

Mustapha Stambouli University, Mascara
Department of Computer Science

Web Services

M2 RSD / M2 SITW

Dr. Rachid Khalladi
Associate Professor

Academic year 2025–2026

Table of Contents

Table of Contents 2
List of Figures 6
List of Tables 7
Foreword 8
Chapter 1 — Introduction to Web Services 9
1.1 History and Origin of Concepts 9
1.1.1 Early Remote Invocation Technologies 9
1.1.2 Limitations of Previous Approaches 9
1.1.3 The Emergence of Web Services Standards 10
1.2 Motivations 10
1.2.1 The Interoperability Problem 10
1.2.2 Technical Motivations 10
1.2.3 Economic Motivations 10
1.3 Formal Definition and Characteristics 11
1.3.1 The W3C Definition 11
1.3.2 Fundamental Characteristics 11
1.3.3 Positioning Relative to Modern APIs 11
Chapter 2 — Principles and Architecture of Web Services 13
2.1 General Organisation 13
2.1.1 The Triangular Model 13
2.1.2 The Three Fundamental Interactions 13
2.2 Architecture and Protocol Stack 13
2.2.1 The WS-* Protocol Stack 14
2.2.2 Complete Interaction Flow 14
2.2.3 Fundamental Characteristics 15
2.3 Types of Web Services and Selection Criteria 15
Chapter 3 — RPC (Remote Procedure Call) 17
3.1 Definition and Principles 17
3.1.1 Fundamental Concept 17
3.1.2 History of RPC 17
3.2 How RPC Works 17
3.2.1 Stub/Skeleton Architecture 18
3.2.2 The Eight Steps of RPC Execution 18
3.2.3 The XDR Format (eXternal Data Representation) 18
3.3 Interface — The IDL File (.x) 18

3.4 Server Process 19
3.4.1 calcul_server.c Skeleton 19
3.5 Client Process 20
3.5.1 calcul_client.c Skeleton 20
3.6 Conclusion: RPC vs SOAP Web Services 21
Chapter 4 — XML (eXtensible Markup Language) 23
4.1 Definition 23
4.2 XML's Predecessor on the Web: HTML 23
4.2.1 From HTML to XML 23
4.3 XML Writing Rules 24
4.4 What XML Makes Possible 24
4.5 XML Namespaces 25
4.6 Style Languages: CSS and XSL 25
4.7 XML Schema (XSD) 25
4.8 Linking and Addressing Languages 26
4.9 The Foreseeable Future of XML 26
Chapter 5 — SOAP (Simple Object Access Protocol) 27
5.1 Definition 27
5.2 Advantages 27
5.3 Procedure Calls 27
5.4 SOAP and XML 27
5.5 The SOAP Envelope 28
5.6 XML Representation 28
5.6.1 SOAP Request Message 28
5.6.2 SOAP Response Message 29
5.7 Data Model 29
5.7.1 Translating a Structure 29
5.7.2 Translating a List (or Array) 29
5.8 The RPC Model 30
5.9 Complete Example with Fault 30
Chapter 6 — UDDI (Universal Description, Discovery and Integration) 32
6.1 Definition 32
6.2 UDDI Registry Data 32
6.2.1 White Pages 32
6.2.2 Yellow Pages 33
6.2.3 Green Pages 33
6.3 Service Type Registration 33

6.4 UDDI and SOAP 33
6.4.1 API (SOAP Messages) 33
6.4.2 Interacting with XML 34
6.4.3 Implementation 34
6.5 Conclusion 34
Chapter 7 — WSDL (Web Services Description Language) 36
7.1 Definition 36
7.2 The Definitions 36
7.3 Namespaces 37
7.4 Complete WSDL Example 37
Chapter 8 — Web Services and the Enterprise 40
8.1 Definition 40
8.2 Building an IT Project 40
8.2.1 Introduction 40
8.2.2 Steps 40
8.3 EAI — Enterprise Application Integration 41
8.3.1 Definition 41
8.3.2 Application Integration 41
8.3.3 Functions of EAI 42
8.4 Business Processes 42
8.4.1 Definition 42
8.4.2 Dialogues 42
8.4.3 e-Business Processes 42
8.4.4 BPML — Business Process Modeling Language 42
8.5 Conclusion 43
Chapter 9 — Security and Reliability of Web Services 45
9.1 Security Mechanisms 45
9.1.1 Authentication 45
9.1.2 Access Authorisation 46
9.1.3 Data Encryption and Decryption Mechanism 46
9.1.4 Digital Signature 47
9.2 Securing Web Services 47
9.2.1 Securing the Infrastructure 47
9.2.2 Securing Connections 47
9.2.3 Authentication and Access Control 47
9.2.4 Securing SOAP 48
9.3 XML Security Standards 48

9.3.1 XML Signature 48
General Conclusion 50
Technology Summary 50
Perspectives 51
Appendix — Glossary 52
Bibliography 54

List of Figures

Figure 1 — General Web Services Architecture — The UDDI/WSDL/SOAP Triangle 14
Figure 2 — RPC Operation Principle — Stub/Skeleton and XDR Serialisation 17
Figure 3 — Comparative Summary — Traditional RPC vs SOAP Web Services 21
Figure 4 — The XML Ecosystem — Satellite Technologies and Associated Standards 24
Figure 5 — Detailed Structure of a SOAP Message — Envelope, Header, Body, Fault 28
Figure 6 — UDDI Registry Structure — White Pages, Yellow Pages and Green Pages 32
Figure 7 — Complete Structure of a WSDL Document — Abstract Part and Concrete Part 36
Figure 8 — EAI — Without Web Services (spaghetti architecture) vs With Web Services (hub-and-spoke) 41
Figure 9 — The Four Pillars of Web Services Security 45
Figure 10 — Comparative Summary — Traditional RPC vs SOAP Web Services (General Conclusion) 50

List of Tables

Table 1 — Chapter Prerequisites Overview (Foreword) 8
Table 2 — Limitations of Early Remote Invocation Technologies (Ch.1, S.1.1.2) 9
Table 3 — Positioning of Web Services Relative to Modern APIs (Ch.1, S.1.3.3) 11
Table 4 — The Triangular Model: Actors, Protocols and Examples (Ch.2, S.2.1.1) 13
Table 5 — The WS-* Protocol Stack (Ch.2, S.2.2.1) 14
Table 6 — Complete Interaction Flow (Ch.2, S.2.2.2) 14
Table 7 — Web Services Selection Criteria: SOAP/WS-* vs REST/JSON (Ch.2, S.2.3) 15
Table 8 — The XDR Data Format (Ch.3, S.3.2.3) 18
Table 9 — RPC vs SOAP Web Services — Comparative Summary (Ch.3, S.3.6) 21
Table 10 — HTML vs XML (Ch.4, S.4.2) 23
Table 11 — XML Style and Query Languages (Ch.4, S.4.6) 25
Table 12 — SOAP Binding Styles and Encoding Combinations (Ch.5, S.5.3) 27
Table 13 — Components of the SOAP Envelope (Ch.5, S.5.5) 28
Table 14 — UDDI SOAP API Operations (Ch.6, S.6.4.1) 33
Table 15 — WSDL Document Elements — Abstract and Concrete Parts (Ch.7, S.7.2) 36
Table 16 — Point-to-Point vs EAI Bus vs Web Services+UDDI Integration (Ch.8, S.8.3.2) 42
Table 17 — XML Security Standards (Ch.9, S.9.3.1) 48
Table 18 — Technology Summary (General Conclusion) 50
Table 19 — Glossary of Terms (Appendix) 52

Foreword

This Web Services course has been designed for second-year Master students in Computer Science (M2 RSD & M2 SITW). It covers the entire Web Services ecosystem in a progressive, structured manner, starting from conceptual foundations (RPC) through to advanced security and enterprise integration mechanisms.

Each chapter is organised around a clear pedagogical progression: formal definition, operating principles, annotated examples (code, diagrams, real-world cases), and review exercises. The examples are drawn from internationally recognised contexts to ground the concepts in professional reality.

Students who master this course will be able to: design a complete Web Services architecture, write a WSDL contract, secure a SOAP service with WS-Security, and integrate multiple services within a BPEL business process.

Chapter	Topic	Prerequisites
1	Introduction to Web Services	Networking basics and programming
2	Principles and Architecture	Chapter 1
3	RPC — Remote Procedure Call	C language, TCP/IP network basics
4	XML — eXtensible Markup Language	HTML, basic data structures
5	SOAP — Simple Object Access Protocol	XML (Chapter 4)
6	UDDI — Service Registry	SOAP (Chapter 5)
7	WSDL — Service Description	XML, SOAP (Chapters 4–5)
8	Web Services and the Enterprise	Chapters 1–7
9	Security and Reliability	Chapters 1–8, basic cryptography

Table 1 — Chapter Prerequisites Overview

Chapter 1 — Introduction to Web Services

1.1 History and Origin of Concepts

1.1.1 Early Remote Invocation Technologies

Web Services emerged from the need to enable heterogeneous applications to communicate over the Internet, regardless of their programming language, operating system, or architecture. To understand their emergence, it is useful to trace the evolution of distributed integration technologies.

As early as the 1980s, Sun Microsystems developed the RPC (Remote Procedure Call) model, which allows a program to execute a procedure on a remote machine as if it were local. This model constitutes the conceptual foundation of Web Services.

- 1984: Sun ONC-RPC — Remote Procedure Call for Unix/NFS
- 1991: CORBA (Common Object Request Broker Architecture) by OMG — multi-language distributed objects
- 1993: DCOM (Distributed Component Object Model) by Microsoft — distributed COM objects
- 1996: Java RMI (Remote Method Invocation) — invocation of remote Java objects
- 1998: XML 1.0 standardised by the W3C — universal exchange format
- 1999: SOAP 1.0 developed by Microsoft, UserLand, Ariba — first XML-based protocol
- 2001: WSDL 1.1 (W3C) and UDDI v2 — complete Web Services ecosystem
- 2003: SOAP 1.2 becomes a W3C Recommendation — stable reference version
- 2004: WS-Security, WS-ReliableMessaging, WS-Addressing — enterprise extensions
- 2007: JAX-WS 2.0 — Java standard API for SOAP Web Services

1.1.2 Limitations of Previous Approaches

Technology	Organisation	Interoperability	Firewall	Main Limitation
Sun ONC-RPC	Sun Microsystems	Unix/C only	Blocked	Single-platform, binary XDR format
Java RMI	SUN/Oracle	JVM only	Blocked	Java only, JVM required everywhere
CORBA/IIOP	OMG	Multi-language	Difficult	Costly ORB, complex configuration
DCOM	Microsoft	Windows only	Blocked	Microsoft proprietary, Windows only
Web Services	W3C/OASIS	Any language, any OS	HTTP 80/443	Open standard, universal adoption

Table 2 — Limitations of Early Remote Invocation Technologies

1.1.3 The Emergence of Web Services Standards

The convergence of three standard technologies made Web Services as we know them possible:

- HTTP: universal protocol crossing all firewalls via port 80 (HTTP) and 443 (HTTPS)
- XML: text-based exchange format, readable by any machine, extensible and self-describing
- Internet: a globally deployed infrastructure accessible from any point in the world

This convergence was formalised by the W3C and OASIS, which defined a coherent set of interoperable standards, often referred to collectively as WS-* (WS-Security, WS-ReliableMessaging, WS-Addressing, WS-Transaction, etc.).

1.2 Motivations

1.2.1 The Interoperability Problem

Organisations rely on diverse computing systems — IBM mainframes, Unix servers, Windows applications, SAP ERP, Oracle databases — that must communicate with each other. Before Web Services, each integration required a specific, costly and brittle development effort.

Example
A company has an SAP ERP (Java/Unix), a Siebel CRM (.NET/Windows), and a logistics application (COBOL/Mainframe). Without a common standard, integrating these three systems requires 3 proprietary connectors, that is N×(N-1) = 6 point-to-point interfaces — a complexity that grows exponentially with the number of systems.

1.2.2 Technical Motivations

- Reusability: expose existing business logic as a service accessible to all consumers
- Loose coupling: client and server evolve independently via the WSDL interface
- Interoperability: communication between Java and C#, Unix and Windows, without adapters
- Discoverability: a UDDI registry enables automatic discovery of available services
- Standardisation: open W3C protocols guaranteeing long-term technological stability
- Composability: assemble new services from existing building blocks (BPEL orchestration)

1.2.3 Economic Motivations

- Reduced integration costs: no more expensive point-to-point developments
- B2B e-commerce: organisations expose their systems to business partners
- Service outsourcing: delegate functionalities to specialised service providers
- Agility: rapid composition of new services from existing ones
- Reduced time-to-market: generate client code from WSDL in seconds
- Pooling: a single service can be consumed by dozens of applications simultaneously

1.3 Formal Definition and Characteristics

1.3.1 The W3C Definition

Definition
A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web Service in the manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialisation in conjunction with other Web-related standards. (W3C, 2004)

1.3.2 Fundamental Characteristics

- Self-describing: the service describes its own interface via WSDL
- Discoverable: can be registered in a UDDI directory for automatic search
- Interoperable: works across different languages, platforms and operating systems
- Loosely coupled: the client depends only on the interface, not the implementation
- Composable: services can be assembled into complex orchestrated processes
- Based on open standards: XML, SOAP, WSDL, UDDI — all standardised by W3C/OASIS

1.3.3 Positioning Relative to Modern APIs

Criterion	SOAP Web Services	REST/JSON	GraphQL	gRPC
Protocol	SOAP over HTTP/HTTPS	HTTP/HTTPS (verbs)	HTTP/HTTPS	HTTP/2 (binary)
Format	XML (verbose)	JSON (lightweight)	JSON	Protocol Buffers
Description	WSDL (formal)	OpenAPI/Swagger	GraphQL Schema	Proto files
Security	WS-Security (rich)	OAuth2/JWT	OAuth2/JWT	TLS + Auth
Transactions	WS-Transaction (ACID)	Not supported	Not supported	Not supported
Dominant use	Enterprise IS, B2B	Public web APIs	Data-oriented APIs	High-perf. microservices

Table 3 — Positioning of Web Services Relative to Modern APIs

Pedagogical Note
SOAP Web Services remain the inescapable standard in enterprise information systems (banking, insurance, government, healthcare) for their robustness, advanced features (transactions, reliability, security) and their ability to manage formal contracts between business partners.

Review Exercises

1. Why could RPC, DCOM and CORBA technologies not cross enterprise firewalls?

2. What are the three foundational technologies that made Web Services possible?
3. Distinguish the technical motivations from the economic motivations of Web Services.
4. What is the fundamental difference between a SOAP Web Service and a REST API?
5. In what enterprise context are SOAP Web Services still preferred today?

Additional Exercises

1. **Case study:** A hospital group operates an SAP HR system (Java), a legacy patient-billing system (COBOL/Mainframe) and a new Salesforce-based patient portal (cloud/REST). Propose a Web Services integration strategy, distinguishing point-to-point from hub-based integration, and justify your choice against the interoperability, security and cost criteria discussed in this chapter.

2. Compare, with concrete arguments, a scenario where a public weather-data API should be exposed as a SOAP Web Service versus one where a REST/JSON API is preferable.

Chapter 2 — Principles and Architecture of Web Services

2.1 General Organisation

2.1.1 The Triangular Model

Web Services are built on a triangular organisational model involving three distinct actors that interact via three standardised protocols. This organisation guarantees a clear separation of responsibilities and maximum flexibility.

Actor	Role	Protocol used	Concrete example
Service Provider	Develops, deploys and hosts the Web Service. Generates the WSDL description.	SOAP (service) + WSDL (description)	Amazon, Salesforce, PayPal
UDDI Registry	Stores descriptions of available services. Central point of discovery.	UDDI (publication and search)	UDDI Business Registry
Consumer	Client application that searches, discovers and invokes services.	UDDI (discovery) + SOAP (call)	Java, C#, PHP client application

Table 4 — The Triangular Model: Actors, Protocols and Examples

2.1.2 The Three Fundamental Interactions

- Interaction 1 — Publication: the provider generates a WSDL file describing its service and registers it in the UDDI registry with its metadata (name, description, category, URL)
- Interaction 2 — Discovery: the consumer queries the UDDI registry with search criteria (name, category, keyword) to find available services matching its needs
- Interaction 3 — Binding and Invocation: the consumer retrieves the WSDL of the selected service, generates a client stub, then invokes the service via SOAP/XML messages over HTTP

2.2 Architecture and Protocol Stack

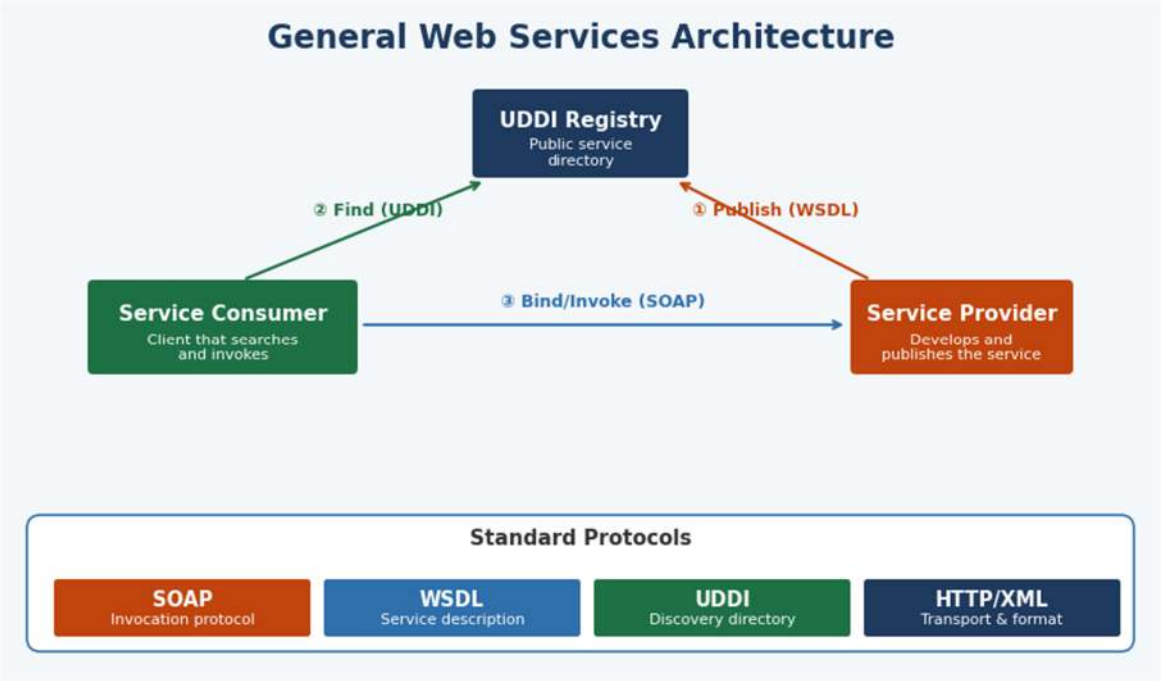


Figure1 — General Web Services Architecture — The UDDI/WSDL/SOAP Triangle

Source: author's elaboration, based on the W3C Web Services Architecture (2004)

2.2.1 The WS-* Protocol Stack

The Web Services architecture is built around several complementary protocol layers forming a coherent stack. Each layer adds functionality without modifying the layers below it.

Layer	Technology	Role	Standard
Business process	BPEL / BPMN	Service orchestration, long-running processes	OASIS WS-BPEL 2.0
Security	WS-Security, SAML	Authentication, authorisation, encryption, signature	OASIS WSS 1.1
Reliability	WS-ReliableMessaging	Guaranteed message delivery	OASIS WS-RM 1.2
Transactions	WS-AtomicTransaction	Distributed ACID transactions	OASIS WS-TX 1.2
Discovery	UDDI v3	Service registration and search	OASIS UDDI 3.0
Description	WSDL 1.1/2.0	Formal description of the service interface	W3C WSDL
Messaging	SOAP 1.1/1.2	Structured XML message exchange	W3C SOAP 1.2
Transport	HTTP/HTTPS, JMS	Message routing over the network	IETF RFC 7230
Format	XML 1.0	Universal data representation	W3C XML

Table 5 — The WS-* Protocol Stack

2.2.2 Complete Interaction Flow

Step	Actor	Action	Protocol
①	Client	Search: "weather service" in the registry	UDDI / SOAP

Step	Actor	Action	Protocol
②	UDDI Registry	Returns WSDL URLs of matching services	UDDI
③	Client	Downloads the WSDL file from the returned URL	HTTP GET
④	WS Server	Provides the WSDL file (contract description)	HTTP 200 + XML
⑤	Client	Generates the client stub from WSDL (wsimport, wsdl.exe)	Local tool
⑥	Client	SOAP call: HTTP POST with XML message in body	SOAP/HTTP
⑦	WS Server	Processes the request, executes the business logic	Internal
⑧	WS Server	Returns the SOAP XML response to the client	SOAP/HTTP 200

Table 6 — Complete Interaction Flow

2.2.3 Fundamental Characteristics

- Interoperability: any client can call any service via SOAP/HTTP
- Loose coupling: the client only knows the WSDL interface, not the implementation
- Self-description: the WSDL fully describes all parameters, types and behaviours
- Dynamic discovery: services can be found at runtime via UDDI
- Composability: multiple services can be combined to create complex processes
- Extensibility: SOAP headers allow adding security and transactions without modifying the body

2.3 Types of Web Services and Selection Criteria

Criterion	Choose SOAP/WS-*	Choose REST/JSON
Transactions	Distributed ACID transactions required	No cross-service transactions needed
Security	WS-Security, XML signatures, SAML tokens	OAuth2, JWT sufficient
Contract	Formal WSDL contract required (SLA, B2B)	OpenAPI/Swagger documentation
Reliability	Guaranteed delivery (WS-RM)	Best-effort HTTP sufficient
Performance	XML overhead acceptable (backend-to-backend)	Performance-critical (mobile, web)
Context	ERP, banking, insurance, healthcare, EDI	Public APIs, microservices, mobile

Table 7 — Web Services Selection Criteria: SOAP/WS-* vs REST/JSON

Review Exercises

1. Describe the three actors of the Web Services triangular model and their respective roles.
2. What are the complete steps of a Web Service invocation, from discovery to response?
3. Explain the difference between the Description layer (WSDL) and the Messaging layer (SOAP) in the protocol stack.
4. In what scenario would you prefer WS-ReliableMessaging over a simple synchronous HTTP call?

Additional Exercises

1. Draw and annotate the sequence of the three fundamental interactions (publish, find, bind/invoke) for a flight-booking web service consumed by a travel agency's client application, including the actors and protocols involved at each step.

2. A company needs guaranteed delivery of purchase-order messages between its ERP and its logistics provider, even during a several-hour partner outage. Which layer(s) of the WS-* protocol stack address this requirement, and why?

Chapter 3 — RPC (Remote Procedure Call)

3.1 Definition and Principles

3.1.1 Fundamental Concept

RPC (Remote Procedure Call) is a mechanism that allows a program to execute a procedure on a remote computer transparently, as if the procedure were local. The programmer writes ordinary function-call code; it is the RPC middleware that handles serialising the parameters, transmitting them over the network, executing the remote procedure, and returning the result.

Definition
Transparency principle: The fundamental objective of RPC is location transparency — the client code calls a remote function in exactly the same way as a local function. The complexity of network communication is entirely hidden by the RPC middleware.

3.1.2 History of RPC

- 1984: Sun ONC-RPC — used in NFS (Network File System)
- 1988: RFC 1057 — standardisation of SUN RPC
- 1990s: DCE-RPC (Microsoft) — foundation of DCOM and MSRPC
- 1997: XML-RPC — first XML-based RPC version (precursor to SOAP)
- 1999: SOAP 1.0 — evolution of XML-RPC with strong typing and namespaces

3.2 How RPC Works

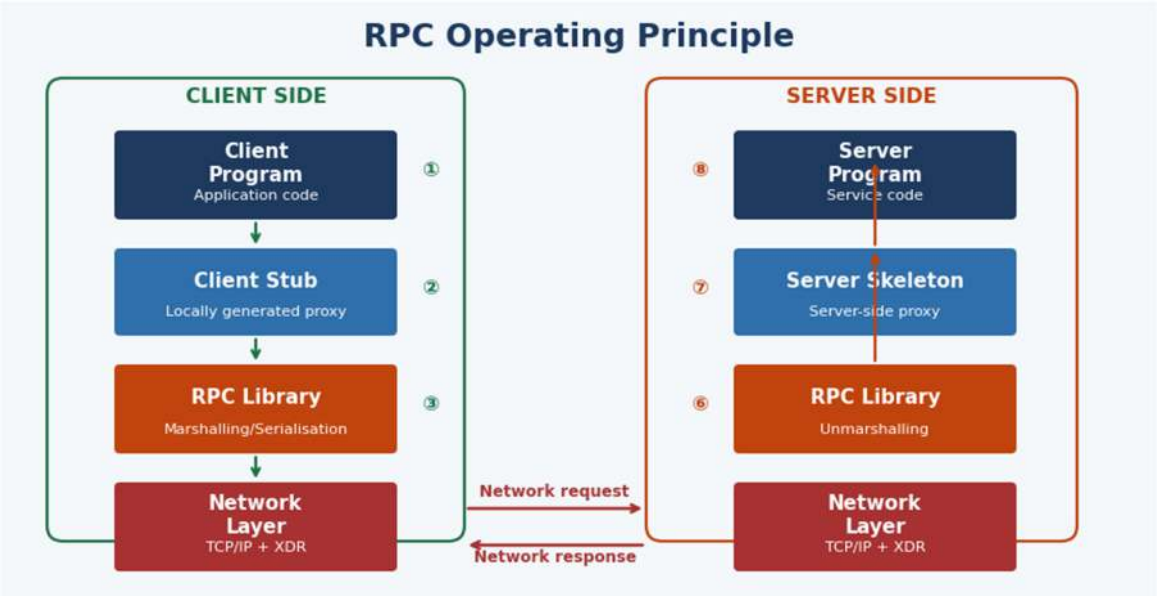


Figure2— RPC Operation Principle — Stub/Skeleton and XDR Serialisation
Source: author's elaboration, based on the Sun ONC-RPC model (RFC 1057)

3.2.1 Stub/Skeleton Architecture

The RPC mechanism relies on two components automatically generated from the IDL (Interface Definition Language) file:

- Client stub (proxy): intercepts local calls, serialises parameters to XDR format, sends over the network
- Server skeleton: receives network messages, deserialises parameters, calls the real procedure

Both components are generated by the rpcgen tool from the .x (IDL) file. The developer only implements the business logic on the server side.

3.2.2 The Eight Steps of RPC Execution

10. The client program calls the remote procedure as a local function
11. The client stub intercepts the call and marshals the parameters to XDR format
12. The RPC client library sends the network message to the server via TCP/UDP
13. Data travels across the network layer to the server (port assigned by portmapper)
14. The RPC server library receives the network message
15. The server skeleton unmarshals the parameters from XDR format
16. The server procedure is executed with the received parameters
17. The result is serialised and returned to the client via the reverse path

3.2.3 The XDR Format (eXternal Data Representation)

XDR Type	Size (bytes)	Representation	Example
int	4	Signed 32-bit big-endian integer	10 → 0x0000000A
float	4	IEEE 754 single precision	3.14f
double	8	IEEE 754 double precision	3.14159265
bool	4	FALSE=0, TRUE=1	true → 0x00000001
string	variable	Length (4 bytes) + data + padding	"hello" → 5+data+3 nuls
struct	sum of fields	Fields concatenated in order	struct {int, float}

Table 8 — The XDR Data Format

3.3 Interface — The IDL File (.x)

RPC communication begins with the definition of an interface in an IDL (Interface Definition Language) file with the .x extension. This file describes the available procedures, their parameters and their types, independently of the implementation language.

```
/* ── File: calcul.x (RPC IDL)  ──────────────────────────────────────── */
/* Defines the distributed computation service                            */
/* Programme number: 0x20000001, Version: 1                              */

struct complex_num {
```

```
    float real_part;
    float imaginary_part;
};

struct result {
    int   code;          /* 0 = success, -1 = error */
    float value;
    string message<256>;
};

program CALC_PROG {
    version CALC_VERS {
        result   ADD(float, float)         = 1;  /* a + b */
        result   SUBTRACT(float, float)    = 2;  /* a - b */
        result   MULTIPLY(float, float)    = 3;  /* a × b */
        result   DIVIDE(float, float)      = 4;  /* a / b */
        float    POWER(float, int)         = 5;  /* a^n  */
        complex_num SQRT_COMPLEX(float)    = 6;  /* sqrt */
    } = 1;
} = 0x20000001;

/* Generation: $ rpcgen calcul.x
 * Produces  : calcul.h, calcul_clnt.c, calcul_svc.c, calcul_xdr.c */
```

3.4 Server Process

3.4.1 calcul_server.c Skeleton

The server implements the procedures defined in the IDL. The skeleton file is generated automatically by rpcgen; the developer only implements the business logic:

```
/* ── calcul_server.c — Server-side implementation ── */
#include "calcul.h"
#include <stdio.h>
#include <math.h>

/* Procedure 1: Addition */
result *add_1_svc(float *a, float *b, struct svc_req *rqstp) {
    static result res;  /* IMPORTANT: static to survive after return */
    printf("[Server] Addition: %f + %f\n", *a, *b);
    res.code    = 0;
    res.value   = *a + *b;
    res.message = "OK";
    return &res;
}

/* Procedure 4: Division with error handling */
result *divide_1_svc(float *a, float *b, struct svc_req *rqstp) {
    static result res;
    if (*b == 0.0f) {
        res.code    = -1;
        res.value   = 0.0f;
        res.message = "Error: division by zero";
    } else {
        res.code    = 0;
        res.value   = *a / *b;
        res.message = "OK";
```

```
    }
    return &res;
}

/* Procedure 6: Complex square root */
complex_num *sqrt_complex_1_svc(float *n, struct svc_req *rqstp) {
    static complex_num c;
    if (*n >= 0) {
        c.real_part      = (float)sqrt((double)*n);
        c.imaginary_part = 0.0f;
    } else {
        c.real_part      = 0.0f;
        c.imaginary_part = (float)sqrt((double)(-(*n)));
    }
    return &c;
}
```

3.5 Client Process

3.5.1 calcul_client.c Skeleton

The client uses the stub generated by rpcgen to call remote procedures. Network management is completely transparent:

```
/* ── calcul_client.c — Client program ── */
#include "calcul.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    CLIENT *clnt;
    result  *res;
    float a = 10.5f, b = 3.2f;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s <server_host>\n", argv[0]);
        exit(1);
    }

    /* ①Connect to server via RPC (TCP transport) */
    clnt = clnt_create(argv[1], CALC_PROG, CALC_VERS, "tcp");
    if (clnt == NULL) { clnt_pcreateerror(argv[1]); exit(1); }

    /* ②Addition call — identical to a local call! */
    res = add_1(&a, &b, clnt);
    if (res && res->code == 0)
        printf("Addition: %.2f + %.2f = %.2f\n", a, b, res->value);

    /* ③Division by zero test */
    float zero = 0.0f;
    res = divide_1(&a, &zero, clnt);
    if (res && res->code != 0)
        printf("Division by 0: ERROR — %s\n", res->message);

    /* ④Square root of -16 */
    float neg = -16.0f;
```

```
    complex_num *c = sqrt_complex_1(&neg, clnt);
    if (c) printf("sqrt(-16): %.2f + %.2fi\n",
                  c->real_part, c->imaginary_part);

    clnt_destroy(clnt);
    return 0;
}

/* Build and run:
 * $ rpcgen calcul.x         # generate stubs
 * $ make -f Makefile.calcul # compile
 * $ ./calcul_server &       # start server
 * $ ./calcul_client localhost
 * Expected output:
 *   Addition: 10.50 + 3.20 = 13.70
 *   Division by 0: ERROR — Error: division by zero
 *   sqrt(-16): 0.00 + 4.00i */
```

3.6 Conclusion: RPC vs SOAP Web Services

Aspect	ONC RPC (Sun)	SOAP Web Services	Improvement
Data format	XDR (binary)	XML (text)	Human-readable, interoperable
Transport	UDP/TCP (fixed ports)	HTTP (port 80)	Crosses firewalls
Discovery	Local portmapper	UDDI (global registry)	Automatic discovery
Description	.x file (IDL)	WSDL (W3C XML)	Standardised, tool-agnostic
Security	AUTH_SYS (Unix UID)	WS-Security (rich)	PKI, tokens, XML signatures
Scope	Local network (LAN)	Global Internet	Worldwide scalability

Table 9 — RPC vs SOAP Web Services — Comparative Summary



Figure3 — Comparative Summary — Traditional RPC vs SOAP Web Services

Source: author's elaboration

Review Exercises

1. Explain the role of the client stub and the server skeleton in the RPC architecture.
2. Why do server-side RPC procedures use static variables for return values?
3. What is the role of the portmapper (rpcbind) in the ONC-RPC architecture?
4. Why can RPC not cross an enterprise firewall while SOAP Web Services can?
5. Write the IDL (.x) file for a banking service exposing: createAccount, deposit(iban, amount), withdraw(iban, amount) and getBalance(iban).

Additional Exercises

1. Extend the calcul.x IDL file (Section 3.3) with a new remote procedure MODULO(int, int) returning a result structure, and write the corresponding server-side skeleton function modulo_1_svc, handling division by zero.

2. Project: implement and test, using rpcgen, a simple remote "echo" service; measure and comment on the round-trip latency compared with a local function call, then discuss why this cost matters for choosing between RPC-style and document-style Web Services.

Chapter 4 — XML (eXtensible Markup Language)

4.1 Definition

XML (eXtensible Markup Language) is a markup meta-language developed by the W3C and standardised in 1998. Unlike HTML, XML does not define predefined tags: developers create their own semantic tags to describe the structure and content of their data.

Definition
XML is a simplified subset of SGML (Standard Generalized Markup Language) designed to be both easy for humans to read and easy to process automatically. It is based on Unicode text, which guarantees its universality. (W3C)

4.2 XML's Predecessor on the Web: HTML

HTML (HyperText Markup Language) is the founding language of the Web. To understand what XML adds, it is helpful to understand why HTML was no longer sufficient for distributed application needs.

Criterion	HTML	XML
Tags	Predefined (h1, p, table…)	Free, defined by the developer
Purpose	Visual presentation for browser	Data exchange and structured storage
Validation	Permissive (unclosed tags tolerated)	Strict: well-formed AND valid (XSD/DTD)
Semantics	None (structural/visual)	Full: each tag has a business meaning
Processing	Rendered by browser	Parsed by any application (Java, Python, C#…)
Extensibility	Not extensible	Extensible: new vocabularies via namespaces

Table 10 — HTML vs XML

4.2.1 From HTML to XML

XML resolves HTML's limitations by completely separating data from presentation, and by allowing the definition of custom semantic tags:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalogue xmlns:prod="http://example.com/product"
           xmlns:stock="http://example.com/stock">
  <product id="PRD-001">
    <reference>PRD-001</reference>
    <name>Dell XPS 15 Laptop Computer</name>
    <price currency="USD">1299.00</price>
    <vat rate="20">259.80</vat>
    <category code="COMPUT">Computing</category>
    <stock:availability>
      <stock:quantity>45</stock:quantity>
      <stock:unit>units</stock:unit>
    </stock:availability>
```

```
  </product>
</catalogue>
```

4.3 XML Writing Rules

XML imposes strict syntactic rules. A well-formed XML document must comply with:

- Mandatory XML declaration on the first line: <?xml version="1.0" encoding="UTF-8"?>
- A single root element enclosing the entire document
- Every opened tag must be closed: <name>…</name> or short form <name/>
- Tags are case-sensitive: <Name> ≠ <name>
- Attributes must always be in double quotes: <price currency="USD">
- Reserved characters must be encoded: < > & " '
- Comments are delimited by <!-- and -->

4.4 What XML Makes Possible

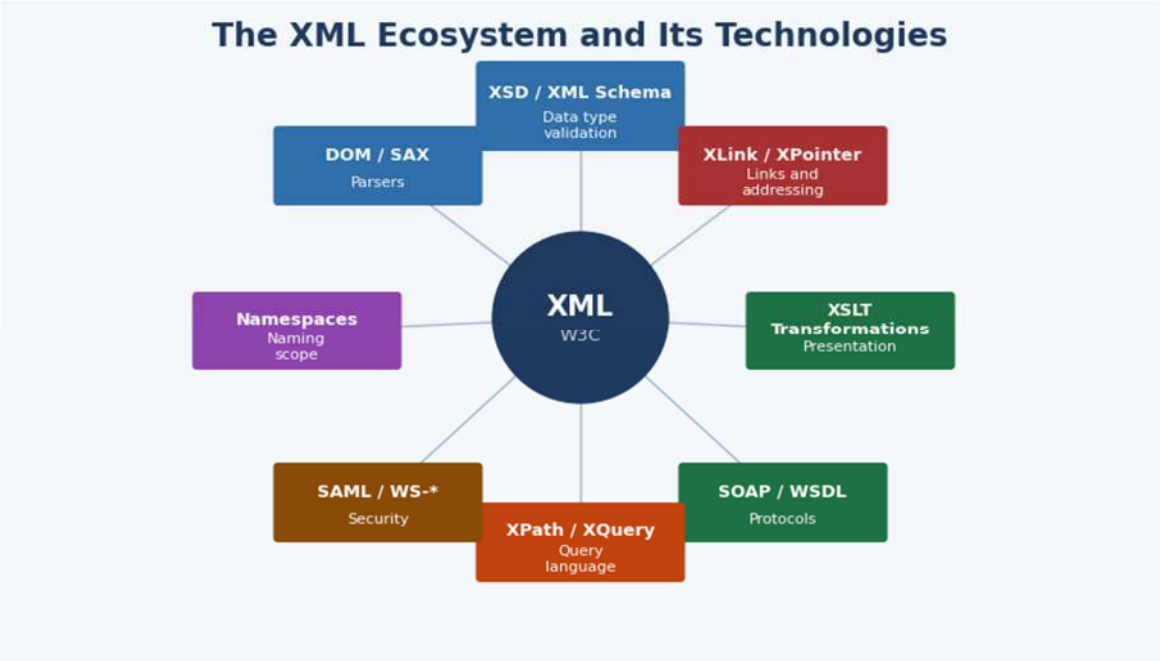


Figure4 — The XML Ecosystem — Satellite Technologies and Associated Standards
Source: author's elaboration, based on the W3C XML, XSD, XSLT, XPath and XLink specifications

- Separation of data and presentation (XSLT transforms the same XML into HTML, PDF, CSV)
- Structure validation via DTD or XML Schema (XSD)
- Navigation via XPath queries and XSLT transformations
- Interoperable exchange between heterogeneous applications
- Native namespace support for combining multiple vocabularies in one document

4.5 XML Namespaces

When multiple XML vocabularies are combined in one document (e.g. SOAP + WS-Security), naming conflicts can arise. Namespaces resolve this problem by prefixing each element with a unique URI.

```
<!-- Namespace: prefix:localName = URI:localName -->
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-wss"
    xmlns:ord="http://example.com/orders/v2">

  <soapenv:Header>
    <wsse:Security>           <!-- WS-Security namespace -->
      <wsse:UsernameToken>
        <wsse:Username>admin</wsse:Username>
      </wsse:UsernameToken>
    </wsse:Security>
  </soapenv:Header>

  <soapenv:Body>
    <ord:createOrder>         <!-- business namespace -->
      <ord:clientId>CLI-42</ord:clientId>
    </ord:createOrder>
  </soapenv:Body>

</soapenv:Envelope>
```

4.6 Style Languages: CSS and XSL

Language	Use	Example
CSS	Apply visual styling to an XML or HTML document	color: red; font-size: 12pt
XSL/XSLT	Transform an XML document into another format (HTML, PDF, CSV)	<xsl:for-each>, <xsl:value-of>
XPath	Navigate the XML tree to extract nodes	//price[@currency='USD']
XQuery	Complex queries on XML documents or databases	FLWOR expressions

Table 11 — XML Style and Query Languages

4.7 XML Schema (XSD)

XML Schema (XSD) allows the definition of the structure, types and constraints of an XML document. It is essential in Web Services to validate exchanged messages.

```
<!-- XML Schema (XSD): order document validation -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="order">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="client" type="xs:string"/>
        <xs:element name="total"  type="xs:decimal"/>
```

```
      </xs:sequence>
      <xs:attribute name="id"     type="xs:string" use="required"/>
      <xs:attribute name="status" type="statusEnum"/>
    </xs:complexType>
  </xs:element>

  <!-- Enumeration of valid statuses -->
  <xs:simpleType name="statusEnum">
    <xs:restriction base="xs:string">
      <xs:enumeration value="pending"/>
      <xs:enumeration value="confirmed"/>
      <xs:enumeration value="shipped"/>
      <xs:enumeration value="cancelled"/>
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

4.8 Linking and Addressing Languages

- XLink (XML Linking Language): simple and extended links between XML documents
- XPointer: addressing fragments within an XML document
- XPath: navigation and query language for traversing the XML tree
- XInclude: inclusion of external XML documents

4.9 The Foreseeable Future of XML

XML remains the central format for enterprise architectures. Although JSON has gained ground for lightweight web APIs, XML retains its dominance in domains requiring rigour and validation: SOAP Web Services, EDI, BPEL, SVG, XHTML, Office Open XML, and all mission-critical systems (banking, healthcare, government).

Review Exercises

1. What is the difference between a well-formed XML document and a valid XML document?
2. Why are XML namespaces indispensable in SOAP messages?
3. Write an XSD schema to validate an XML document representing a student (student ID, last name, first name, date of birth, list of modules with grade).
4. Write an XPath expression to extract the price in USD of all products whose stock quantity is greater than 10.

Additional Exercises

1. Write a complete, well-formed and namespace-qualified XML document representing a small library catalogue (at least two books, each with title, author, ISBN and availability), then write the accompanying XSD schema that validates it.

2. **Case study**: your organisation exchanges purchase orders with three different partners, each historically using a slightly different XML vocabulary. Explain how XML namespaces and XSLT can be combined to reconcile these vocabularies into one canonical format.

Chapter 5 — SOAP (Simple Object Access Protocol)

5.1 Definition

SOAP (Simple Object Access Protocol) is an XML-based messaging protocol for exchanging structured information in a distributed and decentralised environment. It defines the format of messages exchanged between Web Services, independently of the transport used (HTTP, JMS, SMTP, TCP).

Definition
SOAP is a lightweight protocol intended for exchanging structured information in a decentralised and distributed environment. It uses XML to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols. (W3C)

5.2 Advantages

- Transport independence: SOAP works over HTTP, SMTP, JMS, TCP
- Full interoperability: any language supporting XML can consume a SOAP service
- Extensibility: headers allow adding security, transactions, routing
- Strong typing: data types are defined by XML Schema (XSD)
- Transaction support: WS-AtomicTransaction for distributed ACID operations
- Advanced security: WS-Security for message-level signing, encryption and authentication

5.3 Procedure Calls

SOAP supports two binding styles (document and rpc) and two encoding modes (literal and encoded). The recommended combination for new implementations is document/literal.

Combination	Description	Interoperability	WS-I BP
rpc/encoded	RPC style, SOAP Section 5 encoding	Poor	Non-compliant
rpc/literal	RPC style, direct XSD typing	Good	Partial
document/encoded	XML document, SOAP encoding	Very poor	Non-compliant
document/literal	XML document, direct XSD typing	Excellent	Compliant — recommended

Table 12 — SOAP Binding Styles and Encoding Combinations

5.4 SOAP and XML

SOAP is entirely founded on XML. Every SOAP message is a well-formed XML document conforming to the SOAP envelope schema defined by the W3C. The data types exchanged are themselves defined by XML schemas (XSD) referenced in the associated WSDL.

5.5 The SOAP Envelope

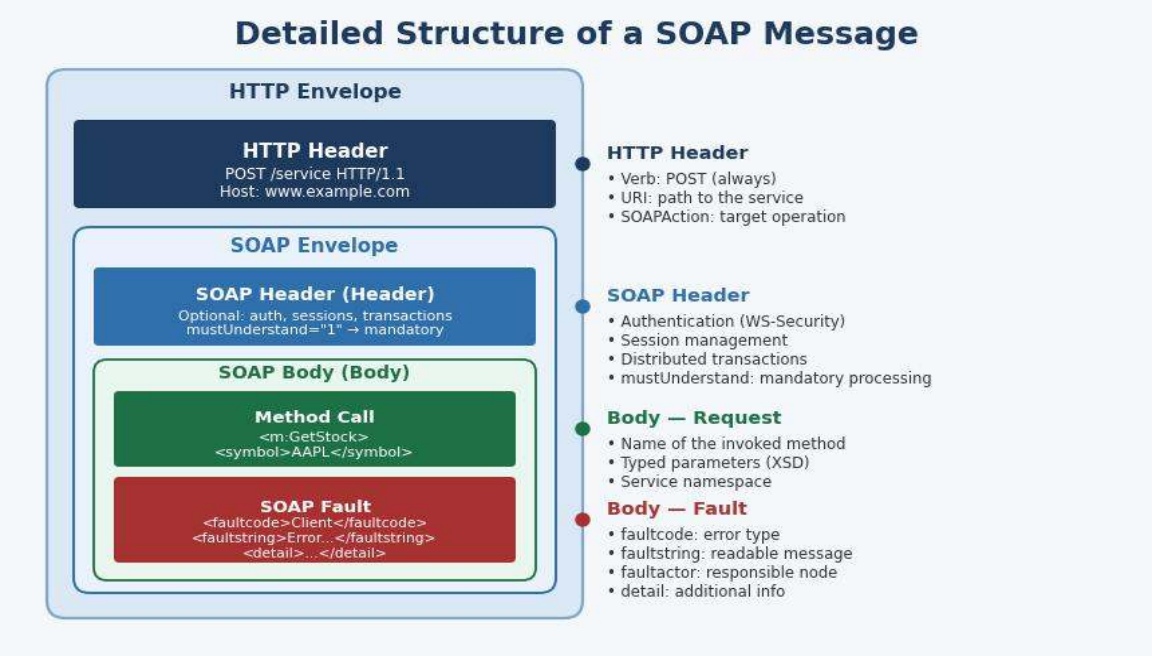


Figure 5— Detailed Structure of a SOAP Message — Envelope, Header, Body, Fault
Source: author's elaboration, based on the W3C SOAP Version 1.2 specification

The SOAP envelope is the container structure of any SOAP message. It is composed of:

Component	Mandatory	Role	Example content
<Envelope>	Yes	Root of the message, declares namespaces	SOAP namespaces, WS-Security…
<Header>	No	Metadata: security, routing, transactions	WS-Security, WS-Addressing
<Body>	Yes	Main content: RPC call or document data	Method call, response
<Fault>	No (errors)	Error reporting: code, message, details	Error code + description

Table 13 — Components of the SOAP Envelope

5.6 XML Representation

5.6.1 SOAP Request Message

```
<!-- SOAP 1.2 Message: getCotation call -->
POST /services/stockquote HTTP/1.1
Host: www.exchange.example.com
Content-Type: application/soap+xml; charset=UTF-8

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
    xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"
    xmlns:stock="http://exchange.example.com/services/stock">
```

```
  <soapenv:Header>
    <wsa:To>http://exchange.example.com/services/stockquote</wsa:To>
    <wsa:Action>getQuote</wsa:Action>
    <wsa:MessageID>urn:uuid:a1b2c3d4-e5f6-7890-abcd-ef1234567890</wsa:MessageID>
  </soapenv:Header>

  <soapenv:Body>
    <stock:getQuoteRequest>
      <stock:symbol>AAPL</stock:symbol>
      <stock:currency>USD</stock:currency>
    </stock:getQuoteRequest>
  </soapenv:Body>

</soapenv:Envelope>
```

5.6.2 SOAP Response Message

```
<!-- HTTP/SOAP Response -->
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=UTF-8

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"
                  xmlns:stock="http://exchange.example.com/services/stock">
  <soapenv:Body>
    <stock:getQuoteResponse>
      <stock:quote>
        <stock:symbol>AAPL</stock:symbol>
        <stock:price>182.50</stock:price>
        <stock:change>+1.75</stock:change>
        <stock:volume>54823100</stock:volume>
        <stock:timestamp>2025-04-06T10:30:00-04:00</stock:timestamp>
        <stock:currency>USD</stock:currency>
      </stock:quote>
    </stock:getQuoteResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

5.7 Data Model

5.7.1 Translating a Structure

Complex data structures are represented in XML according to their XSD definition in the WSDL. Each field of the structure becomes a child XML element.

```
<!-- Quote structure in XML (document/literal) -->
<stock:quote>
  <stock:symbol>AAPL</stock:symbol>        <!-- xsd:string  -->
  <stock:price>182.50</stock:price>         <!-- xsd:decimal -->
  <stock:change>+1.75</stock:change>        <!-- xsd:decimal -->
  <stock:volume>54823100</stock:volume>     <!-- xsd:long    -->
  <stock:currency>USD</stock:currency>      <!-- xsd:string  -->
</stock:quote>
```

5.7.2 Translating a List (or Array)

```
<!-- List of order lines in XML -->
<ord:lines>
  <ord:line num="1">
    <ord:reference>PRD-001</ord:reference>
    <ord:quantity>3</ord:quantity>
    <ord:price currency="USD">150.00</ord:price>
  </ord:line>
  <ord:line num="2">
    <ord:reference>PRD-002</ord:reference>
    <ord:quantity>1</ord:quantity>
    <ord:price currency="USD">85.00</ord:price>
  </ord:line>
</ord:lines>
```

5.8 The RPC Model

In the rpc/literal style, the SOAP Body directly encapsulates the method call with its parameters, as if calling a local function. The operation name corresponds to the name of the root element of the Body.

```
<!-- RPC call: getQuote(symbol, currency) -->
<soapenv:Body>
  <stock:getQuote>              <!-- operation name -->
    <stock:symbol>AAPL</stock:symbol>      <!-- parameter 1 -->
    <stock:currency>USD</stock:currency>  <!-- parameter 2 -->
  </stock:getQuote>
</soapenv:Body>
```

5.9 Complete Example with Fault

```
<!-- SOAP 1.2 Fault: symbol not found -->
HTTP/1.1 500 Internal Server Error
Content-Type: application/soap+xml

<soapenv:Envelope  xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
  <soapenv:Body>
    <soapenv:Fault>
      <soapenv:Code>
        <soapenv:Value>soapenv:Sender</soapenv:Value>
        <soapenv:Subcode>
          <soapenv:Value>stock:UnknownSymbol</soapenv:Value>
        </soapenv:Subcode>
      </soapenv:Code>
      <soapenv:Reason>
        <soapenv:Text xml:lang="en">Symbol UNKNOWN does not exist on this
exchange</soapenv:Text>
      </soapenv:Reason>
      <soapenv:Detail>
        <stock:error code="E-404">
          <stock:invalidSymbol>UNKNOWN</stock:invalidSymbol>
        </stock:error>
      </soapenv:Detail>
    </soapenv:Fault>
  </soapenv:Body>
</soapenv:Envelope>
```

Review Exercises

1. Identify the four components of a SOAP message and specify which ones are mandatory.
2. What is the difference between a Fault with Sender code and a Fault with Receiver code?
3. Explain why the document/literal combination is recommended by the WS-I Basic Profile.
4. Annotate the following Java class to make it a JAX-WS Web Service: convertCurrency(double amount, String from, String to) : double.

Additional Exercises

1. Rewrite the getQuote SOAP request of Section 5.6.1 in the rpc/literal style described in Section 5.8, and explain what changes and what stays the same compared with the document/literal version.

2. Design a complete SOAP Fault message (with faultcode, faultstring and a detail element) for a bank-transfer service that receives an amount exceeding the account balance, following the model in Section 5.9.

Chapter 6 — UDDI (Universal Description, Discovery and Integration)

6.1 Definition

UDDI (Universal Description, Discovery and Integration) is a registry directory specification enabling enterprises to publish descriptions of their Web Services and consumers to discover them. Developed by Ariba, IBM and Microsoft in 2000, standardised by OASIS through successive versions up to UDDI v3.

Definition
UDDI is the equivalent of a Yellow Pages directory for Web Services: organisations publish their services with categorisation metadata, and clients can search for them by name, business category or technical characteristics.

6.2 UDDI Registry Data

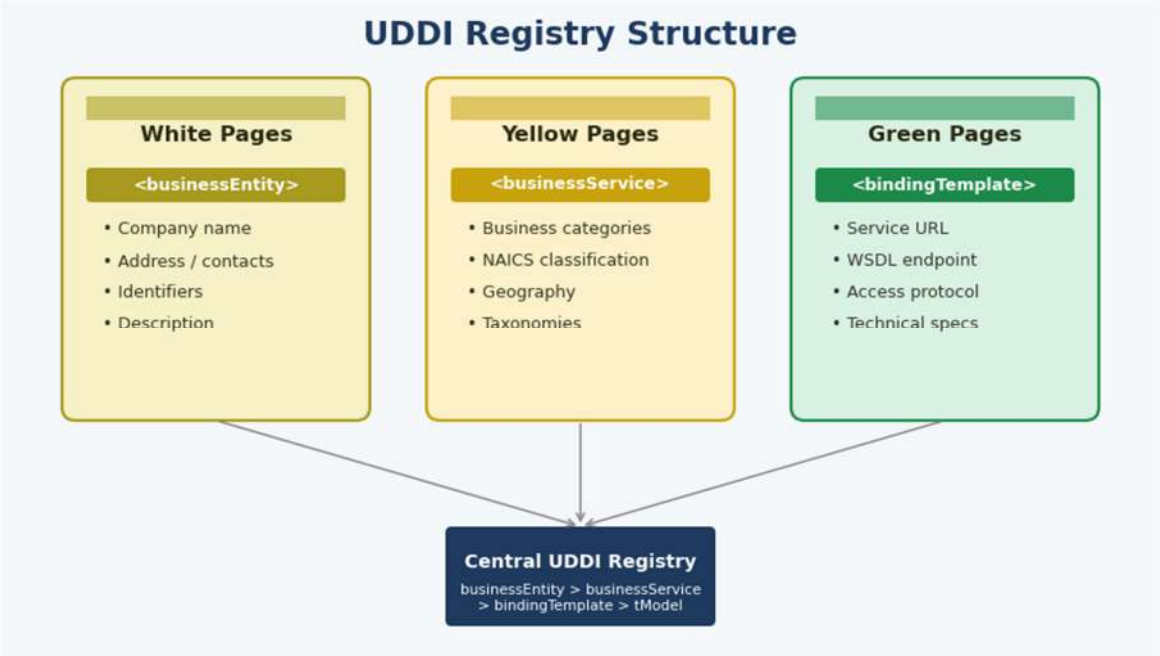


Figure6 — UDDI Registry Structure — White Pages, Yellow Pages and Green Pages
Source: author's elaboration, based on the OASIS UDDI Version 3.0.2 specification

6.2.1 White Pages

White Pages contain basic identification information for a business: name, description, contacts, and legal identifiers (DUNS, tax ID). They allow searching for a company by name or identification number.

```
<!-- businessEntity: White Pages -->
<businessEntity businessKey="uuid:A1B2C3D4-E5F6-7890-ABCD-EF1234567890">
  <name xml:lang="en">National Telecommunications Company</name>
```

```
  <description xml:lang="en">National telecom operator — B2B
services</description>
  <contacts>
    <contact useType="technical">
      <personName>Mr. John Smith — CIO</personName>
      <email>ws-team@telecom.example.com</email>
    </contact>
  </contacts>
  <identifierBag>
    <keyedReference keyName="DUNS" keyValue="123456789"/>
  </identifierBag>
</businessEntity>
```

6.2.2 Yellow Pages

Yellow Pages contain the categorisation information of a service: business sector (NAICS), geographic area (ISO-3166), service type. They allow finding services by field of activity, like a professional directory.

6.2.3 Green Pages

Green Pages contain the technical access information for the service: endpoint URL, WSDL file URL, supported transport protocol. These are the pieces of information a developer needs to invoke the service.

6.3 Service Type Registration

tModels (Technical Models) constitute the fourth key element of UDDI. They describe the technical specifications that a service complies with (e.g. WSDL 1.1, WS-I Basic Profile 1.0). They serve as fingerprints allowing services to be filtered by technical compatibility.

6.4 UDDI and SOAP

6.4.1 API (SOAP Messages)

UDDI exposes its own API as SOAP messages. Two main APIs are defined:

- Inquiry API: searching and consulting information (find_*, get_* operations)
- Publisher API: registering and updating information (save_*, delete_* operations)

UDDI Operation	Type	Description
find_business	Inquiry	Search for businesses by name, category, identifier
find_service	Inquiry	Search for services by name, category, tModel
find_binding	Inquiry	Search for the technical access points of a service
get_businessDetail	Inquiry	Retrieve complete details of a businessEntity
get_serviceDetail	Inquiry	Retrieve complete description of a businessService

UDDI Operation	Type	Description
save_business	Publisher	Register or update a business
save_service	Publisher	Register or update a service
delete_service	Publisher	Remove a service from the registry

Table 14 — UDDI SOAP API Operations

6.4.2 Interacting with XML

Here is an example UDDI request and response via SOAP to search for stock management services:

```
<!-- SOAP request: find_service by name -->
POST / HTTP/1.1
Host: www.uddi.org
Content-Type: text/xml; charset="utf-8"

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <find_service generic="2.0" maxRows="10"
                  xmlns="urn:uddi-org:api_v2">
      <name>Stock</name>
      <categoryBag>
        <keyedReference keyName="NAICS 2002" keyValue="541511"/>
      </categoryBag>
    </find_service>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

6.4.3 Implementation

Several UDDI implementations are available to deploy a private (intranet) or public registry:

- jUDDI (Apache): open-source Java implementation, based on JPA/Hibernate
- Microsoft UDDI Services: integrated into Windows Server, web management interface
- IBM WebSphere UDDI Registry: included in WebSphere Application Server
- SAP NetWeaver UDDI Registry: for the SAP Enterprise SOA ecosystem
- WSO2 API Manager: modern alternative with integrated developer portal

6.5 Conclusion

UDDI represents the discovery layer in the Web Services ecosystem. Although the global public UDDI Business Registry (UBR) was discontinued in 2005 by IBM and Microsoft due to lack of adoption, UDDI principles continue to be applied in private enterprise registries and SOA/ESB service catalogues. Modern architectures use solutions such as Apache ServiceMix, WSO2 Registry, or API catalogues like Apigee.

Review Exercises

1. Describe the four UDDI data structures and their correspondence with White Pages, Yellow Pages and Green Pages.
2. Which UDDI API would you use to: (a) publish a new service, (b) search for all services of a business, (c) retrieve the WSDL URL of a given service?
3. What were the reasons for the failure of the public global UDDI Business Registry (UBR) in 2005?
4. Compare UDDI v3 and WSO2 API Manager as a Web Services governance solution for an enterprise.

Additional Exercises

1. For a university exposing a "StudentRecordsService", specify realistic content for its White Pages, Yellow Pages and Green Pages entries in a UDDI registry.

2. Project: since the public UDDI Business Registry was discontinued in 2005 (Section 6.5), propose and justify a modern replacement architecture for service discovery inside a large enterprise (e.g. an API catalogue or private registry), listing the concrete tools you would use.

Chapter 7 — WSDL (Web Services Description Language)

7.1 Definition

WSDL (Web Services Description Language) is an XML language standardised by the W3C that formally and completely describes a Web Service. It acts as a contract between the service provider and consumer, precisely specifying: which operations the service offers, what parameters each operation accepts, what data types are used, and how to contact the service over the network.

Definition
WSDL is the equivalent of a Java interface or a C++ header file for Web Services. It describes WHAT the service can do and HOW to call it, without revealing the internal implementation. Once the WSDL is available, tools such as wsimport (Java), wsdl.exe (.NET) or soapUI can automatically generate the client code.

7.2 The Definitions

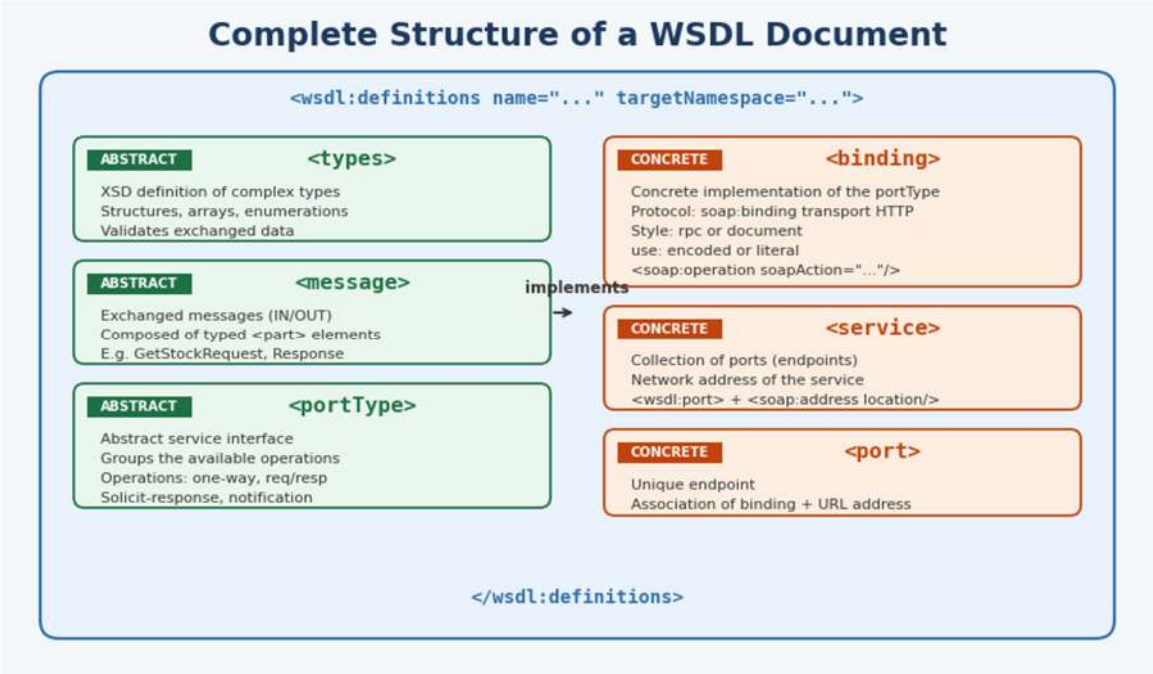


Figure7 — Complete Structure of a WSDL Document — Abstract Part and Concrete Part
Source: author's elaboration, based on the W3C WSDL 1.1 specification

A WSDL document is divided into two logical parts:

- Abstract part (interface): data types, messages, portType — describes what without how
- Concrete part (implementation): binding, service, port — describes how and where

WSDL Element	Part	Role	Java Analogy
<types>	Abstract	XSD definition of exchanged data types	Imported classes/types

WSDL Element	Part	Role	Java Analogy
<message>	Abstract	IN and OUT parameters of operations	Method parameters
<portType>	Abstract	Interface grouping available operations	Java interface
<operation>	Abstract	One operation with input/output/fault messages	Interface method
<binding>	Concrete	Protocol and style of the implementation	Class implementing the interface
<service>	Concrete	Collection of ports (endpoints)	Deployed server
<port>	Concrete	Network endpoint (URL + binding)	Deployment URL

Table 15 — WSDL Document Elements — Abstract and Concrete Parts

7.3 Namespaces

A WSDL document uses many XML namespaces to reference the various standards involved:

```
<wsdl:definitions
    name="StockQuoteService"
    targetNamespace="http://example.com/stockquote/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://example.com/stockquote/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
  <!-- WSDL body... -->
</wsdl:definitions>
```

7.4 Complete WSDL Example

Here is a complete and annotated WSDL document for a student management service:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
    name="StudentService"
    targetNamespace="http://university.example.com/services/students/"
    xmlns:tns="http://university.example.com/services/students/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">

  <!-- ①TYPES: XSD complex types -->
  <wsdl:types>
    <xsd:schema
targetNamespace="http://university.example.com/services/students/">
      <xsd:complexType name="Student">
        <xsd:sequence>
          <xsd:element name="studentId"     type="xsd:string"/>
          <xsd:element name="lastName"      type="xsd:string"/>
          <xsd:element name="firstName"     type="xsd:string"/>
          <xsd:element name="dateOfBirth"  type="xsd:date"/>
          <xsd:element name="major"         type="xsd:string"/>
```

```
        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>
  </wsdl:types>

  <!-- ②MESSAGES -->
  <wsdl:message name="GetStudentRequest">
    <wsdl:part name="studentId" type="xsd:string"/>
  </wsdl:message>
  <wsdl:message name="GetStudentResponse">
    <wsdl:part name="student" type="tns:Student"/>
  </wsdl:message>
  <wsdl:message name="ServiceFault">
    <wsdl:part name="code"    type="xsd:string"/>
    <wsdl:part name="message" type="xsd:string"/>
  </wsdl:message>

  <!-- ③PORT TYPE: Abstract interface -->
  <wsdl:portType name="StudentPortType">
    <wsdl:operation name="getStudent">
      <wsdl:input  message="tns:GetStudentRequest"  name="getStudentIn"/>
      <wsdl:output message="tns:GetStudentResponse" name="getStudentOut"/>
      <wsdl:fault  message="tns:ServiceFault"       name="getStudentFault"/>
    </wsdl:operation>
  </wsdl:portType>

  <!-- ④BINDING: SOAP 1.1 over HTTP, document/literal -->
  <wsdl:binding name="StudentSOAPBinding" type="tns:StudentPortType">
    <soap:binding                  transport="http://schemas.xmlsoap.org/soap/http"
style="document"/>
    <wsdl:operation name="getStudent">
      <soap:operation soapAction="getStudent" style="document"/>
      <wsdl:input>  <soap:body use="literal"/> </wsdl:input>
      <wsdl:output> <soap:body use="literal"/> </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>

  <!-- ⑤SERVICE: Deployment endpoint -->
  <wsdl:service name="StudentService">
    <wsdl:port name="StudentPort" binding="tns:StudentSOAPBinding">
      <soap:address location="https://si.university.example.com/ws/students"/>
    </wsdl:port>
  </wsdl:service>

</wsdl:definitions>
```

Review Exercises

1. Explain the difference between the abstract part and the concrete part of a WSDL document.
2. What is the role of the <binding> element in a WSDL document? How does it differ from <portType>?
3. Describe the four MEP (Message Exchange Patterns) supported by WSDL 1.1.
4. Write the WSDL for a weather service exposing getWeather(String city, String country) returning: temperature (float), description (String), humidity (int).

Additional Exercises

1. Complete the exercise begun in Section 7.4: extend the StudentService WSDL with a second operation, listStudentsByMajor(major: string): Student[], including its message, portType operation, binding and any necessary XSD complex types.

2. Explain, using the getWeather WSDL you wrote for Exercise 38, what would need to change in the abstract part versus the concrete part if the same service were later deployed at a second endpoint using SOAP over JMS instead of HTTP.

Chapter 8 — Web Services and the Enterprise

8.1 Definition

Web Services play a central role in modern enterprise information systems. They are the reference integration technology for interconnecting heterogeneous applications, automating exchanges between business partners and building service-oriented architectures (SOA — Service Oriented Architecture).

Definition
SOA — Service Oriented Architecture: Service-oriented architecture is an architectural paradigm in which the functionalities of an information system are broken down into reusable, loosely coupled services exposed via standardised interfaces. Web Services are the most widely deployed technical implementation of SOA.

8.2 Building an IT Project

8.2.1 Introduction

The adoption of Web Services profoundly transforms the way enterprise IT projects are designed and built. Rather than developing monolithic applications, teams build modular, reusable services.

8.2.2 Steps

- Analysis: identify the business functionalities to expose as services, define the contracts (WSDL)
- Design: SOA architecture, service identification, interface definition
- Development: implement the services, generate stubs from the WSDL
- Testing: unit tests of services, integration tests via soapUI or Postman
- Deployment: publish on the application server, UDDI registration
- Monitoring: performance monitoring, availability, SLA compliance

```
// Enterprise Web Service example: Order Management (JAX-WS)
@WebService(
    name            = "OrderService",
    serviceName     = "OrderService",
    targetNamespace = "http://order.enterprise.com/"
)
public class OrderServiceImpl {

    @WebMethod
    public OrderResult createOrder(
        @WebParam(name = "clientId") String clientId,
        @WebParam(name = "items")    List<OrderLine> items) {

        // ①Check stock via a SOAP call to the stock service
        for (OrderLine line : items) {
            boolean ok = stockService.checkAvailability(
```

```
                                line.getReference(), line.getQuantity());
            if (!ok)
                return new OrderResult(false, "Insufficient stock: " +
line.getReference());
        }
        // ②Create and persist the order
        Order order = new Order(clientId, items);
        String id = dao.save(order);
        // ③Deduct stock
        items.forEach(l -> stockService.deductStock(l.getReference(),
l.getQuantity()));
        return new OrderResult(true, id);
    }
}
```

8.3 EAI — Enterprise Application Integration

8.3.1 Definition

EAI (Enterprise Application Integration) refers to the set of methods, technologies and tools that enable disparate applications within the same organisation to exchange data and processes in a consistent manner.

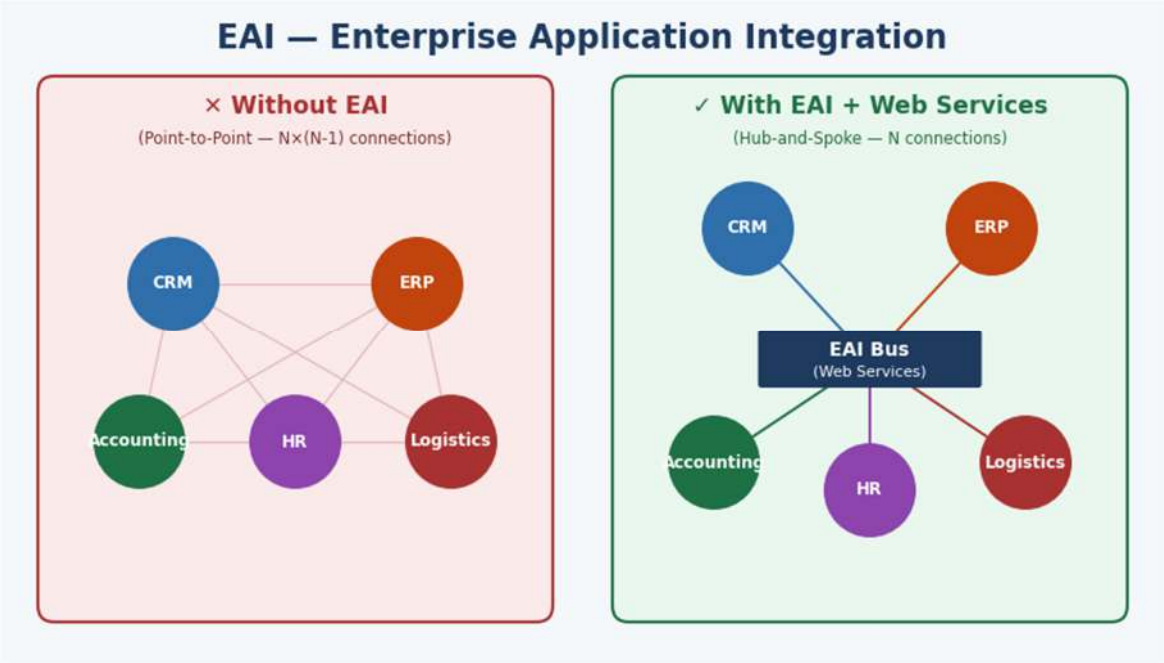


Figure 8— EAI — Without Web Services (spaghetti architecture) vs With Web Services (hub-and-spoke)
Source: author's elaboration

8.3.2 Application Integration

Before Web Services, application integration was done point-to-point: every pair of applications required a specific connector. For N applications this means N×(N-1)/2 connections — a complexity that grows quadratically.

Approach	Number of connections	Example (5 apps)	Maintainability
Point-to-Point	N×(N-1)/2	10 connections	Very difficult — 'spaghetti'
EAI Bus (Hub-Spoke)	N connections	5 connections	Simple — 1 central hub
Web Services + UDDI	N connections	5 connections	Simple + auto-discovery

Table 16 — Point-to-Point vs EAI Bus vs Web Services+UDDI Integration

8.3.3 Functions of EAI

- Transport and routing: route messages between applications
- Transformation: convert data formats between systems (XML↔EDI, JSON↔XML)
- Orchestration: coordinate execution of multi-application business processes
- Error management: automatic retry, dead-letter queues, alerts
- Monitoring: flow monitoring, integration dashboard
- Security: centralised authentication, encryption, access control

8.4 Business Processes

8.4.1 Definition

A business process is a structured sequence of activities aimed at producing a specific result for an organisation or customer. Web Services make it possible to automate and compose these processes through service orchestrations.

8.4.2 Dialogues

In the context of Web Services, business processes involve three types of dialogue between services:

- Synchronous dialogues: the client waits for the response before continuing (classic SOAP/HTTP call)
- Asynchronous dialogues: the client sends and continues without waiting (SOAP over JMS, callbacks)
- Long-running dialogues: processes that can last hours or days (compensation, rollback)

8.4.3 e-Business Processes

Web Services enable the automation of inter-enterprise B2B processes:

- Order-to-Cash: receive order → check stock → invoice → ship → receive payment
- Procure-to-Pay: purchase request → bidding → purchase order → receipt → payment
- Hire-to-Retire: recruitment → onboarding → HR management → training → departure
- Lead-to-Order: prospect → qualification → quote → negotiation → contract → order

8.4.4 BPML — Business Process Modeling Language

BPML (Business Process Modeling Language) and its successor BPEL (Business Process Execution Language) allow formal XML description of Web Services orchestrations:

```
<!-- BPEL excerpt: Order processing process -->
<process name="OrderProcessing"
         xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">

  <partnerLinks>
    <partnerLink name="StockService"    partnerLinkType="tns:StockLink"/>
    <partnerLink name="PaymentService" partnerLinkType="tns:PaymentLink"/>
  </partnerLinks>

  <sequence>
    <!-- ①Receive order -->
    <receive partnerLink="OrderService" operation="receiveOrder"
             variable="order" createInstance="yes"/>

    <!-- ②Check stock -->
    <invoke partnerLink="StockService" operation="checkStock"
            inputVariable="order" outputVariable="stockResult"/>

    <!-- ③Decision -->
    <if>
      <condition>$stockResult.available = true</condition>
      <sequence>
        <!-- ④Payment + notification in parallel -->
        <flow>
          <invoke partnerLink="PaymentService" operation="processPayment"
                  inputVariable="order" outputVariable="payResult"/>
          <invoke partnerLink="MailService" operation="sendConfirmation"
                  inputVariable="order"/>
        </flow>
      </sequence>
      <else>
        <invoke partnerLink="MailService" operation="sendOutOfStock"
                inputVariable="order"/>
      </else>
    </if>

    <!-- ⑤Final response -->
    <reply partnerLink="OrderService" operation="receiveOrder"
           variable="order"/>
  </sequence>

</process>
```

8.5 Conclusion

Web Services constitute the technical backbone of modern enterprise architectures. They make it possible to decouple information systems, automate cross-application and cross-enterprise business processes, and build agile, scalable architectures. EAI based on Web Services advantageously replaces costly and brittle point-to-point integrations.

Review Exercises

1. Distinguish orchestration from choreography in the context of business processes.
2. Calculate the number of point-to-point connections for an IS composed of 8 applications. How many remain with a central EAI bus?
3. What are the five main functions of an ESB? Give a concrete example for each function.
4. Propose an SOA architecture for a university information system: identify the services, their interfaces and the integration flows.

Additional Exercises

1. Model, as a BPEL-style sequence similar to Section 8.4.4, the "Procure-to-Pay" business process introduced in Section 8.4.3, identifying the partner links, the decision points and the parallel activities.

2. Case study: for the university information system proposed in Exercise 42, calculate the number of point-to-point connections needed for 8 core applications versus the number required with an EAI bus, and estimate the reduction in integration effort.

Chapter 9 — Security and Reliability of Web Services

9.1 Security Mechanisms

Securing Web Services is a critical challenge, particularly when they expose sensitive functionality over the Internet. Four fundamental mechanisms must be put in place:

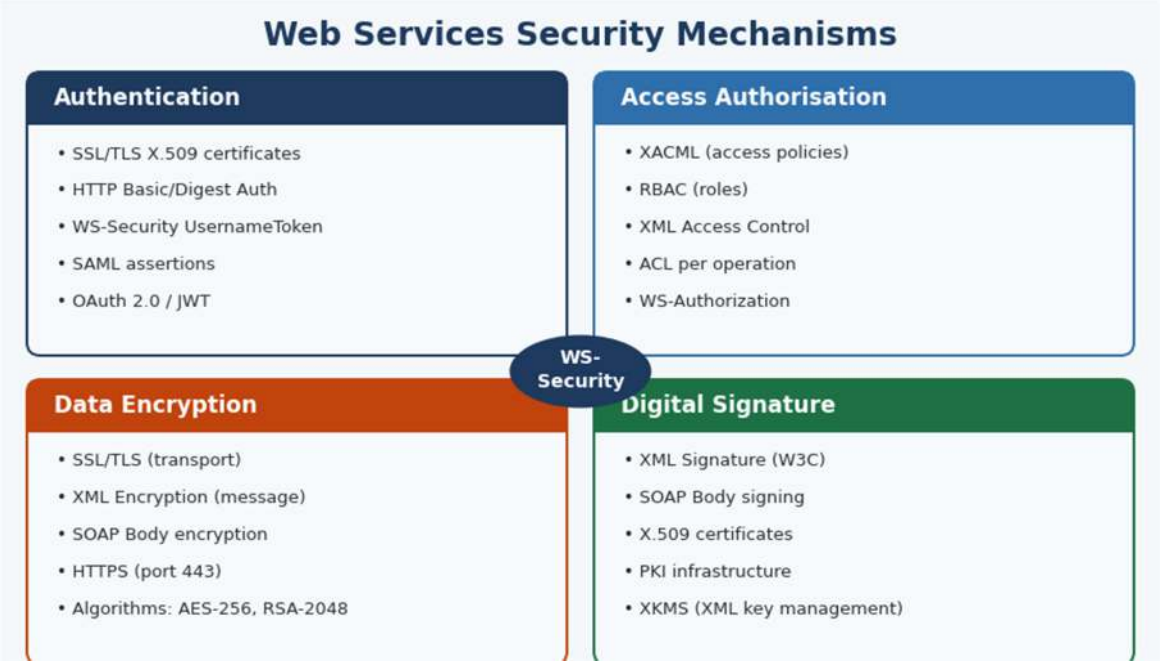


Figure9 — The Four Pillars of Web Services Security

Source: author's elaboration, based on the OASIS WS-Security 1.1 standard

9.1.1 Authentication

Authentication consists of verifying the identity of a service consumer before allowing access. Several mechanisms are available depending on the required security level:

- HTTP Basic Authentication: username/password encoded in Base64 — simple but must only be used over HTTPS
- HTTP Digest Authentication: challenge-response with MD5 hash — more secure than Basic
- SSL/TLS Mutual Authentication: X.509 certificate authentication on both client AND server
- WS-Security UsernameToken: credentials in the SOAP header — transport-independent
- SAML (Security Assertion Markup Language): XML assertions signed by an identity provider
- OAuth 2.0 + JWT: bearer tokens for REST APIs and microservice architectures

```
<!-- WS-Security: UsernameToken in the SOAP header -->
<SOAP-ENV:Header>
  <wsse:Security
      xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/..."
      SOAP-ENV:mustUnderstand="1">

    <wsse:UsernameToken  wsu:Id="token-001">
```

```
      <wsse:Username>john.doe@university.example.com</wsse:Username>
      <!-- Hash: Base64(SHA-1(nonce + created + password)) -->
      <wsse:Password
Type="...#PasswordDigest">6S3P2EWNP3lQf+9FkFyRfw==</wsse:Password>
      <wsse:Nonce>oG7iFKe5GHM+GVIYexEzUg==</wsse:Nonce>
      <wsu:Created>2025-04-06T10:00:00Z</wsu:Created>
    </wsse:UsernameToken>

    <!-- Validity window: message expires after 5 minutes -->
    <wsu:Timestamp>
      <wsu:Created>2025-04-06T10:00:00Z</wsu:Created>
      <wsu:Expires>2025-04-06T10:05:00Z</wsu:Expires>
    </wsu:Timestamp>

  </wsse:Security>
</SOAP-ENV:Header>
```

9.1.2 Access Authorisation

Authorisation determines what an authenticated user is allowed to do. Common mechanisms in Web Services include:

- RBAC (Role-Based Access Control): rights based on the user's roles (ADMIN, USER, GUEST)
- XACML (eXtensible Access Control Markup Language): XML language for granular access policies
- ACL at WSDL operation level: certain operations reserved for specific profiles only
- WS-Authorization: WS-Security extension for authorisation management

9.1.3 Data Encryption and Decryption Mechanism

Data encryption ensures the confidentiality of exchanges. Two levels can be applied:

- Transport-level encryption (SSL/TLS): encrypts the entire HTTP channel → HTTPS — simple and effective
- Message-level encryption (XML Encryption): selectively encrypts specific parts of the SOAP message

```
<!-- XML Encryption: SOAP Body encrypted with AES-256 -->
<!-- Before encryption: banking data in plaintext -->
<SOAP-ENV:Body>
  <m:BankTransaction>
    <cardNumber>4532-0156-7890-1234</cardNumber>
    <cvv>123</cvv>
    <amount>1500.00</amount>
  </m:BankTransaction>
</SOAP-ENV:Body>

<!-- After XML Encryption (AES-256-CBC) -->
<SOAP-ENV:Body>
  <xenc:EncryptedData
      xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
      Type="http://www.w3.org/2001/04/xmlenc#Element">
    <xenc:EncryptionMethod Algorithm="...aes256-cbc"/>
    <xenc:CipherData>
```

```
      <xenc:CipherValue>AGB1C2D3E4F5G6H7I8J9K0...</xenc:CipherValue>
    </xenc:CipherData>
  </xenc:EncryptedData>
</SOAP-ENV:Body>
```

9.1.4 Digital Signature

A digital signature guarantees the integrity and authenticity of a message: it proves that the message has not been altered in transit and that it genuinely comes from the claimed sender.

```
/* XML Signature process:
   1. Canonicalise the SOAP Body (C14N — invariant canonical form)
   2. Compute SHA-256 hash of the canonicalised Body
   3. Encrypt with the signer's RSA private key → Signature
   4. Include in the SOAP header with the X.509 certificate */

<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  <ds:SignedInfo>
    <ds:CanonicalizationMethod Algorithm="...exc-c14n#"/>
    <ds:SignatureMethod Algorithm="...rsa-sha256"/>
    <ds:Reference URI="#body">
      <ds:DigestMethod Algorithm="...sha-256"/>
      <ds:DigestValue>abc123XYZ+hash256==</ds:DigestValue>
    </ds:Reference>
  </ds:SignedInfo>
  <ds:SignatureValue>rXdG7Hk9P...rsaSignature==</ds:SignatureValue>
</ds:Signature>
```

9.2 Securing Web Services

9.2.1 Securing the Infrastructure

- DMZ (Demilitarised Zone): place Web Service servers in a separate network zone
- Application firewall (WAF): filter malicious HTTP requests (XML injection, DoS)
- Load balancer: distribute traffic and absorb peaks (DoS protection)
- Private network: isolate internal service communications from the public network

9.2.2 Securing Connections

- Mandatory HTTPS: TLS 1.2 minimum for all production communications
- X.509 certificates: use certificates issued by an enterprise PKI or a recognised CA
- Mutual TLS (mTLS): bidirectional certificate authentication on both client and server
- Certificate Pinning: enforce use of a specific certificate to prevent MITM attacks

9.2.3 Authentication and Access Control

- WS-Security UsernameToken with PasswordDigest + Timestamp for sensitive SOAP services
- Per-WSDL-operation access control (RBAC): only authorised operations are accessible
- Throttling: limit call rates per client (e.g. 100 req/min) to prevent abuse
- IP Whitelisting: restrict access to the authorised IP addresses of known partners

9.2.4 Securing SOAP

- Validate all inputs with a strict XSD schema (whitelist approach, not blacklist)
- Never expose exception stack traces in SOAP Faults in production
- Enable audit logging of all calls (who, when, which operation, from where)
- Use WS-Addressing for message correlation and non-repudiation

9.3 XML Security Standards

9.3.1 XML Signature

XML Signature (XMLDSig) is a W3C standard for the digital signing of XML elements or entire documents. It guarantees integrity (the content has not been modified) and authenticity (the sender is genuinely who they claim to be).

Standard	Organisation	Main use
XML Signature (XMLDSig)	W3C	Digital signature of XML documents and messages
XML Encryption (XMLEnc)	W3C	Selective encryption of XML elements
XKMS	W3C	Public key management via Web Services
SAML	OASIS	Federated authentication and authorisation (SSO)
XACML	OASIS	Granular access control policies in XML
WS-Security	OASIS	SOAP message security framework
WS-Trust	OASIS	Security token issuance and validation
WS-Federation	OASIS	Identity federation between partner organisations

Table 17 — XML Security Standards

Pedagogical Note
XKMS allows delegation of PKI (Public Key Infrastructure) management to Web Services. Instead of handling X.509 certificates directly, applications query an XKMS service that manages public keys, their validation and their revocation. This considerably simplifies the implementation of cryptography in Web Services.

Review Exercises

1. What are the six main security threats facing Web Services? For each one, specify the appropriate WS-* countermeasure.
2. Explain the difference between transport-level encryption (TLS) and message-level encryption (XML Encryption). In what scenario is XML Encryption indispensable?
3. Describe the complete process of XML-signing a SOAP message: what steps are needed and what guarantees does this signature provide?

4.. Design the complete security policy for a Web Service accessing medical records in a hospital: authentication, authorisation, confidentiality, integrity and audit.

Additional Exercises

1. For the medical-records Web Service of Exercise 46, write the WS-Security UsernameToken header (following the model in Section 9.1.1) that would authenticate a specific physician, including a Nonce and a Timestamp with a five-minute validity window.

2. Compare, for a mobile banking application calling a Web Service, the trade-offs between transport-level encryption (HTTPS only) and message-level encryption (XML Encryption on top of HTTPS), and state which combination you would recommend and why.

General Conclusion

This course has presented the entire Web Services ecosystem, from historical foundations (RPC, XML) through to advanced security mechanisms, covering the central protocols SOAP, UDDI and WSDL, and their enterprise applications.



Comparison: Traditional RPC vs Web Services

Criterion	Traditional RPC	SOAP Web Services
Protocol	Proprietary / UDP	Standard HTTP (port 80/443)
Format	Binary (XDR)	Readable XML text
Firewall	Blocked by default	Traverses (HTTP)
Interoperability	Same language required	Multi-language, multi-platform
Typing	Native C types	XSD types (int, string, float...)
Transport	TCP/UDP	HTTP, HTTPS, SMTP
Standard	SUN ONC-RPC / DCOM	W3C (SOAP, WSDL) + OASIS (UDDI)
Discovery	Manual (portmapper)	Automatic (UDDI)
Coupling	Tight (shared stubs)	Loose (WSDL interface)

Figure10 — Comparative Summary — Traditional RPC vs SOAP Web Services

Source: author's elaboration (see Chapter 3, Section 3.6)

Technology Summary

Technology	Role in the ecosystem	Current status
XML 1.0	Universal exchange format — basis of all WS-* standards	Foundational standard — ubiquitous
SOAP 1.2	Structured messaging protocol between services	Stable — dominant in enterprise IS
WSDL 1.1	Formal service contract — machine-readable description	De facto standard — still dominant
UDDI v3	Service discovery registry	Replaced by API portals in practice
WS-Security 1.1	SOAP message security framework	Enterprise standard — banking, healthcare
WS-BPEL 2.0	Business process orchestration across services	Deployed in ESB and SOA platforms
REST/JSON	Public web APIs and microservices	Dominant for modern web APIs
gRPC / Protocol Buffers	High-performance inter-microservice communication	Strong growth — Google, Netflix, CNCF

Table 18 — Technology Summary

Perspectives

- SOAP Web Services remain the backbone of enterprise IS (banking, insurance, healthcare, government) for their robustness and advanced features
- REST/JSON has displaced SOAP for public web APIs and microservice architectures
- GraphQL provides additional flexibility for data-oriented APIs with variable granularity
- gRPC (Google, 2016) offers superior performance for inter-microservice communication via binary Protocol Buffers
- The foundational principles of Web Services (contract, discovery, interoperability, composability) persist in cloud-native architectures (API Gateway, Service Mesh, Istio)

Appendix — Glossary

Term	Definition
API	Application Programming Interface — programming interface exposing functionalities accessible from outside
B2B	Business to Business — automated electronic exchanges between partner companies
BPEL	Business Process Execution Language — XML language for describing business processes as service orchestrations
CGI	Common Gateway Interface — interface allowing web servers to execute dynamic scripts
CORBA	Common Object Request Broker Architecture — OMG multi-language distributed object middleware
DCOM	Distributed Component Object Model — Microsoft extension of COM for distributed Windows objects
DMZ	Demilitarised Zone — buffer network zone between the internal network and the Internet
EAI	Enterprise Application Integration — integration of heterogeneous enterprise applications via a common bus
HTTP	HyperText Transfer Protocol — Web transfer protocol; secure version HTTPS on port 443
IDL	Interface Definition Language — language-independent interface description language
JAX-WS	Java API for XML-Based Web Services — Java standard API for developing and consuming SOAP Web Services
JMS	Java Message Service — asynchronous Java messaging API
ORB	Object Request Broker — CORBA component managing communication between distributed objects
PKI	Public Key Infrastructure — infrastructure for managing certificates and asymmetric keys
RMI	Remote Method Invocation — Java mechanism for invoking remote object methods
RPC	Remote Procedure Call — mechanism for calling a procedure on a remote machine transparently
SAML	Security Assertion Markup Language — XML standard for exchanging authentication assertions (SSO)
SOA	Service Oriented Architecture — enterprise architecture organised as reusable, loosely coupled services
SOAP	Simple Object Access Protocol — XML message exchange protocol for Web Services
SSL/TLS	Secure Sockets Layer / Transport Layer Security — network communications encryption protocol
tModel	Technical Model — technical specification model in UDDI
UDDI	Universal Description Discovery and Integration — universal Web Services discovery registry
URI	Uniform Resource Identifier — universal resource identifier on the Internet
W3C	World Wide Web Consortium — international standardisation body for Web technologies

Term	Definition
WSDL	Web Services Description Language — XML language for formally describing a Web Service
WSS	WS-Security — OASIS security standard for SOAP messages
XDR	External Data Representation — portable binary serialisation format used by Sun RPC
XKMS	XML Key Management Specifications — XML protocol for managing cryptographic keys via Web Services
XML	eXtensible Markup Language — markup meta-language standardised by the W3C in 1998
XSD	XML Schema Definition — language for defining and validating the structure of XML documents

Table 19 — Glossary of Terms

Appendix — Practical Work: Web Services with Python

Practical Work — Web Services

Four progressive worksheets

SOAP • WSDL • REST / WADL • Security and Composition

Teaching support — Python

Academic year 2026–2027

Practical Work Sheet No. 1

Title: Development and testing of a SOAP Web Service with Python

1. Objectives

- Understand the principle of a SOAP Web Service and the role of the HTTP protocol.
- Develop a SOAP service in Python using the Spyne library.
- Publish a local service and access its generated WSDL description.
- Test the operations using SoapUI.

2. Work Environment

- Python 3.10 or later.
- Spyne to implement the SOAP service.
- SoapUI to send SOAP requests.
- A web browser to view the WSDL.

3. Installation

```
python --version
python -m pip install spyne
```

Create a folder named TP1_SOAP. In this folder, create the file service.py.

4. Development of the DeliveryService Service

The service exposes three operations: calculate the cost of a delivery, convert an amount into dinars, and check the availability of a delivery.

```
from spyne import Application, rpc, ServiceBase, Float, String, Boolean
from spyne.protocol.soap import Soap11
from spyne.server.wsgi import WsgiApplication
from wsgiref.simple_server import make_server


class DeliveryService(ServiceBase):
    @rpc(Float, _returns=Float)
    def calculate_cost(ctx, weight):
        if weight <= 0:
            return 0.0
        if weight <= 2:
            return 500.0
        if weight <= 5:
            return 800.0
        return 1200.0

    @rpc(Float, _returns=Float)
    def convert_euro_to_dinar(ctx, amount):
        rate = 145.0
        return amount * rate

    @rpc(String, _returns=Boolean)
    def check_availability(ctx, city):
        cities = ["Oran", "Algiers", "Mascara"]
        return city in cities


application = Application(
    [DeliveryService],
    tns="services.delivery",
    in_protocol=Soap11(validator="lxml"),
    out_protocol=Soap11()
)
wsgi_app = WsgiApplication(application)

if __name__ == "__main__":
```

```
server = make_server("127.0.0.1", 8000, wsgi_app)
print("SOAP service available at http://127.0.0.1:8000")
server.serve_forever()
```

5. Execution and Verification

1. Open a terminal in the project folder.
2. Start the server with the command python service.py.
3. Open the address http://127.0.0.1:8000/?wsdl in a browser.
4. Check that the XML document describing the service is displayed.

Port 8000 is a development choice. You may replace it with another free port, provided you use the same address in the testing tools

.

6. Tests with SoapUI

1. Launch SoapUI and create a SOAP project.
2. Enter the WSDL address: http://127.0.0.1:8000/?wsdl.
3. Expand the service tree and open each operation.
4. Send the test values below and record the responses.

Operation	Input	Expected Result
calculate_cost	weight = 1.5	500
calculate_cost	weight = 4	800
calculate_cost	weight = 8	1200
convert_euro_to_dinar	amount = 10	1450
check_availability	city = Oran	true
check_availability	city = Tlemcen	false

7. Required Work

- Add an operation estimate_delivery_time that receives a city and returns an estimated delay in days.
- Modify the conversion rate and test the service again.
- Test a negative value for the weight. Describe the observed behaviour.
- Present a screenshot of the WSDL and two SOAP exchanges.

8. Questions

1. What is the role of SOAP in a Web Services architecture?
2. What is the difference between a SOAP request and a SOAP response?
3. Why does the service expose a WSDL document?
4. Is port 8000 imposed by SOAP? Justify your answer.

9. Deliverables and Evaluation

To submit: the service.py file, the SoapUI screenshots, the answers to the questions and a short report.

Criterion	Points
Installation and execution	3
Implementation of the operations	7
SoapUI tests	5
Analysis and report	5
Total	20

Practical Work Sheet No. 2

Title: Description, manipulation and creation of a WSDL contract

1. Objectives

- Identify the role of WSDL in describing a SOAP service.
- Analyse the main structures of a WSDL 1.1 document.
- Modify a WSDL contract and distinguish description from implementation.
- Create a simple contract and import it into SoapUI.

2. Prerequisites and Environment

Have completed TP 1 and have the DeliveryService service running. Tools: browser, text editor or VS Code, SoapUI.

3. Part A — Retrieving the Generated WSDL

1. Start the TP 1 server with python service.py.
2. Access http://127.0.0.1:8000/?wsdl.
3. Save the document under the name DeliveryService_Generated.wsdl.
4. Open the file in an XML editor.

4. Part B — Analysis of the WSDL Document

Identify the following elements and specify their function.

Element	Role to identify
definitions	Root element and namespaces of the contract.
types	Data types used by the messages.
message	Data exchanged as input and output.
portType	Abstract operations offered by the service.
binding	Protocol and communication format associated with the service.
service	Concrete access point to the service.

Complete a table that associates each DeliveryService operation with its input and output messages.

5. Part C — Manipulating the Generated WSDL

- Make a copy of the original file under the name DeliveryService_Modified.wsdl.
- Change the service name in the copy.
- Locate the address defined in soap:address and propose another test address.
- Identify the types associated with the parameters and results.
- Check that modifying the local file does not change the server's Python code.

To remember: modifying a copy of the WSDL changes the local contract. It does not automatically change the Spyne implementation or the service that is already running.

6. Part D — Manual Creation of a WSDL 1.1 Document

Create the file DeliveryCalculation.wsdl. It describes an operation calculateDelivery that receives a weight of type xsd:float and returns a result of type xsd:float.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
    xmlns="http://schemas.xmlsoap.org/wsdl/"
```

```
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://example.org/delivery"
    targetNamespace="http://example.org/delivery"
    name="DeliveryCalculation">
    <types>
        <xsd:schema targetNamespace="http://example.org/delivery">
            <xsd:element name="weight" type="xsd:float"/>
            <xsd:element name="calculateDeliveryResponse">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="result" type="xsd:float"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:schema>
    </types>
    <message name="calculateDeliveryInput">
        <part name="weight" element="tns:weight"/>
    </message>
    <message name="calculateDeliveryOutput">
        <part name="result" element="tns:calculateDeliveryResponse"/>
    </message>
    <portType name="DeliveryCalculationPortType">
        <operation name="calculateDelivery">
            <input message="tns:calculateDeliveryInput"/>
            <output message="tns:calculateDeliveryOutput"/>
        </operation>
    </portType>
    <binding name="DeliveryCalculationBinding"
             type="tns:DeliveryCalculationPortType">
        <soap:binding style="document"
                      transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="calculateDelivery">
            <soap:operation soapAction="calculateDelivery"/>
            <input><soap:body use="literal"/></input>
            <output><soap:body use="literal"/></output>
        </operation>
    </binding>
    <service name="DeliveryCalculationService">
        <port name="DeliveryCalculationPort"
              binding="tns:DeliveryCalculationBinding">
            <soap:address location="http://127.0.0.1:8081/delivery"/>
        </port>
    </service>
</definitions>
```

This contract is used to analyse the WSDL structure. On its own, it does not create an operational server at the stated address.

7. Part E — Importing into SoapUI and MockService

1. Create a SOAP project in SoapUI from the DeliveryCalculation.wsdl file.
2. Examine the request generated for the calculateDelivery operation.
3. Create a MockService from the binding if your version of SoapUI allows it.
4. Configure a test response for a weight of 1.5, then another for a weight of 4.
5. Send the requests to the MockService and compare the responses.

The MockService simulates responses. It does not replace the Python implementation from TP 1.

8. Exercises

- Add an operation convertEuroToDinar with a parameter and a result of type xsd:float.
- Add a city parameter of type xsd:string to an operation of your choice.
- Explain the consequences of renaming a message or an operation.

9. Questions

1. What is the difference between portType and binding?
2. What is the role of soap:address?
3. Why does WSDL separate the abstract description from the binding details?
4. What is the difference between a WSDL and a data XML file?

10. Deliverables and Evaluation

To submit: generated WSDL, analysis table, manual WSDL, SoapUI screenshots and answers to the questions.

Criterion	Points
Retrieval and analysis	5
Contract manipulation	4
WSDL creation	6
SoapUI tests and analysis	5
Total	20

References

W3C, Web Services Description Language (WSDL) 1.1: https://www.w3.org/TR/wsdl.html • W3C, SOAP 1.2: https://www.w3.org/TR/soap12-part1/ • SmartBear, SoapUI documentation: https://www.soapui.org/docs/

Practical Work Sheet No. 3

Title: Development of a RESTful service and description with WADL

1. Objectives

- Understand the principles of a REST architecture.
- Develop a REST API with Python and Flask.
- Work with HTTP methods and JSON representations.
- Describe the resources and operations with a WADL document.
- Compare REST/WADL with SOAP/WSDL.

2. Environment

Python 3, Flask, Postman and a text editor.

```
python -m pip install flask
python –version
```

3. Part A — Development of the REST API

Create a folder TP3_REST then the file api.py.

```
from flask import Flask, jsonify, request

app = Flask(__name__)

deliveries = {
    1: {"client": "Ali", "city": "Oran", "weight": 1.5},
    2: {"client": "Sara", "city": "Mascara", "weight": 4.0}
}


def calculate_price(weight):
    if weight <= 2:
        return 500
    if weight <= 5:
        return 800
    return 1200


@app.get("/deliveries")
def list_deliveries():
    return jsonify(list(deliveries.values()))


@app.get("/deliveries/<int:id>")
def get_delivery(id):
    delivery = deliveries.get(id)
    if delivery is None:
        return jsonify({"error": "Delivery not found"}), 404
    return jsonify(delivery)


@app.post("/deliveries")
def add_delivery():
    data = request.get_json(silent=True) or {}
    fields = ["client", "city", "weight"]
    if any(field not in data for field in fields):
        return jsonify({"error": "Required fields missing"}), 400
    try:
        weight = float(data["weight"])
    except (TypeError, ValueError):
        return jsonify({"error": "Invalid weight"}), 400
    if weight <= 0:
        return jsonify({"error": "Weight must be positive"}), 400
    new_id = max(deliveries.keys(), default=0) + 1
```

```
    deliveries[new_id] = {
        "client": str(data["client"]),
        "city": str(data["city"]),
        "weight": weight
    }
    return jsonify({"id": new_id, **deliveries[new_id]}), 201


@app.put("/deliveries/<int:id>")
def update_delivery(id):
    if id not in deliveries:
        return jsonify({"error": "Delivery not found"}), 404
    data = request.get_json(silent=True) or {}
    delivery = deliveries[id]
    for field in ["client", "city"]:
        if field in data:
            delivery[field] = str(data[field])
    if "weight" in data:
        try:
            weight = float(data["weight"])
        except (TypeError, ValueError):
            return jsonify({"error": "Invalid weight"}), 400
        if weight <= 0:
            return jsonify({"error": "Weight must be positive"}), 400
        delivery["weight"] = weight
    return jsonify(delivery)


@app.delete("/deliveries/<int:id>")
def delete_delivery(id):
    if id not in deliveries:
        return jsonify({"error": "Delivery not found"}), 404
    del deliveries[id]
    return jsonify({"message": "Delivery deleted"})


if __name__ == "__main__":
    app.run(host="127.0.0.1", port=5000, debug=False)
```

4. Part B — Testing with Postman

Start the server with python api.py. Use the base address http://127.0.0.1:5000.

Method	URL	Action
GET	/deliveries	List the deliveries.
GET	/deliveries/1	Display delivery 1.
POST	/deliveries	Create a delivery with a JSON body.
PUT	/deliveries/1	Modify the fields of a delivery.
DELETE	/deliveries/1	Delete a delivery.

Example JSON body for POST, to select in Postman via Body then raw then JSON.

```
{
    "client": "Amine",
    "city": "Algiers",
    "weight": 3.2
}
```

Also test a non-existent identifier, a negative weight and a missing required field. Record the HTTP codes returned.

5. Part C — Describing the API with WADL

Create a file named API_Delivery.wadl. WADL describes the resources, HTTP methods and representations exchanged.

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
    <resources base="http://127.0.0.1:5000">
        <resource path="/deliveries">
            <method name="GET" id="listDeliveries">
                <response>
                    <representation mediaType="application/json"/>
                </response>
            </method>
            <method name="POST" id="addDelivery">
                <request>
                    <representation mediaType="application/json"/>
                </request>
                <response status="201">
                    <representation mediaType="application/json"/>
                </response>
            </method>
            <resource path="{id}">
                <param name="id" style="template" type="xsd:int"
                       xmlns:xsd="http://www.w3.org/2001/XMLSchema"/>
                <method name="GET" id="getDelivery">
                    <response>
                        <representation mediaType="application/json"/>
                    </response>
                </method>
                <method name="PUT" id="updateDelivery">
                    <request>
                        <representation mediaType="application/json"/>
                    </request>
                    <response>
                        <representation mediaType="application/json"/>
                    </response>
                </method>
                <method name="DELETE" id="deleteDelivery">
                    <response/>
                </method>
            </resource>
        </resource>
    </resources>
</application>
```

The WADL document here is a description separate from the API. It does not automatically generate the Flask server.

6. Part D — Analysis and Comparison

Criterion	SOAP / WSDL	REST / WADL
Style	SOAP messaging protocol	REST architectural style
Description	WSDL	WADL, historically proposed
Common exchange format	XML SOAP envelope	Often JSON or XML
Operations	Operations declared in the contract	Resources and HTTP methods
Tools used in this TP	Spyne and SoapUI	Flask and Postman

7. Exercises

- Add a /clients resource with GET and POST.
- Add a city query parameter to filter the deliveries.
- Update the WADL document to reflect the new routes.
- Compare the HTTP codes 200, 201, 400 and 404 observed during testing.

8. Questions

1. What is a resource in REST?
2. What is the difference between PUT and POST in this TP?
3. Why does a REST API use HTTP status codes?
4. Is WADL mandatory for developing a REST API? Justify.
5. Why is OpenAPI more common than WADL today for documenting APIs?

9. Deliverables and Evaluation

To submit: api.py, API_Delivery.wadl, Postman collection or screenshots, answers to the questions and a report.

Criterion	Points
Flask implementation	7
HTTP testing and validation	5
WADL description	4
Comparison and report	4
Total	20

Practical Work Sheet No. 4

Title: Security and composition of Web Services

1. Objectives

- Identify the main security risks of Web Services.
- Set up JWT token authentication for a REST API.
- Understand the principles of WS-Security for SOAP services.
- Compose several elementary services into a composite service.
- Analyse errors and unavailability in an orchestration.

2. Environment and Installation

```
python -m pip install flask pyjwt requests
```

Create a folder TP4_Security_Composition. The examples use four local servers. Each service must be started in a separate terminal.

3. Part A — Security Concepts

Concept	Role
Authentication	Verify the identity of the requester.
Authorisation	Verify the actions permitted for that identity.
HTTPS/TLS	Protect exchanges during transport.
JWT	Convey signed information as a token.
WS-Security	Add security mechanisms at the SOAP message level.

Risks to study: unauthorised access, identity spoofing, interception, message tampering, injection and denial of service.

4. Part B — JWT Authentication for a REST API

Create the file security.py. The code is a local demonstration. The credentials and the key are deliberately simplified for this TP.

```
from flask import Flask, request, jsonify
import jwt
from datetime import datetime, timedelta, timezone
from functools import wraps

app = Flask(__name__)
SECRET_KEY = "demo-key-to-replace"


def token_required(function):
    @wraps(function)
    def wrapper(*args, **kwargs):
        header_value = request.headers.get("Authorization", "")
        if not header_value.startswith("Bearer "):
            return jsonify({"error": "Missing token"}), 401
        token = header_value.split(" ", 1)[1]
        try:
            payload = jwt.decode(
                token, SECRET_KEY, algorithms=["HS256"]
            )
        except jwt.InvalidTokenError:
            return jsonify({"error": "Invalid token"}), 401
        request.user = payload["user"]
        return function(*args, **kwargs)
    return wrapper
```

```
@app.post("/login")
def login():
    data = request.get_json(silent=True) or {}
    if data.get("username") == "student" and data.get("password") == "123456":
        now = datetime.now(timezone.utc)
        token = jwt.encode({
            "user": "student",
            "exp": now + timedelta(minutes=30)
        }, SECRET_KEY, algorithm="HS256")
        return jsonify({"token": token})
    return jsonify({"error": "Incorrect credentials"}), 401


@app.get("/deliveries")
@token_required
def protected_deliveries():
    return jsonify({
        "user": request.user,
        "deliveries": ["Delivery 1", "Delivery 2"]
    })


if __name__ == "__main__":
    app.run(port=5000, debug=False)
```

5. JWT Tests with Postman

1. Start python security.py.
2. Send POST to http://127.0.0.1:5000/login with the following JSON body.

```
{
    "username": "student",
    "password": "123456"
}
```

3. Copy the returned token.
4. Send GET to http://127.0.0.1:5000/deliveries without a token. Observe the 401 code.
5. Try again with Authorization, type Bearer Token, then paste the token.
6. Change one character of the token and observe the result.

6. Part C — Securing SOAP Messages and WS-Security

WS-Security defines extensions to protect SOAP messages. It can notably carry authentication elements, signatures and encryption mechanisms.

```
<soap:Header>
    <wsse:Security
        xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/
        oasis-200401-wss-wssecurity-secext-1.0.xsd">
        <wsse:UsernameToken>
            <wsse:Username>student</wsse:Username>
            <wsse:Password>demo-password</wsse:Password>
        </wsse:UsernameToken>
    </wsse:Security>
</soap:Header>
```

Required work: identify the SOAP header, the Security block, UsernameToken, Username and Password. Explain why this XML example is not sufficient to secure a production service. Study the role of TLS, signatures and encryption.

7. Part D — Service Composition

Orchestration consists of coordinating several services from a central component. In this TP, the composite service requests the price from one service and the availability from another.

7.1 Price Calculation Service

Create price_service.py.

```
from flask import Flask, jsonify

app = Flask(__name__)


@app.get("/price/<float:weight>")
def price(weight):
    if weight <= 0:
        return jsonify({"error": "Invalid weight"}), 400
    amount = 500 if weight <= 2 else 800 if weight <= 5 else 1200
    return jsonify({"weight": weight, "price": amount})


if __name__ == "__main__":
    app.run(port=5001, debug=False)
```

7.2 Availability Service

Create availability_service.py.

```
from flask import Flask, jsonify

app = Flask(__name__)
available_cities = ["Oran", "Algiers", "Mascara"]


@app.get("/availability/<city>")
def availability(city):
    return jsonify({
        "city": city,
        "available": city in available_cities
    })


if __name__ == "__main__":
    app.run(port=5002, debug=False)
```

7.3 Composite Service

Create composite_service.py. It calls the two elementary services and combines their responses.

```
from flask import Flask, jsonify, request
import requests

app = Flask(__name__)


@app.get("/quote")
def quote():
    city = request.args.get("city", "Oran")
    try:
        weight = float(request.args.get("weight", "1"))
        if weight <= 0:
            return jsonify({"error": "Invalid weight"}), 400
        price_response = requests.get(
            f"http://127.0.0.1:5001/price/{weight}", timeout=3
        )
        availability_response = requests.get(
            f"http://127.0.0.1:5002/availability/{city}", timeout=3
        )
        price_response.raise_for_status()
        availability_response.raise_for_status()
        price = price_response.json()
        availability = availability_response.json()
        return jsonify({
            "city": city,
            "weight": weight,
            "price": price["price"],
            "available": availability["available"]
        })
```

```
    except (requests.RequestException, ValueError):
        return jsonify({"error": "An elementary service is unavailable or the request
is invalid"}), 503


if __name__ == "__main__":
    app.run(port=5003, debug=False)
```

8. Running the Composed Architecture

1. Terminal 1: python price_service.py.
2. Terminal 2: python availability_service.py.
3. Terminal 3: python composite_service.py.
4. Test http://127.0.0.1:5003/quote?city=Oran&weight=3.
5. Test http://127.0.0.1:5003/quote?city=Tlemcen&weight=8.
6. Stop the price service and repeat a test to observe how unavailability is handled.

Test	Expected Result
Oran, weight 3	Price 800, availability true.
Tlemcen, weight 8	Price 1200, availability false.
Elementary service stopped	Controlled error response from the composite service.

9. Further Exercises

- Protect the /quote route with JWT. Reuse the authentication mechanism from Part B.
- Add a third discount service. Grant a 10% reduction for a weight of 5 kg or less.
- Define a maximum delay for each call and explain the role of the timeout parameter.
- Propose a logging strategy for calls and errors.
- Explain how to secure communications between the composite service and the elementary services.

10. Questions

1. What is the difference between authentication and authorisation?
2. What elements does a JWT allow you to verify?
3. Why is a signed JWT not necessarily encrypted?
4. What is the role of WS-Security in SOAP?
5. What is the difference between orchestration and choreography?
6. What happens if an elementary service becomes unavailable?
7. Why must the data received by an API be validated?

11. Deliverables and Evaluation

To submit: the four Python files, the Postman tests, a representation of the composition architecture, the answers to the questions and a report.

Criterion	Points
Analysis of security mechanisms	4
JWT authentication and tests	5
Implementation of the elementary services	4
Orchestration and error handling	5
Report	2
Total	20

Bibliography

Books

[1] Erl, T. (2005). Service-Oriented Architecture: Concepts, Technology, and Design. Upper Saddle River, NJ: Prentice Hall PTR. ISBN 0-13-185858-0.

[2] Newcomer, E., & Lomow, G. (2005). Understanding SOA with Web Services. Boston, MA: Addison-Wesley Professional. ISBN 0-321-18086-0.

[3] Kalin, M. (2013). *Java Web Services: Up and Running* (2nd ed.). O'Reilly Media. ISBN 978-1-4493-7385-6.

[4] Richardson, L., & Ruby, S. (2007). *RESTful Web Services*. O'Reilly Media. ISBN 978-0-596-52926-0.

[5] Yudin, A. (2020). *Building Versatile Mobile Apps with Python and REST: RESTful Web Services with Django and React*. Apress. ISBN 978-1-4842-6333-4.

Standards and Specifications

[6] World Wide Web Consortium (W3C). (2008). Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation. https://www.w3.org/TR/xml/

[7] OASIS. (2004). UDDI Version 3.0.2. OASIS Standard. https://uddi.org/pubs/uddi_v3.htm

[8] OASIS. (2006). Web Services Security: SOAP Message Security 1.1. OASIS Standard. https://docs.oasis-open.org/wss/v1.1/

[9] OASIS. (2007). Web Services Business Process Execution Language Version 2.0 (WS-BPEL). OASIS Standard. https://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html

[10] OASIS. (2005). Security Assertion Markup Language (SAML) V2.0. OASIS Standard. https://docs.oasis-open.org/security/saml/v2.0/

[11] OASIS. (2013). eXtensible Access Control Markup Language (XACML) Version 3.0. OASIS Standard. https://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html

[12] World Wide Web Consortium (W3C). (2004). Web Services Architecture. W3C Working Group Note. https://www.w3.org/TR/ws-arch/

[13] World Wide Web Consortium (W3C). (2007). SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). W3C Recommendation. https://www.w3.org/TR/soap12-part1/

[14] World Wide Web Consortium (W3C). (2001). Web Services Description Language (WSDL) 1.1. W3C Note. https://www.w3.org/TR/wsdl