
// XML string to be passed to the web service
$xmlInput = <<<EOD
<PARAM>
  <FLD NAME="ITSSTR" >$itsstr</FLD>
  <FLD NAME="ITSEND" >$itsend</FLD>
  <FLD NAME="NBITS" >10</FLD>
</PARAM>
EOD;

// Run the subprogram using the SoapClient variable and Calling Context defined earlier
$result = $client->run($CContext,"GETITMPRI",$xmlInput);
```



## 7.5 – PHP Development

### Calling the **save** method (Objects)

As described in earlier sections, the **save** method takes the calling context, the public web service name and the XML input as arguments:

```
// XML string to be passed to the web service
$xmlInput = <<<EOD
<PARAM>
  <GRP ID="SOH0_1" >
    <FLD NAME="BPCORD" >$custid</FLD>
  </GRP>
  <TAB ID="SOH4_1" >
    <LIN>
      <FLD NAME="ITMREF" >$product</FLD>
      <FLD NAME="QTY" >$qty</FLD>
    </LIN>
  </TAB>
</PARAM>
EOD;

// Run save method
$result = $client->save($CContext, "SOH", $xmlInput);
```



## 7.5 – PHP Development

### The result XML stream

The result XML stream returned by the web service contains the usual structures, as described in the wsdl definition:

- `$result->resultXml`: The result XML data (String)
- `$result->status`: Return status (String)
- `$result->messages`: X3 messages (Array of *message* structure)
  - `$result->messages[i]->message`: X3 message (String)
  - `$result->messages[i]->type`: X3 message type (String)
- ...



## 7.5 – PHP Development

### Interpreting the results: An example

```
// store the returned XML string
$xml = simplexml_load_string($result->resultXml);
// Get status
$status = (int)$result->status;
if ($status == 1) {
    // loop through the XML and extract the Sales Order Number
    foreach ($xml->GRP->FLD as $name)
    {
        switch((string) $name['NAME'])
        {
            case 'SOHNUM':
                $order = $name;
                break;
        }
    }
}
} else {
    // Return array of messages
    $messages = $result->messages;

    // First message will be error message
    $message = $messages[0]->message;
    // Message type
    $type = $messages[0]->type;

    $parse_error = True;
    $err_text     = "ERROR: ($type) $message \n";
    $err_mesg     = "&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&1.&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&$err_text<br />";
    ...
}
```



## 7.6 – Debugging Web Services

### The Java debugger

Web service execution on the application side (X3 server) may be debugged using the Java Debugger, available for download from the X3 web server in the *Home / Tools* section.

If the debugger is opened, when a web service is run the system will halt in process AWEB (before execution), and at any `debug` instruction encountered in the X3 code.

The correct parameters must be set in the **requestConfig** property of the **CAdxCallContext** instance used by the web service method:

- Debugger on/off
- Debugger host (Debugger client machine name)
- Debugger port (1789 by default)

```
// Calling context - class CAdxCallContext
CAdxCallContext cc = new CAdxCallContext(); // Instance of CAdxCallContext

[...]

cc.requestConfig = "...&adonix.debug.on=on&adonix.debug.host=localhost&adonix.debug.port=1789";
```





## 8 – Best Practices & Performances

Contents

# 8.1 – Publication

## Entry Transaction

- It is highly recommended to customize the transaction used for publishing the WS.  
Reduce number of X3 data integrity controls (less time in Application server)  
Reduce number of publishable fields (65000 max)  
Simplify and reduce size of XML stream.





## 8.2 – Web Services Pool

### Pool Configuration

- Pool size and configuration adapted to your set of data  
Several Pools (Separate contexts: slow/fast requests, etc.)  
Several channels per pools.
- The web application will need to know the pool configuration to distribute requests into the matching pools.
- Switching language is not the most costly. Switching user is more costly
- Have enough channels per pools  
users\*language recommended  
1 channel per user at least  
OR  
5 channels minimum



## 8.3 – Miscellaneous

### Tools

- Know the tools delivered around Web services to enhance network traffic or XML management

Apache (GZIP compression : 5 MB XML stream → 50KB XML stream)



## 8.3 – Miscellaneous

### Analysis and customization of X3 code

- It may be possible that the standard X3 code is not optimized  
Use the X3 **TIMER**.



# REFERENCE

## AIRFRANCE

80000 users

30 orders per minutes, ~ 1 order every 2 seconds

300 WS calls per minute

40 licenses WS spread on 2 web servers

A load balancer working on the 2 web servers

~ 40 Web Services

