**Introduction to Web Services**

**What is a Web Service?**

A **Web Service** is a standardized way of **communicating between client and server applications over the internet** using HTTP. It's an **interface** that allows different applications to talk to each other irrespective of their platform, language, or technology stack.

**Characteristics:**

* Platform-independent (Java, .NET, Python, etc.)
* Language-independent
* Based on HTTP (or sometimes other protocols)
* Uses XML or JSON for data exchange

**Why Do We Need Web Services?**

**Real-time Needs:**

1. **Interoperability**: A Java application needs to communicate with a .NET or Python service.
2. **Reusable Logic**: Centralized business logic exposed to multiple consumers.
3. **Loose Coupling**: Frontend and backend are independent and communicate via services.

**Use Cases:**

* A mobile app fetching weather data from a public API
* A banking app checking account balance via a secure REST service
* A telecom billing system exposing SOAP services to third-party vendors

**Types of Web Services**

| **Feature** | **SOAP** | **REST** |
| --- | --- | --- |
| Protocol | Uses strict XML over HTTP, SMTP | Uses HTTP methods |
| Message Format | XML (heavyweight) | JSON or XML (lightweight) |
| Standards | WS-Security, WS-ReliableMessaging | Lightweight, less formal |
| Performance | Slower due to XML overhead | Faster, more efficient |
| Tooling | WSDL for contract | OpenAPI/Swagger for docs |

**1. SOAP (Simple Object Access Protocol)**

**SOAP** is a protocol that relies heavily on XML. It defines rules for structuring messages and uses **WSDL (Web Service Description Language)** for describing the services.

**Example (Java with JAX-WS):**

@WebService

public class BankService {

@WebMethod

public double getBalance(String accountId) {

// logic to get balance

return 1000.50;

}

}

* Deployed on a server
* Accessed by client using WSDL

**Use Cases:**

* Financial services (banking APIs)
* Payment gateways (require strict security)
* Telecom services (postpaid bill reconciliation)

**2. REST (Representational State Transfer)**

**REST** is an architectural style using **HTTP methods (GET, POST, PUT, DELETE)**. It supports multiple formats: **JSON, XML**, etc. Lightweight and widely used in mobile and web applications.

**Example (Java with Spring Boot):**

@RestController

@RequestMapping("/api/accounts")

public class AccountController {

@GetMapping("/{id}/balance")

public double getBalance(@PathVariable String id) {

return 1000.50;

}

}

* Returns JSON like: { "balance": 1000.50 }
* Simple, fast, and easy to integrate

**Use Cases:**

* Mobile apps (food delivery, ride booking)
* E-commerce systems (inventory, orders)
* Telecom recharge APIs

**Deciding Between SOAP and REST**

| **Scenario** | **Recommended** |
| --- | --- |
| Banking, finance (high security, ACID) | SOAP |
| Mobile apps, UI-heavy apps | REST |
| Asynchronous communication, reliable messaging | SOAP |
| Fast development, easy testing | REST |

**Real-time Java-based Use Cases**

**🔸 Banking System**

* SOAP: Used for secure internal services like money transfer, KYC validation.
* REST: Used for mobile banking apps to check balance, view transaction history.

**🔸 Telecom Services**

* SOAP: Billing systems between operators (like Vodafone to BSNL).
* REST: Recharge portals, plan recommendations to customers.

**🔸 E-commerce**

* REST: APIs for product catalog, user carts, order tracking.
* SOAP: Integration with payment gateways or invoice generation systems.

**🔹 Tools and Technologies in Java**

| **Purpose** | **Tools** |
| --- | --- |
| SOAP implementation | JAX-WS, Apache CXF |
| REST implementation | Spring Boot, Jersey, RestEasy |
| Testing | Postman (REST), SoapUI (SOAP) |
| Documentation | WSDL (SOAP), Swagger/OpenAPI (REST) |

**✅ Summary**

| **Topic** | **Summary** |
| --- | --- |
| Web Service | Interface for system-to-system communication |
| SOAP | Protocol, XML-based, secure, used in enterprise & secure environments |
| REST | Lightweight, fast, JSON/XML, popular in modern apps |
| Java Usage | JAX-WS for SOAP, Spring Boot for REST |
| When to Use | SOAP for secure, contract-driven; REST for flexible, scalable services |

**Characteristics of Web Services**

Web services enable communication between different applications over the internet using standard protocols. Below are their key characteristics with **Java-based examples**.

**🔹 1. Platform and Language Independent**

Web services use **standard protocols (like HTTP, XML, JSON)**, which makes them **language-agnostic**.

**Java Example (REST API using Spring Boot):**

@RestController

public class HelloController {

@GetMapping("/hello")

public String sayHello() {

return "Hello from Java Web Service!";

}

}

This API can be consumed by clients built in **Python, PHP, .NET**, etc.

**🔹 2. Interoperability**

They allow **different systems and technologies** to interact. A Java web service can interact with a .NET or Python client.

**✅ Example:**

A **Java-based telecom recharge service** exposes a REST endpoint:

GET /plans → [{ "id": "P1", "name": "Data Pack", "price": 199 }]

A **React Native mobile app** (built in JavaScript) can call this endpoint to show plans to users.

**🔹 3. Uses Standard Protocols**

Web services use **HTTP, HTTPS, XML, JSON, SOAP**, etc., which are **W3C** or **industry standards**.

**✅ Example:**

* RESTful Web Service uses HTTP + JSON
* SOAP Web Service uses HTTP + XML + WSDL

**🔹 4. Self-describing (WSDL/Swagger)**

SOAP services describe themselves using **WSDL (Web Services Description Language)**. REST services use **Swagger/OpenAPI**.

**✅ SOAP WSDL example in Java (JAX-WS):**

@WebService

public class PaymentService {

@WebMethod

public String payBill(String customerId, double amount) {

return "Paid successfully!";

}

}

* When deployed, a WSDL is generated like:

http://localhost:8080/ws/payment?wsdl

**🔹 5. Loosely Coupled**

Client and server interact via a defined interface (API), not direct method calls or tight integration.

**✅ Example:**

@RestController

@RequestMapping("/customer")

public class CustomerController {

@GetMapping("/{id}")

public Customer getCustomer(@PathVariable String id) {

// Get customer from DB

}

}

If the **data store changes**, the frontend doesn't need to change because it depends only on the API.

**🔹 6. Supports Stateless Communication**

Each request is **independent**, meaning no session is required to maintain previous interactions.

**✅ Example:**

@GetMapping("/balance")

public double getBalance(@RequestParam String mobile) {

// returns balance each time independently

}

You can call this endpoint any number of times — the server doesn’t remember previous requests.

**🔹 7. Supports Multiple Data Formats**

* REST supports **JSON, XML, YAML**
* SOAP supports **XML**

**✅ REST Example Returning JSON:**

@GetMapping("/plans")

public List<Plan> getPlans() {

return Arrays.asList(

new Plan("P1", "1GB Data", 199, "28 Days")

);

}

This will return:

[

{

"id": "P1",

"name": "1GB Data",

"price": 199,

"validity": "28 Days"

}

]

**🔹 8. Discoverable (via UDDI, Swagger)**

Some web services are **discoverable dynamically** using tools like **UDDI (for SOAP)** or **Swagger UI (for REST)**.

**✅ Swagger Example (Spring Boot + Swagger):**

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-boot-starter</artifactId>

<version>3.0.0</version>

</dependency>

After setup, Swagger UI will be available at:

http://localhost:8080/swagger-ui/

**🧾 Summary Table**

| **Characteristic** | **Description** | **Java Example** |
| --- | --- | --- |
| Platform independent | Can work across Java, Python, .NET | Spring REST |
| Interoperability | Different languages talk to each other | JSON APIs |
| Standard protocols | HTTP, SOAP, XML, JSON | Spring, JAX-WS |
| Self-describing | WSDL, Swagger to document services | Swagger UI |
| Loosely coupled | Clients depend on API, not implementation | REST APIs |
| Stateless communication | Every request is independent | GET methods |
| Multiple data formats | JSON, XML support | JSON output |
| Discoverable | Registered in Swagger, UDDI | OpenAPI docs |

**REST vs SOAP Web Services**

| **Feature** | **REST** | **SOAP** |
| --- | --- | --- |
| **Definition** | REST (Representational State Transfer) is an **architectural style** for web services | SOAP (Simple Object Access Protocol) is a **protocol** |
| **Protocol** | Uses **HTTP** | Uses **HTTP**, SMTP, TCP |
| **Message Format** | **JSON**, XML, YAML | **XML only** |
| **Transport** | Only **HTTP** | HTTP, SMTP, JMS, FTP |
| **Performance** | **Lightweight and fast** (uses JSON) | Slower (due to XML overhead) |
| **Security** | Uses HTTPS, OAuth, JWT (application level) | WS-Security (built-in) |
| **Standards** | No strict standards, simple and flexible | Strict standards and structure |
| **Best for** | Public APIs, Mobile apps, Microservices | Enterprise-level, secure, reliable messaging |
| **Interface Definition** | OpenAPI (Swagger), not mandatory | Uses **WSDL** (mandatory contract) |
| **Statefulness** | Stateless (usually) | Can be **stateful** (supports session) |
| **Error Handling** | Simple HTTP status codes | Uses detailed **SOAP fault XML** |

**✅ Java Example: REST Web Service (Spring Boot)**

A simple **RESTful API** that returns a list of recharge plans.

@RestController

@RequestMapping("/api")

public class RechargeController {

@GetMapping("/plans")

public List<Plan> getPlans() {

return List.of(

new Plan("P1", "1GB/day", 199, "28 Days"),

new Plan("P2", "2GB/day", 249, "35 Days")

);

}

}

**Output (JSON):**

[

{

"id": "P1",

"name": "1GB/day",

"price": 199,

"validity": "28 Days"

},

{

"id": "P2",

"name": "2GB/day",

"price": 249,

"validity": "35 Days"

}

]

**Java Example: SOAP Web Service (JAX-WS)**

A **SOAP-based service** for checking account balance.

import javax.jws.WebMethod;

import javax.jws.WebService;

@WebService

public class BankService {

@WebMethod

public double getBalance(String accountNumber) {

return 1500.50; // Example response

}

}

**WSDL Generated:**

http://localhost:8080/BankService?wsdl

**SOAP Request (XML):**

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:web="http://webservice/">

<soapenv:Header/>

<soapenv:Body>

<web:getBalance>

<accountNumber>12345</accountNumber>

</web:getBalance>

</soapenv:Body>

</soapenv:Envelope>

**When to Use What?**

| **Scenario** | **Use REST** | **Use SOAP** |
| --- | --- | --- |
| Mobile apps, Web frontends | ✅ | ❌ |
| Real-time public APIs (weather, maps, news) | ✅ | ❌ |
| Enterprise apps requiring high security | ❌ | ✅ (WS-Security, ACID) |
| Messaging over multiple transport protocols | ❌ | ✅ (SMTP, JMS supported) |
| Fast and flexible integration | ✅ | ❌ |
| Strict contracts and schema enforcement | ❌ | ✅ |

**Summary**

| **Aspect** | **REST** | **SOAP** |
| --- | --- | --- |
| Simpler & faster | ✅ | ❌ |
| Flexible data (JSON) | ✅ | ❌ (Only XML) |
| Strong security needed | ❌ (external like OAuth) | ✅ (WS-Security) |
| Loose contract | ✅ | ❌ (WSDL required) |
| Best use case | Mobile & Web APIs | Enterprise, banking, telecom billing |

**BankingHybridAPI**

**✅ Use Case:**

* **REST API** (for customers):
  + Check account details
  + Transfer funds
* **SOAP API** (for internal systems or partners):
  + Get account balance
  + Audit transaction history

**Technology Stack**

| **Layer** | **Tech Used** |
| --- | --- |
| Language | Java 8+ |
| Framework | Spring Boot |
| REST API | Spring Web |
| SOAP API | JAX-WS (with Metro or CXF) |
| Data | In-memory data (or H2 DB optionally) |
| Build Tool | Maven |

**📁 Project Structure**

BankingHybridAPI/

├── src/

│ └── main/

│ ├── java/com/bank/

│ │ ├── controller/ # REST API

│ │ ├── soap/ # SOAP API

│ │ ├── model/ # Entities

│ │ ├── service/ # Business Logic

│ │ └── BankingApplication.java # Main class

│ └── resources/

│ └── application.properties

├── pom.xml

**🔹 pom.xml – Dependencies**

<dependencies>

<!-- Spring Boot REST -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!-- JAX-WS for SOAP -->

<dependency>

<groupId>com.sun.xml.ws</groupId>

<artifactId>jaxws-ri</artifactId>

<version>2.3.3</version>

</dependency>

</dependencies>

**🔹 model/Account.java**

public class Account {

private String accountId;

private String holderName;

private double balance;

// Constructors, Getters, Setters

}

**🔹 model/Transaction.java**

import java.time.LocalDateTime;

public class Transaction {

private String fromAccount;

private String toAccount;

private double amount;

private LocalDateTime timestamp;

// Constructors, Getters, Setters

}

**🔹 service/BankService.java**

@Service

public class BankService {

private final Map<String, Account> accounts = new HashMap<>();

private final List<Transaction> transactions = new ArrayList<>();

public BankService() {

accounts.put("101", new Account("101", "Amit", 5000));

accounts.put("102", new Account("102", "Preety", 3000));

}

public Account getAccount(String id) {

return accounts.get(id);

}

public List<Transaction> getTransactions(String accountId) {

return transactions.stream()

.filter(t -> t.getFromAccount().equals(accountId) || t.getToAccount().equals(accountId))

.collect(Collectors.toList());

}

public Transaction transfer(String fromId, String toId, double amount) {

Account from = accounts.get(fromId);

Account to = accounts.get(toId);

if (from.getBalance() < amount) {

throw new RuntimeException("Insufficient balance");

}

from.setBalance(from.getBalance() - amount);

to.setBalance(to.getBalance() + amount);

Transaction txn = new Transaction(fromId, toId, amount, LocalDateTime.now());

transactions.add(txn);

return txn;

}

public double getBalance(String accountId) {

return accounts.get(accountId).getBalance();

}

}

**🔹 REST Controller**

**controller/BankRestController.java**

@RestController

@RequestMapping("/api")

public class BankRestController {

@Autowired

private BankService bankService;

@GetMapping("/account/{id}")

public Account getAccount(@PathVariable String id) {

return bankService.getAccount(id);

}

@PostMapping("/transfer")

public Transaction transfer(@RequestParam String from,

@RequestParam String to,

@RequestParam double amount) {

return bankService.transfer(from, to, amount);

}

}

**🔹 SOAP Web Service**

**soap/BankSoapService.java**

import javax.jws.WebMethod;

import javax.jws.WebService;

@WebService

public class BankSoapService {

@Autowired

private BankService bankService;

@WebMethod

public double getBalance(String accountId) {

return bankService.getBalance(accountId);

}

@WebMethod

public List<Transaction> getTransactionHistory(String accountId) {

return bankService.getTransactions(accountId);

}

}

**🔹 Main Class with Endpoint Publisher**

**BankingApplication.java**

@SpringBootApplication

public class BankingApplication implements CommandLineRunner {

@Autowired

private BankSoapService bankSoapService;

public static void main(String[] args) {

SpringApplication.run(BankingApplication.class, args);

}

@Override

public void run(String... args) {

Endpoint.publish("http://localhost:8081/ws/bank", bankSoapService);

System.out.println("SOAP service available at: /ws/bank?wsdl");

}

}

**🔹 application.properties**

server.port=8080

**Test the APIs**

**🔸 REST**

* **Check account**:  
  GET http://localhost:8080/api/account/101
* **Transfer funds**:  
  POST http://localhost:8080/api/transfer?from=101&to=102&amount=500

**🔸 SOAP**

* WSDL URL:  
  http://localhost:8081/ws/bank?wsdl
* Use SoapUI to call methods:
  + getBalance(String accountId)
  + getTransactionHistory(String accountId)

**Summary**

| **Feature** | **REST (Customer)** | **SOAP (Internal)** |
| --- | --- | --- |
| Account details | GET /api/account/{id} | getBalance(accountId) |
| Fund transfer | POST /api/transfer?from=... | — |
| Transaction history | — | getTransactionHistory(accountId) |
| Interface | Swagger/OpenAPI (optional) | WSDL (auto generated) |

**1. Web Service Roles**

In a standard web service architecture, there are **three key roles** defined by W3C:

**A. Service Provider**

* Hosts and publishes the web service.
* Makes the service available to consumers.

🔹 *Java Example*:

@RestController

public class AccountService {

@GetMapping("/balance")

public double getBalance(@RequestParam String accNo) {

return 1000.50;

}

}

A Spring Boot REST API acting as the **service provider**.

**B. Service Requestor (Consumer/Client)**

* Invokes the web service provided by the service provider.
* Can be written in any language (Java, Python, etc.)

🔹 *Java REST Client Example* (using RestTemplate):

RestTemplate restTemplate = new RestTemplate();

double balance = restTemplate.getForObject("http://localhost:8080/balance?accNo=12345", Double.class);

**C. Service Registry (UDDI)**

* Optional.
* A centralized directory where services are **published and discovered**.
* Primarily used with **SOAP** and **WSDL**.

Not commonly used today in RESTful architectures, but conceptually important.

**2. Web Service Protocol Stack**

The **Web Service Protocol Stack** is a layered architecture. Each layer provides a specific function.

**Layered View:**

+------------------------+

| Service Description | ← WSDL (SOAP) / OpenAPI (REST)

+------------------------+

| Service Discovery | ← UDDI (Optional, mostly for SOAP)

+------------------------+

| Service Messaging | ← SOAP / HTTP / JSON

+------------------------+

| Service Transport | ← HTTP, SMTP, FTP

+------------------------+

**Layer Breakdown**

| **Layer** | **Function** | **Technology Example** |
| --- | --- | --- |
| **Service Transport** | Transmits messages between apps | **HTTP**, **SMTP**, FTP |
| **Messaging Layer** | Defines message format | **SOAP**, **JSON**, XML |
| **Description Layer** | Describes public interface of service | **WSDL**, **OpenAPI (Swagger)** |
| **Discovery Layer** | Lets clients find available services | **UDDI** (mostly legacy) |

**3. Service Transport Layer**

**What is Service Transport?**

The **transport layer** is the **lowest layer** in the web service stack. It is responsible for **actual data transmission** between the **client** and **server**.

**🔹 Common Transport Protocols**

| **Protocol** | **Use Case** | **Characteristics** |
| --- | --- | --- |
| **HTTP** | Most commonly used for REST & SOAP | Stateless, fast, firewall-friendly |
| **HTTPS** | Secure version of HTTP | Encrypted (SSL/TLS) |
| **SMTP** | Email-based web service transport | Slower, async messaging |
| **FTP** | File-based service interactions | Large file transfer |

**🔹 Java Example Using HTTP (Spring Boot)**

RestController

@RequestMapping("/account")

public class AccountController {

@GetMapping("/balance")

public double getBalance(@RequestParam String accNo) {

return 5000.00;

}

}

Here, the REST service is **transported over HTTP**. If HTTPS is used, Spring Boot can easily enable SSL.

**🔹 SOAP Example Using HTTP Transport**

@WebService

public class BankSoapService {

@WebMethod

public double getBalance(String accountId) {

return 1000.00;

}

}

* This is transported via HTTP or HTTPS.
* The endpoint is accessed via:  
  http://localhost:8081/ws/bank?wsdl

**Summary Table**

| **Concept** | **Description** | **Technology Example** |
| --- | --- | --- |
| **Web Service Roles** | Provider, Consumer, Registry | REST API, Client App, UDDI |
| **Protocol Stack** | Transport → Messaging → Description → Discovery | HTTP, SOAP, WSDL, UDDI |
| **Transport Layer** | Sends actual data | HTTP, SMTP, FTP |

Here’s a clear and detailed explanation of the core concepts related to **Web Services**:

* ✅ XML-RPC
* ✅ SOAP
* ✅ UDDI
* ✅ WSDL

All explained with real-world meaning and **Java examples** or usage relevance.

**1. XML-RPC (XML Remote Procedure Call)**

**✅ Definition:**

**XML-RPC** is a simple protocol that uses **XML to encode its requests and HTTP as the transport protocol**. It allows software running on different OSes and in different environments to make procedure calls over a network.

**Characteristics:**

| **Feature** | **Detail** |
| --- | --- |
| Protocol | XML over HTTP |
| Message Format | Request and response in **XML** |
| Simplicity | Simpler than SOAP, but not widely used now |
| Structure | Very structured, predefined XML tags |

**Example (Conceptual):**

<methodCall>

<methodName>getBalance</methodName>

<params>

<param>

<value><string>12345</string></value>

</param>

</params>

</methodCall>

🔹 You'd send this XML via HTTP POST. The server reads the XML and responds with an XML message.

**Status:**

* Obsolete in modern development
* Replaced by REST, SOAP, or GraphQL

**2. SOAP (Simple Object Access Protocol)**

**Definition:**

**SOAP** is a **protocol** for exchanging structured information in web services. It uses **XML** to format messages and **WSDL** for service description.

**Key Features:**

| **Feature** | **Detail** |
| --- | --- |
| Format | XML |
| Transport | Typically HTTP, but can use SMTP, JMS |
| Security | High (WS-Security, XML Encryption) |
| Contract-based | Uses WSDL to define strict interface |

**SOAP Request Example (Java):**

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"

xmlns:web="http://bank.example.com/">

<soapenv:Body>

<web:getBalance>

<web:accountId>101</web:accountId>

</web:getBalance>

</soapenv:Body>

</soapenv:Envelope>

**Java Example (JAX-WS):**

@WebService

public class BankService {

@WebMethod

public double getBalance(String accountId) {

return 1500.0;

}

}

**3. UDDI (Universal Description, Discovery, and Integration)**

**Definition:**

**UDDI** is a platform-independent registry that allows **businesses to publish and discover web services**.

**Role:**

| **Feature** | **Detail** |
| --- | --- |
| Acts as | A **directory** for web services (like DNS for services) |
| Stores | Service metadata (name, endpoint, WSDL) |
| Used with | **SOAP-based** services |

**Real-World Analogy:**

Think of UDDI as **Yellow Pages for web services**:

* A business publishes its service in UDDI
* Clients browse or search for needed services

**Status:**

* Rarely used today
* RESTful services usually rely on manual discovery or Swagger/OpenAPI

**4. WSDL (Web Services Description Language)**

**Definition:**

**WSDL** is an **XML-based language** used to **describe SOAP web services** — it defines the **operations**, **parameters**, **data types**, and **endpoints** of the service.

**Components of WSDL:**

| **Part** | **Purpose** |
| --- | --- |
| <types> | Defines data types |
| <message> | Defines input/output messages |
| <portType> | Interface with operations |
| <binding> | Protocol and format |
| <service> | Service endpoint |

**Sample WSDL Snippet:**

<wsdl:service name="BankService">

<wsdl:port name="BankPort" binding="tns:BankBinding">

<soap:address location="http://localhost:8081/ws/bank"/>

</wsdl:port>

</wsdl:service>

**Auto-generated in Java (JAX-WS):**

When you create a SOAP service, WSDL is generated at:

http://localhost:8081/ws/bank?wsdl

Clients can use this to generate code using tools like:

* wsimport (Java)
* SoapUI
* .NET Service Reference

**Summary Table**

| **Term** | **Stands For** | **Purpose / Role** | **Usage in Java** | **Status** |
| --- | --- | --- | --- | --- |
| **XML-RPC** | XML Remote Procedure Call | Remote calls using XML over HTTP | Legacy; rarely used | Obsolete |
| **SOAP** | Simple Object Access Protocol | Standard protocol for XML-based services | JAX-WS, Apache CXF | Active |
| **UDDI** | Universal Description, Discovery & Integration | Registry for discovering SOAP services | Rare in modern REST apps | Deprecated |
| **WSDL** | Web Services Description Language | XML doc describing SOAP service | Generated from JAX-WS service | Active in SOAP |

Setting up the IDE for Web service, Building Web Services with JAX-WS, Creating a Simple Web Service and Clients with JAX-WS, Requirements of a JAX-WS Endpoint, Coding the Service Endpoint Implementation Class,

**step-by-step guide** to setting up your IDE and building **SOAP Web Services using JAX-WS** in Java.

We'll cover:

1. ✅ Setting up the IDE
2. ✅ Building Web Services with JAX-WS
3. ✅ Creating a Simple Web Service and Client
4. ✅ Requirements of a JAX-WS Endpoint
5. ✅ Coding the Service Endpoint Implementation Class

**1. Setting Up the IDE for Web Services**

**Recommended IDEs:**

* **IntelliJ IDEA (Community or Ultimate)**
* **Eclipse IDE for Enterprise Java Developers**
* **NetBeans** (has built-in SOAP support)

**Tools Required:**

| **Tool** | **Purpose** |
| --- | --- |
| JDK 8+ | Java runtime + compiler |
| Maven | Dependency management (optional) |
| GlassFish / Tomcat | Web container (if needed) |
| wsimport | Tool to generate client stubs |

**Optional Libraries:**

* jaxws-rt for standalone JAX-WS
* Use **Metro** or **Apache CXF** for better features

**2. Building Web Services with JAX-WS (SOAP)**

JAX-WS is Java’s official API for building SOAP web services.

You can:

* Create a web service using annotations
* Deploy it in a standalone app (via Endpoint.publish) or a WAR file

**3. Creating a Simple Web Service and Client with JAX-WS**

**A. Create the Web Service Endpoint**

// File: CalculatorService.java

import javax.jws.WebMethod;

import javax.jws.WebService;

@WebService

public class CalculatorService {

@WebMethod

public int add(int a, int b) {

return a + b;

}

@WebMethod

public int multiply(int a, int b) {

return a \* b;

}

}

**B. Publish the Web Service**

// File: Publisher.java

import javax.xml.ws.Endpoint;

public class Publisher {

public static void main(String[] args) {

Endpoint.publish("http://localhost:8080/ws/calculator", new CalculatorService());

System.out.println("Service is published at http://localhost:8080/ws/calculator?wsdl");

}

}

* Run Publisher.java
* Go to http://localhost:8080/ws/calculator?wsdl in browser
* You’ll see the **WSDL** document

**4. Requirements of a JAX-WS Endpoint**

| **Requirement** | **Description** |
| --- | --- |
| A public class | The endpoint class must be public |
| No-arg constructor | Required for instantiation |
| Annotated with @WebService | Marks it as a web service |
| @WebMethod (optional) | Annotate exposed methods |
| Runs inside a servlet container or via Endpoint.publish() | Deployment options |

**5. Coding the Service Endpoint Implementation Class**

**Guidelines:**

@WebService

public class BankService {

@WebMethod

public String getBalance(String accountNo) {

// Business logic

return "Balance of " + accountNo + " is ₹5000";

}

@WebMethod(exclude = true)

public void helperMethod() {

// Not exposed to client

}

}

* **@WebService** → Declares this class as a SOAP web service
* **@WebMethod** → Marks individual methods as exposed operations
* **@WebMethod(exclude=true)** → Hides methods from WSDL

**Client Setup Using wsimport**

1. Generate client stubs from WSDL:

wsimport -keep -p com.bank.client http://localhost:8080/ws/calculator?wsdl

1. Use in Java:

CalculatorServiceService service = new CalculatorServiceService();

CalculatorService port = service.getCalculatorServicePort();

int result = port.add(10, 20);

System.out.println("Sum = " + result);

**Summary Workflow**

| **Step** | **Tool / Class** | **Purpose** |
| --- | --- | --- |
| Write service class | @WebService class | Define business methods |
| Publish endpoint | Endpoint.publish() | Make service available over HTTP |
| Generate WSDL | Auto at /service?wsdl | Service contract |
| Generate client stubs | wsimport tool | Consume web service in Java |
| Call from client | Proxy classes | Access remote service methods |

**Project Name: CalculatorSOAPService**

**Project Type: WAR (Web Application Archive)**

To deploy a JAX-WS service to **Tomcat**, we must:

* Package it as a **.war**
* Use **JAX-WS servlet (like Metro)** to expose the endpoint

**1. Tools Required**

| **Tool** | **Version** |
| --- | --- |
| Java | 8 or above |
| Apache Tomcat | 9 or 10 (Servlet container) |
| Maven | For build |
| IDE | IntelliJ / Eclipse |
| JAX-WS RI | Metro (or built-in in Java 8) |

**Project Structure (Maven WAR)**

CalculatorSOAPService/

├── src/

│ └── main/

│ ├── java/com/example/

│ │ ├── service/Calculator.java

│ │ └── service/CalculatorImpl.java

│ └── webapp/

│ ├── WEB-INF/

│ │ └── sun-jaxws.xml

│ │ └── web.xml

├── pom.xml

**2. pom.xml**

<project xmlns="http://maven.apache.org/POM/4.0.0" ...>

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>CalculatorSOAPService</artifactId>

<version>1.0</version>

<packaging>war</packaging>

<dependencies>

<!-- Metro (JAX-WS RI) -->

<dependency>

<groupId>com.sun.xml.ws</groupId>

<artifactId>jaxws-rt</artifactId>

<version>2.3.3</version>

</dependency>

</dependencies>

<build>

<finalName>CalculatorSOAPService</finalName>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-war-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

**3. Java Classes**

**Calculator.java – Service Interface**

package com.example.service;

import javax.jws.WebMethod;

import javax.jws.WebService;

@WebService

public interface Calculator {

@WebMethod

int add(int a, int b);

@WebMethod

int multiply(int a, int b);

}

**CalculatorImpl.java – Service Implementation**

package com.example.service;

import javax.jws.WebService;

@WebService(endpointInterface = "com.example.service.Calculator")

public class CalculatorImpl implements Calculator {

public int add(int a, int b) {

return a + b;

}

public int multiply(int a, int b) {

return a \* b;

}

}

**4. sun-jaxws.xml – JAX-WS Configuration**

Place in src/main/webapp/WEB-INF/

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

<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime"

version="2.0">

<endpoint

name="CalculatorService"

implementation="com.example.service.CalculatorImpl"

url-pattern="/calculator"/>

</endpoints>

**5. web.xml – Deployment Descriptor**

<web-app xmlns="http://java.sun.com/xml/ns/javaee"

version="3.0">

<display-name>CalculatorSOAPService</display-name>

<listener>

<listener-class>com.sun.xml.ws.transport.http.servlet.WSServletContextListener</listener-class>

</listener>

<servlet>

<servlet-name>CalculatorServlet</servlet-name>

<servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>CalculatorServlet</servlet-name>

<url-pattern>/calculator</url-pattern>

</servlet-mapping>

</web-app>

**6. Build and Deploy**

**A. Package the WAR**

In terminal:

mvn clean package

target/CalculatorSOAPService.war

**B. Deploy to Tomcat**

1. Copy the .war file into TOMCAT\_HOME/webapps/
2. Start Tomcat:

cd TOMCAT\_HOME/bin

./startup.sh # or startup.bat on Windows

**C. Access the WSDL**

Open in browser:

http://localhost:8080/CalculatorSOAPService/calculator?wsdl

You should see the WSDL for your service.

**Summary**

| **Component** | **File / Tool** |
| --- | --- |
| Service Code | Calculator.java, CalculatorImpl.java |
| Configuration | sun-jaxws.xml, web.xml |
| Packaging Tool | Maven (WAR file) |
| Server | Apache Tomcat |
| Access URL | http://localhost:8080/CalculatorSOAPService/calculator?wsdl |