Java Enterprise Edition 6

Web Services Developer

SOAP, RESTful API I BEZPIECZEŃSTWO

PRZYGOTOWANIE DO EGZAMINU
ORACLE CERTIFIED EXPERT 1Z0-897

MICHAŁ OLSZA

Słowo wstępne

Niniejsze materiały zostały przygotowane w 2014 roku jako kompendium wiedzy do egzaminu **Oracle Certified Expert, Java EE 6 Web Services Developer** (1Z0-897). Nie są one jednak suchym wykazem zagadnień egzaminacyjnych — celem było ułożenie ich w spójną, logiczną całość, tak aby czytelnik rozumiał zależności między technologiami, a nie jedynie zapamiętał listę faktów.

Dla kogo jest ta książka

Materiały skierowane są do programistów Java, którzy:

- posiadają co najmniej podstawową wiedzę o web serwisach — znają pojęcia takie jak REST, SOAP, HTTP czy XML, choć niekoniecznie zagłębiali się w szczegóły każdego z nich,
- mają zdany egzamin **OCJP** (Oracle Certified Java Programmer) lub jego poprzednik SCJP — jest to formalny warunek uzyskania certyfikatu OCEJWSD,
- chcą usystematyzować i pogłębić wiedzę z zakresu tworzenia usług sieciowych na platformie Java EE,
- przygotowują się do egzaminu lub chcą poszerzyć kompetencje zawodowe w obszarze integracji systemów.

Materiały **nie** są wprowadzeniem do języka Java ani do programowania obiektowego. Zakłada się, że czytelnik swobodnie posługuje się językiem Java i orientuje się w podstawach platformy Java EE (serwlety, EJB, JNDI).

Struktura książki

Książka podzielona jest na pięć części tematycznych, które prowadzą czytelnika od ogólnych pojęć architektonicznych do szczegółowych API platformy Java EE.

Część I — Podstawy (rozdziały 1–3)

Stawia fundamenty pojęciowe. Rozdział *Service Oriented Architecture* opisuje styl architektoniczny, który stanowi tło dla wszystkich omawianych technologii. Rozdział *Usługi sieciowe* wprowadza definicję web serwisu, omawia dwa główne style (REST i SOAP) oraz wymienia ciała normalizacyjne odpowiedzialne za stosowane standardy. Rozdział *Język XML i XMLSchema* dostarcza szczegółowej wiedzy o formacie danych, który jest wspólnym mianownikiem obu stylów usług.

Część II — REST (rozdziały 4–5)

Omawia lżejszy z dwóch stylów usług sieciowych. Rozdział *Usługi sieciowe w stylu REST* tłumaczy zasady architektoniczne REST: zasoby, reprezentacje, bezstanowość i HATEOAS. Rozdział *JAX-RS* opisuje standardowe API Javy do budowania i konsumowania serwisów RESTful, włącznie z negocjacją reprezentacji, cyklem życia

zasobu i mechanizmem providerów.

Część III — SOAP i JAX-WS (rozdziały 6–12)

Stanowi najobszerniejszą część, odpowiadającą największej liczbie pytań egzaminacyjnych. Rozdziały *SOAP* i *WSDL* omawiają protokół i język opisu usług. Rozdziały *JAXP* i *JAXB* opisują niskopoziomowe API do przetwarzania XML oraz mechanizm wiązania XML z obiektami Javy, który jest silnikiem JAX-WS. Rozdział *Standardy WS* objaśnia WS-Addressing i mechanizmy przesyłania załączników binarnych (MTOM/XOP). Rozdział *SAAJ* pokazuje niskopoziomowy dostęp do wiadomości SOAP bezpośrednio z poziomu Java API. Rozdział *JAX-WS* jest centralnym rozdziałem tej części — opisuje kompletne API do tworzenia usług SOAP: adnotacje, strategie WSDL□Java, handlery, komunikację asynchroniczną i obsługę wyjątków.

Część IV — Wdrożenie i transport (rozdziały 13–14)

Rozdział *WSEE* opisuje architekturę Java EE dla web serwisów: pakowanie komponentów, deskryptory i cykl życia usługi w kontenerze. Rozdział *JMS* omawia kolejkowanie wiadomości jako alternatywny transport dla SOAP.

Część V — Bezpieczeństwo i tematy zaawansowane (rozdziały 15–18)

Rozdział *Bezpieczeństwo* obejmuje uwierzytelnianie HTTP, szyfrowanie SSL/TLS, XML Security, WS-Security oraz SAML — odpowiada znaczącej części egzaminu. Rozdział *WSIT* prezentuje implementację zaawansowanych rekomendacji WS-* w projekcie Metro/Glassfish. Rozdział *ebXML* opisuje standardy handlu elektronicznego B2B zbudowane na bazie XML i SOAP. Ostatni rozdział zawiera informacje o samym egzaminie: zakres pytań i progi zaliczenia.

Jak korzystać z materiałów

Rozdziały ułożone są w kolejności rosnącej złożoności i wzajemnych zależności — lektura od początku do końca zapewnia najbardziej płynne przyswajanie wiedzy. Czytelnik przygotowujący się do egzaminu powinien jednak zwrócić szczególną uwagę na rozdziały JAX-WS, Bezpieczeństwo i WSEE, które odpowiadają za największą liczbę pytań.

Przykłady kodu w rozdziałach JAXP, JAXB, JAX-RS i JAX-WS warto uruchomić samodzielnie — nic nie zastąpi bezpośredniego kontaktu z działającym kodem podczas przygotowań do egzaminu praktycznego.

Spis Treści

Część I: Podstawy

1. Service Oriented Architecture

1.1. Motywacja

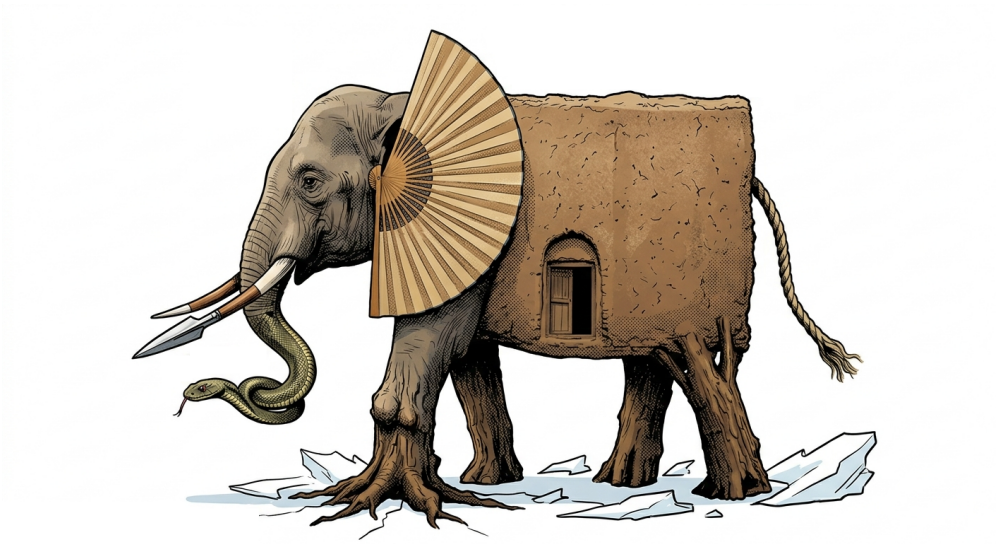
Jedną z cech systemów rozproszonych jest ich *kruchość* (`brittleness`), za pomocą której określamy podatność systemu na zmiany komponentów z którymi jest powiązany. Z kruchością mamy do czynienia gdy:

- program nie może pracować na innym systemie operacyjnym,
- system nie może pracować z inną bazą danych,
- mała zmiana w jednym komponencie wymusza lawinowe zmiany w innych,
- niewielka zmiana modelu biznesowego kończy się koniecznością aktualizacji całego oprogramowania.

Service Oriented Architecture (SOA) jest to styl architektoniczny oparty na konkretnych autonomicznych jednostkach, z których każda dostarcza konkretną usługę i działa niezależnie od pozostałych. Głównym celem takiego podejścia jest zmniejszenie kruchości budowanych systemów. Należy mieć na uwadze, że SOA samo w sobie nie jest implementacją, ani nie jest to też zbiór narzędzi.

> *SOA są jak płatki śniegu — nie ma dwóch jednakowych.*
>
> — David Linthicum

Postrzeganie SOA jest podobne do rozpoznawania słonia przez sześciu niewidomych mężczyzn w wierszu Johna Saxe'a[1] — w zależności od tego, której części słonia dotknęli rozpoznawali go jako węża, włócznię, drzewo, wachlarz, linę lub mur.



Rysunek 1. Słoń jako metafora SOA

1.2. Pojęcie serwisu w SOA

SOA opiera się na pojęciu serwisu, który niekoniecznie musi być web serwisem — system może być oparty o rozwiązania takie jak CORBA lub DCOM. Serwis nie musi być powiązany z interfejsem sieciowym, może to być także demon w systemach linuksowych lub serwis na platformie Windows.

Serwisy w rozumieniu SOA:

- **mają wyraźne granice** — każdy serwis musi mieć dobrze określone granice, z dobrze zdefiniowanymi interfejsami, pomiędzy którymi przesyłane są wiadomości,
- **są autonomiczne** — każdy serwis podlega własnemu niezależnemu procesowi rozwoju, jest tworzony, wersjonowany i publikowany niezależnie od innych,
- **serwisy wymieniają się kontraktami, nie klasami** — interakcja pomiędzy serwisami musi odbywać się tylko poprzez ich publiczne interfejsy. Choć czasami kuszące wydaje się udostępnianie wewnętrznych fragmentów kodu innym podsystemom, to jednak utrzymanie takiego środowiska prędzej czy później staje się bardzo trudne,
- **charakterystyka serwisów opisana jest przez polityki** — kompatybilność pomiędzy systemami zależy od bardzo wielu czynników i wymagań, które dany serwis musi spełniać. Polityki są właśnie metodą na przedstawienie w szczegółowy sposób możliwości serwisu, przy jednoczesnym oderwaniu od implementacji.

1.3. Role w architekturze SOA

Współpraca w architekturze zorientowanej na usługi opiera się na trzech rolach:

Dostawca serwisu (*service provider*)

Implementuje i udostępnia serwis. Opisuje serwis za pomocą kontraktu i publikuje informację o nim w rejestrze, tak aby potencjalni konsumenci mogli go odnaleźć.

Konsument serwisu (*service consumer*)

Wyszukuje potrzebny serwis w rejestrze, pobiera jego opis i nawiązuje połączenie bezpośrednio z dostawcą. Po nawiązaniu połączenia rejestr nie bierze już udziału w komunikacji.

Rejestr serwisów (*service registry*)

Centralny katalog dostępnych serwisów. Przechowuje kontrakty i opisy serwisów, umożliwiając ich wyszukiwanie według nazwy, kategorii lub wymaganych możliwości.

W świecie web serwisów rolę rejestru pełni standard **UDDI** (*Universal Description, Discovery and Integration*), który jest szczegółowo omówiony w kolejnym rozdziale.

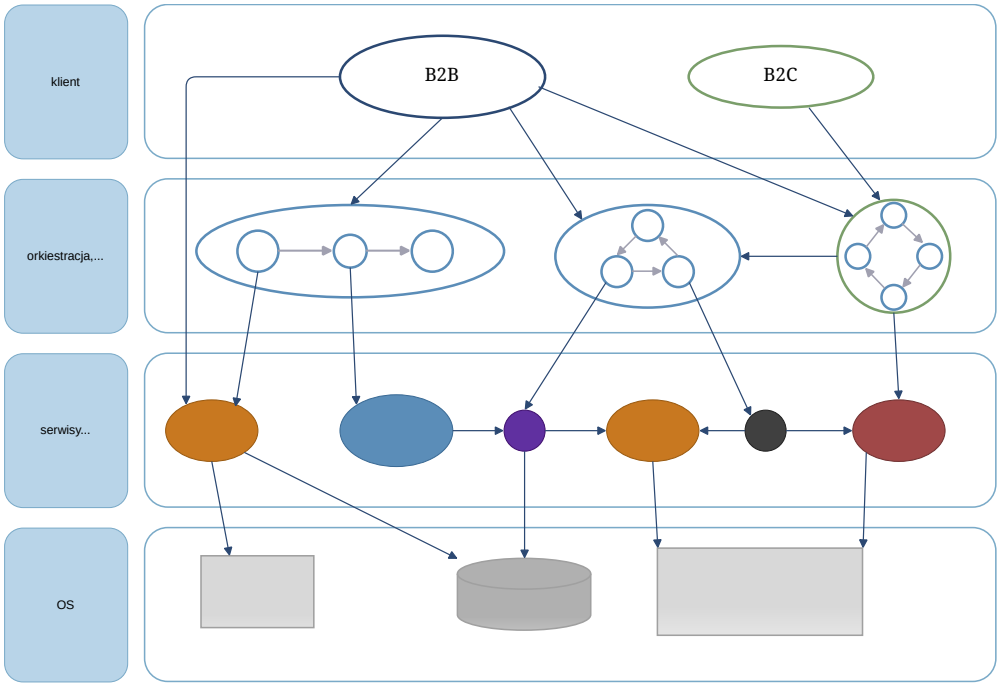
1.4. Elementy architektoniczne i charakterystyka SOA

W trakcie tworzenia architektury systemu SOA tworzone są następujące artefakty:

- zestaw serwisów, definiujący dostarczane usługi,
- konkretna architektura określająca używaną technologię (jedną z nich mogą być web serwisy),
- zasady architektoniczne, wzorce projektowe, kryteria oceny,
- model programowania (konkretne frameworki, języki, narzędzia służące do implementacji),
- rozwiązania wspomagające, takie jak grupowanie serwisów, orkiestracja, monitoring, zarządzanie.

Natomiast sam proces budowy cechuje się tym, że:

- **system budowany jest iteracyjnie**
  Zbudowanie i wdrożenie całego systemu w jednym akcie jest wielce ryzykowne. SOA pozwala wprowadzać każdy system do przedsiębiorstwa stopniowo poprzez zastępowanie lub zrównoleglanie (w fazie przejściowej) pojedynczych pracujących komponentów.
- **luźno związane komponenty mogą ewoluować niezależnie**
  Każdy serwis czy podsystem wystawiając tylko jeden, rozszerzalny, publiczny interfejs ukrywa całą swoją wewnętrzną strukturę, przez co jej zmiany nie mają wpływu na działanie całego systemu.
- **architektura jest fraktalna, a nie warstwowa**
  W celu dostarczenia usług, które będą miały wartość dla klienta, konieczne jest często łączenie funkcjonalności wielu komponentów systemu. Dlatego proste serwisy łączy się w grupy, z których korzystają inne serwisy (wyższego poziomu), te natomiast mogą być częściami procesów, którymi zarządzają kolejne serwisy.



Rysunek 2. Schemat systemów opartych o serwisy

1.5. Kompozycja serwisów — orkiestracja i choreografia

Wartość biznesowa SOA ujawnia się często dopiero w połączeniu prostych serwisów w złożone procesy biznesowe. Wyróżniamy dwa podejścia do takiej kompozycji:

Orkiestracja (*orchestration*)

Jeden serwis — dyrygent (*orchestrator*) — wywołuje pozostałe serwisy w określonej kolejności, podejmuje decyzje warunkowe i agreguje wyniki. Cała logika procesu zdefiniowana jest centralnie, co ułatwia monitorowanie i diagnozowanie błędów. Standardem opisu orkiestracji jest *Business Process Execution Language* (BPEL).

Choreografia (*choreography*)

Brak centralnego koordynatora — każdy serwis reaguje na zdarzenia i samodzielnie decyduje o kolejnych krokach, publikując własne zdarzenia dla pozostałych uczestników. Logika procesu jest rozproszona między uczestnikami, co zwiększa odporność na awarie, ale utrudnia pełne śledzenie przebiegu procesu.

Oba podejścia mogą być stosowane jednocześnie: orkiestracja wewnątrz domeny, choreografia na styku różnych organizacji lub systemów.

1.6. SOA a web serwisy

SOA jest stylem architektonicznym — nie narzuca konkretnej technologii implementacji. Serwisy mogą być zbudowane z użyciem CORBA, DCOM, RMI lub, co jest przedmiotem niniejszego opracowania, **web serwisów**. Web serwisy bazujące na otwartych standardach (XML, SOAP, WSDL, UDDI) stanowią dziś dominującą technologię realizacji

architektury SOA w środowiskach heterogenicznych. Kolejne rozdziały opisują te standardy oraz odpowiadające im API platformy Java EE.

[1] *The Blind Men And The Elephant*

2. Usługi sieciowe

2.1. Czym jest Web Service?

> *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 a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.*
>
> — Web Services Architecture, W3C Working Group Note, 11 February 2004

> *Web services are modular business process applications, based on open Internet standards, that are able to describe their own functionality, locate and dynamically interact with other Web services. They provide a method for different organizations to conduct dynamic e-business across the Internet, regardless of the application or the language in which the service was implemented.*
>
> — James McGovern, Sameer Tyagi, Michael Stevens and Sunil Matthew, Java Web Services Architecture, 2003

2.2. Podstawowe cechy Web Serwisów

Usługi sieciowe charakteryzują się tym, że:

- są **hermetyczne** (*self-contained*) — każda usługa jest kompletną, samodzielną jednostką funkcjonalną,
- dostępne przez **otwarte protokoły** (HTTP, SMTP, JMS),
- komunikują się za pomocą **ustandaryzowanych i przenośnych formatów** wiadomości (XML, JSON),
- są **kontraktowe** (*self-describing*) — opisują swoje możliwości za pomocą schematów (WSDL, WADL, XMLSchema),
- **przeznaczone do użycia przez inne aplikacje** — nie przez człowieka bezpośrednio,
- są **wykrywalne** — mogą być rejestrowane i wyszukiwane w katalogach (UDDI, inne rejestry).

2.2.1. Web Serwisy na tle innych systemów rozproszonych

Model przetwarzania rozproszonego oferowanego przez web serwisy jest podobny do innych rozwiązań takich, jak:

- CORBA,
- Distributed Component Object Model (DCOM),

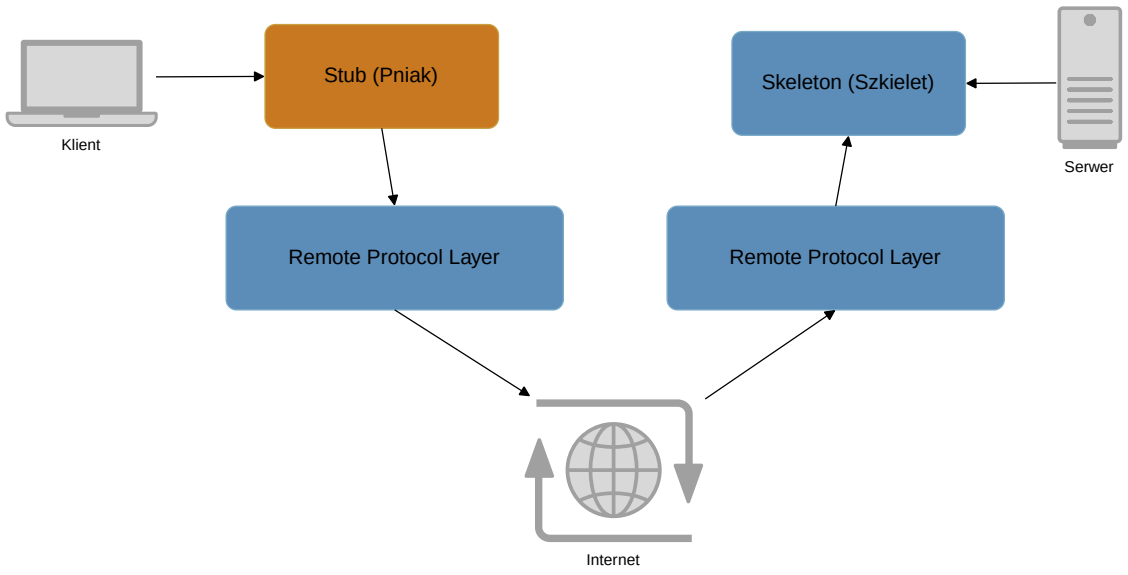
- Java RMI.

Web serwisy wyróżniają się użyciem otwartych i popularnych standardów, co rozwiązuje wiele problemów:

- są interoperacyjne (*interoperability*),
- są stworzone do działania w Internecie,
- odeszły od stylu RPC (*Remote Procedure Call*) na korzyść przesyłania dokumentów.

W technologiach takich jak RMI czy CORBA użytkownik zwolniony jest z programowania warstwy komunikacji sieciowej (*Remote Protocol Layer*). W rozwiązaniach tych użytkownik używa namiastek obiektów, czyli obiektów odpowiedzialnych za serializowanie i przesyłanie wiadomości przez sieć. Namiastki używane są w taki sposób, jakby konkretne obiekty znajdowały się lokalnie. Program wywołuje metodę na namiastce z konkretnymi parametrami, które następnie są serializowane i wysyłane do serwera, który przeprowadza proces deserializacji w swojej namiastce i wywołuje konkretną metodę na obiekcie docelowym. Następnie zwracany wynik metody przechodzi odwrotny proces.

Namiastki obiektów po stronie klienta nazywa się *pniakami (stub)*, a po stronie serwera *szkieletem (skeleton)*.



Rysunek 3. Model pniak – szkielet

Trzeba jednak zaznaczyć, że web serwisy nie stanowią idealnego rozwiązania w każdym przypadku. W systemach lokalnych lub takich, gdzie nie napotyka się na problemy związane z zaporami sieciowymi, a technologie tworzenia poszczególnych komponentów są pod kontrolą jednej organizacji, rozwiązania takie jak CORBA czy RMI stanowią znakomitą alternatywę, szczególnie w przypadku systemów, które z natury opierają się na modelu RPC.

W razie potrzeby możliwe jest użycie CORBY czy RMI wraz z HTTP, co pozwala na komunikację przez Internet, jednak takie rozwiązania wprowadzają dodatkowy narzut protokołu i są rzadko stosowane w środowiskach heterogenicznych, gdzie nie ma kontroli nad technologią po obu stronach połączenia.

2.3. HTTP i XML

Hypertext Transfer Protocol (HTTP) jest protokołem aplikacyjnym, opracowanym przez Internet Engineering Task Force wspólnie z W3C. HTTP pracuje w modelu klient-serwer (*request-response*), jest oparty na niezawodnym protokole TCP i powszechnie wspierany przez infrastrukturę sieciową (zapory, proxy, bramki).

2.3.1. Metody HTTP

HTTP definiuje zestaw metod (czasowników) określających zamierzoną akcję na zasobie:

Metoda	Cechy	Opis
`GET`	bezpieczna, idempotentna	Pobieranie zasobu bez efektów ubocznych po stronie serwera
`POST`	—	Tworzenie zasobu podrzędnego lub wywołanie akcji
`PUT`	idempotentna	Zastąpienie zasobu lub jego utworzenie pod podanym URI
`DELETE`	idempotentna	Usunięcie zasobu
`PATCH`	—	Częściowa aktualizacja zasobu
`HEAD`	bezpieczna, idempotentna	Jak GET, lecz bez ciała odpowiedzi — zwraca wyłącznie nagłówki
`OPTIONS`	bezpieczna, idempotentna	Informacja o metodach obsługiwanych przez dany zasób

Metoda **bezpieczna** (*safe*) nie zmienia stanu serwera. Metoda **idempotentna** (*idempotent*) daje ten sam wynik niezależnie od liczby powtórzeń — ponowne wysłanie żądania jest bezpieczne. Właściwości te mają bezpośrednie przełożenie na projekt serwisów RESTful i sposób obsługi błędów sieciowych.

SOAP over HTTP używa wyłącznie metody `POST` niezależnie od semantyki operacji.

2.3.2. Kody odpowiedzi HTTP

Każda odpowiedź serwera zawiera trzycyfrowy kod statusu, pogrupowany w pięć klas:

Klasa	Znaczenie i przykłady
1xx	Informacyjne — żądanie przyjęte, przetwarzanie w toku
2xx	Sukces — `200 OK`, `201 Created`, `204 No Content`, `202 Accepted`
3xx	Przekierowanie — `301 Moved Permanently`, `304 Not Modified`, `307 Temporary Redirect`
4xx	Błąd klienta — `400 Bad Request`, `401 Unauthorized`, `403 Forbidden`, `404 Not Found`, `415 Unsupported Media Type`
5xx	Błąd serwera — `500 Internal Server Error`, `503 Service Unavailable`

W kontekście web serwisów szczególne znaczenie mają: `200 OK` dla odpowiedzi pozytywnych, `400 Bad Request` dla nieprawidłowo sformatowanych żądań, `401 Unauthorized` i `403 Forbidden` dla problemów z autoryzacją, `404 Not Found` gdy zasób nie istnieje oraz `500 Internal Server Error` — używany przez SOAP do sygnalizowania błędu `SOAP Fault`.

eXtensible Markup Language (XML) jest rozszerzalnym językiem znaczników:

- dokumenty są czytelne zarówno dla człowieka, jak i dla maszyny,
- niezależny od systemu operacyjnego i nośnika,
- prosty, ogólny i rozszerzalny,
- powszechnie obsługiwany przez biblioteki na wszystkich platformach.

Dlatego HTTP i XML są tak chętnie używane w SOA — oba opierają się na otwartych standardach i działają bez ograniczeń przez zapory sieciowe. Szczegółowy opis struktury dokumentów XML oraz XMLSchema zawarty jest w następnym rozdziale.

2.4. Ciała nadzorujące rozwój

Technologia web serwisów opiera się o prawie 100 standardów i rekomendacji wydanych przez:

- World Wide Web Consortium (W3C),
- Internet Engineering Task Force (IETF),
- Organization for the Advancement of Structured Information Standards (OASIS),
- Web Services Interoperability Organization (WS-I).

2.5. Rejestr UDDI

Universal Description, Discovery and Integration (UDDI) jest standardem rejestru usług sieciowych utrzymywanym przez OASIS. Pełni rolę analogiczną do katalogu telefonicznego — pozwala dostawcom publikować opisy swoich serwisów, a

konsumentom wyszukiwać usługi spełniające ich wymagania. UDDI bezpośrednio realizuje rolę *rejestru serwisów* opisaną w architekturze SOA.

Rejestr UDDI przechowuje trzy rodzaje informacji:

Białe strony (*white pages*)

Dane identyfikacyjne firmy: nazwa, adres, numery kontaktowe.

Żółte strony (*yellow pages*)

Kategoryzacja firmy i jej usług według standardowych taksonomii branżowych (np. kody NAICS lub UNSPSC).

Zielone strony (*green pages*)

Techniczne szczegóły serwisów: adresy endpointów i referencje do dokumentów WSDL opisujących interfejs.

Komunikacja z rejestrem UDDI odbywa się przez API oparte na SOAP i HTTP. W praktyce publiczne rejestry UDDI (takie jak prowadzone niegdyś przez IBM i Microsoft) nie zyskały szerokiego zastosowania — firmy preferują udostępnianie opisów WSDL bezpośrednio pod ustalonym adresem URL, zamiast publikowania ich w centralnym rejestrze.

2.6. Dwa style usług sieciowych: SOAP i REST

Usługi sieciowe dzielimy na dwie kategorie:

- **Activity-oriented** (jak SOAP)
- **Resource-oriented** (REST)

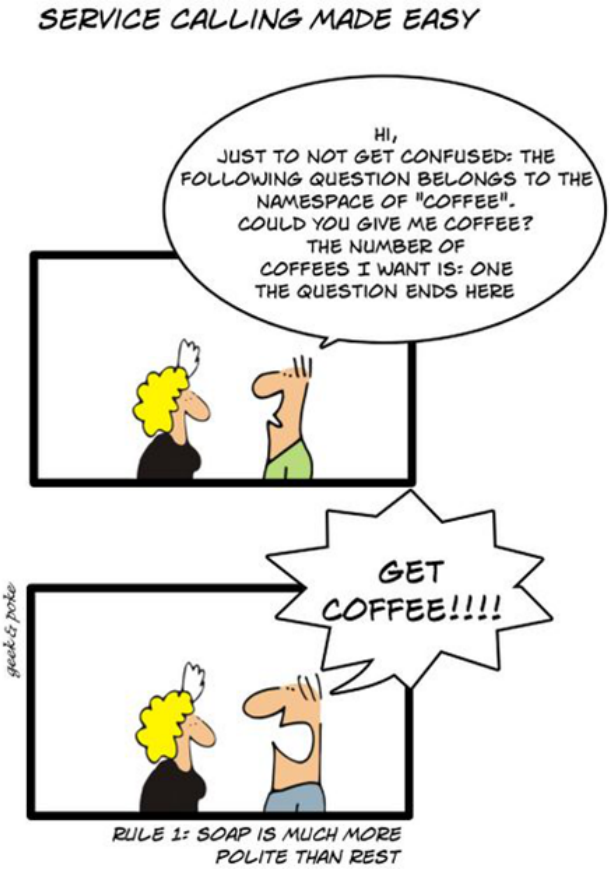
Choć zgodnie z definicją W3C tylko usługi wykorzystujące protokół SOAP mogą być nazywane Web Service'ami, to jednak nazwa ta jest często stosowana do wszystkich usług sieciowych opartych o wywoływanie akcji.

Przykładami ciekawych protokołów są:

- SOAP,
- JSON-RPC, JSON-WSP,
- Hessian (binarny protokół, dostępny dla wielu języków),
- Burlap (XML RPC) stworzony jako protokół do komunikacji *Enterprise Java Beans* z klientami nie napisanymi w Javie.

SOAP — Simple Object Access Protocol	**REST — REpresentational State Transfer**
Protokół oparty na XML, niezależny od warstwy transportowej (HTTP, JMS, SMTP)	Styl architektoniczny ściśle związany z HTTP i jego metodami

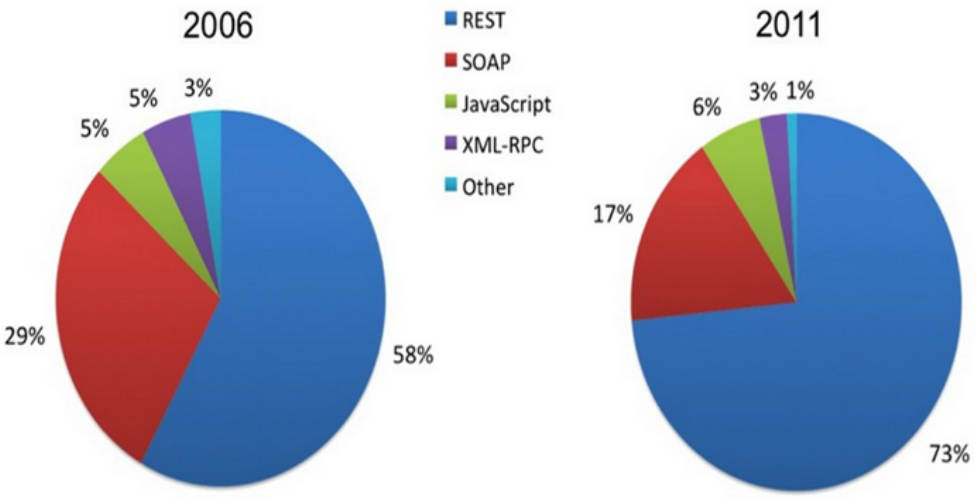
SOAP — Simple Object Access Protocol	REST — REpresentational State Transfer
Komunikat ma ściśle określoną strukturę: koperta, nagłówek, ciało	Brak narzuconej struktury wiadomości — format dowolny (XML, JSON, HTML…)
Opisuje operacje i typy danych (WSDL + XMLSchema)	Zasoby identyfikowane przez URI; operacje wyrażone metodami HTTP
Bardzo sformalizowany — bogaty ekosystem standardów WS-* (WS-Security, WS-ReliableMessaging…)	Brak formalnych standardów poza HTTP — prostota kosztem standaryzacji
Wbudowana obsługa transakcji, bezpieczeństwa na poziomie wiadomości, niezawodnego dostarczania	Bezstanowość i cacheowalność wbudowane w HTTP; bezpieczeństwo na poziomie transportu (TLS)
Stosowany w systemach korporacyjnych wymagających zaawansowanych gwarancji i interoperacyjności	Dominuje w publicznych API i aplikacjach webowych ze względu na prostotę i lekkość



Rysunek 4. SOAP jest grzeczniejszy niż REST

Częstym argumentem zwolenników REST jest jego popularność mierzona liczbą dostępnych serwisów w różnych publicznych katalogach. Przewaga usług tego typu wynika bezpośrednio z ich charakteru — są to często małe, proste, często bezpłatne usługi dostępne dla wszystkich, np. zmiana wielkości obrazków. Serwisy typu SOAP są to

duże, specjalistyczne i zaawansowane aplikacje, do których uzyskanie dostępu i ich wdrożenie wymaga znacznej pracy i dlatego nie są to usługi, które nadają się do publicznych katalogów.



Rysunek 5. Udziały REST i SOAP w rynku

3. Język XML i XMLSchema

XML jest podstawowym formatem danych w ekosystemie web serwisów — SOAP, WSDL oraz XMLSchema są dokumentami XML i w pełni opierają się na jego składni i mechanizmach. Przestrzenie nazw XML pozwalają SOAP-owi oddzielać elementy protokołu (kopertę) od treści wiadomości, a XMLSchema stanowi system typów wbudowany w WSDL. Niniejszy rozdział opisuje te fundamenty w zakresie niezbędnym do zrozumienia kolejnych rozdziałów.

3.1. Co to jest XML?

XML jest skrótem od *Extensible Markup Language* — jest to uniwersalny i formalny język przeznaczony do reprezentowania i wymiany danych w sposób ustrukturyzowany, czytelny zarówno dla człowieka, jak i dla maszyny. Autor dokumentu definiuje własny zestaw znaczników odpowiadający reprezentowanym danym.

Poniżej zamieszczony został przykład dokumentu XML reprezentującego opis książki. Granice znaczników (tagi) umieszczane są w ostrokątnych nawiasach `<>`. Atrybut jest w formacie `nazwa="wartość"`, np. `<book id="bk001">`. Jeśli znacznik nie posiada zawartości można użyć skrótowego zapisu `<author/>`, który jest równoznaczny z `<author></author>`.

```
1 <?xml version="1.0"?>
2 <book id="bk001">
3   <author>Hightower, Kim</author>
4   <title>The First Book</title>
5   <genre>Fiction</genre>
6   <price>44.95</price>
7   <pub_date>2000-10-01</pub_date>
8   <review>An amazing story of nothing.</review>
9 </book>
```

Pierwsza linia jest deklaracją XML — opisuje wersję i często kodowanie dokumentu. Jeśli jest używana, musi rozpoczynać dokument. Jest to przykład instrukcji procesowej (PI), które ogólnie ograniczane są przez `<? ?>`.

3.2. Dobre formatowanie i poprawność

Aby można było dokument nazwać dokumentem XML musi on być dobrze sformatowany (`well-formed`):

- wszystkie otwarte tagi muszą być zamknięte,
- tagi nie mogą się przecinać, np: `<author><title></author></title>`,
- może być jeden i tylko jeden tag główny (korzeń),
- używa tylko jednego kodowania znaków zgodnego z tym w deklaracji XML.

Każde odstępstwo od powyższych reguł definiowane jest jako błąd krytyczny (`fatal`

`error`).

Dokument jest **poprawny** (`valid`) wtedy, gdy jest zgodny z przyjętym schematem, który może być zdefiniowany za pomocą DTD (*Document Type Definition*) lub XML Schema. Odstępstwa od poprawności określane są mianem błędu (`error`).

3.3. Przestrzenie nazw (namespaces)

Możliwość definiowania własnych znaczników powoduje ryzyko powstania kolizji nazw. Dwa lub więcej znaczników pochodzących z różnych modeli może posiadać takie same nazwy, a ze względu na rozszerzalność języka XML mogą wystąpić w jednym dokumencie.

Rozwiązaniem tego problemu są przestrzenie nazw, które grupują znaczniki jednego modelu nadając im globalną unikalność.

```
1 <?xml version="1.0"?>
2 <x:book id="bk001" xmlns:x="urn:books">
3   <x:author>Hightower, Kim</x:author>
4   <x:title>The First Book</x:title>
5   <x:genre>Fiction</x:genre>
6   <x:price>44.95</x:price>
7   <x:pub_date>2000-10-01</x:pub_date>
8   <x:review>An amazing story of nothing.</x:review>
9 </x:book>
```

Rozpoczęcie użycia przestrzeni nazw definiowane jest za pomocą atrybutu `xmlns`, po którym, po dwukropku, występuje skrót (`prefix`), który będzie odnosił się do definiowanej przestrzeni w dokumencie. Przestrzeń nazw jest unikalnym URI.

Tabela 1. Podstawowe terminy — przykład dla elementu `book`

Termin	Wartość
qualified name	`x:book`
local name	`book`
namespace prefix	`x`
namespace name	`urn:books`
expanded name	`{urn:books}book`

3.4. Document Type Definition

Część standardu XML określająca jego strukturę. Za pomocą *Document Type Definition* (DTD) możemy opisywać co powinno znajdować się w dokumencie, definiować obligatoryjność elementów, wskazywać na domyślne wartości. Wyjątkową cechą DTD jest możliwość definiowania encji (*entity*), które mogą być rozwijane do określonych

wartości lub wskazywać na zewnętrzne zasoby, np. pliki binarne.

DTD:

- jest częścią standardu XML,
- pozwala opisywać strukturę dokumentu,
- definicję elementów i atrybutów,
- w ograniczonym zakresie ich typy,
- encje i notacje.

DTD przenoszone jest razem z dokumentami, które opisuje. Deklaracja DTD umieszczana jest za deklaracją XML i przed pierwszym elementem.

```
1 <?xml version="1.0"?>
2 <!DOCTYPE book [
3   <!ELEMENT book (#PCDATA)>
4   <!ATTLIST book author CDATA #IMPLIED>
5 ]>
6 <book id="bk001">...</book>
```

3.4.1. Definiowanie elementów

Elementy definiuje się za pomocą wpisu `!ELEMENT`:

```
1 <!ELEMENT nazwa-elementu typ-zawartosci-elementu>
```

Typ zawartości może przyjąć jedną z wartości:

- `ANY` — zawartość dowolna,
- `EMPTY` — element pusty,
- `#PCDATA` — zawartość tekstowa,
- wyrażenia regularne, np: `(#PCDATA | nazwa-elementu1 | nazwa-elementu2 ⋯)*`.

3.4.2. Definiowanie atrybutów

Atrybuty definiuje się za pomocą listy `!ATTLIST`:

```
1 <!ATTLIST nazwa-elementu
2     nazwa-atrybutu1 typ-zawartosci1 opis-parametru1
3     nazwa-atrybutu2 typ-zawartosci2 opis-parametru2
4 >
```

Typ zawartości może przyjąć jedną z wartości:

- `CDATA` — tekst,

- `ID` — unikalna nazwa w całym dokumencie, identyfikator elementu,
- `IDREF` — typ, który jest odnośnikiem do typów ID, zawiera referencję do innego elementu,
- `NMTOKEN, NMTOKENS` — tokeny tekstowe,
- *(token1 | token2 ...)* — jedna z wartości w nawiasie,
- `ENTITY, ENTITIES, NOTATION`.

Natomiast *opis-parametru* może przyjąć wartość:

- `#REQUIRED` — atrybut jest wymagany,
- `#IMPLIED` — atrybut opcjonalny,
- `"wartość-domyślna"` — atrybut z wartością domyślną, którą przyjmuje kiedy nie jest zdefiniowany w dokumencie,
- `#FIXED "wartość"` — atrybut opcjonalny, z wartością nie inną niż *"wartość"*.

3.4.3. Encje i notacje

Encje deklaruje się za pomocą `!ENTITY`. Mają one swoją nazwę i wartość:

```
1 <!ENTITY nazwa-encji wartosc-encji>
```

W dokumencie do tak zdefiniowanej encji odwołuje się za pomocą jej nazwy `&nazwa_encji;`.

```
1 <!ENTITY wewnetrzna "encyjka &lt;bla&gt;123&lt;/bla&gt;">
2
3 <!ENTITY zewnetrzna_id_systemowy
4     SYSTEM "http://example.com/file.xml">
5
6 <!ENTITY zewnetrzna_nieprzetwarzana
7          SYSTEM "../grafix/image.gif"
8          NDATA gif>
```

Encje nieprzetwarzane oznaczane są za pomocą atrybutu `NDATA` i następującej za nim *notacji*. Notacje służą do opisu nie-XMLowych formatów danych, do których referencje znajdują się w dokumencie.

```
1 <!NOTATION nazwa SYSTEM "SystemID">
2 <!NOTATION nazwa PUBLIC "PublicID">
3 <!NOTATION nazwa PUBLIC "PublicID" "SystemID">
4
5 <!NOTATION gif PUBLIC "GIF 1.0" "gimp.exe">
```

DTD jest pierwszym i już rzadko wykorzystywanym standardem do opisu struktury dokumentów XML. Jego miejsce zajęły potężniejsze

> języki schematów takie jak: *XMLSchema*, *RELAX NG* czy *Schematron*.

3.5. Opis struktury dokumentów za pomocą XMLSchema

XML Schema czyli *Schemat Rozszerzalnego Języka Znaczników* został opracowany przez W3C jako standard służący do opisu struktury dokumentów XML. XML Schema określa zbiór znaczników, przyjmowane przez nie wartości oraz ich wzajemne relacje. W odróżnieniu od DTD, XML Schema jest dokumentem XML i wspiera przestrzenie nazw.

Specyfikacja XML Schema została podzielona na 3 części:

- XML Schema Part 0: Primer — https://www.w3.org/TR/xmlschema-0/
- XML Schema Part 1: Structures — https://www.w3.org/TR/xmlschema-1/
- XML Schema Part 2: Datatypes — https://www.w3.org/TR/xmlschema-2/

XML Schema definiuje:

- elementy i atrybuty występujące w dokumencie XML,
- elementy potomne,
- kolejność występowania elementów potomnych,
- liczbę elementów potomnych,
- typy danych elementów i atrybutów,
- domyślne i wymagane wartości dla elementów i atrybutów,
- przestrzenie nazw (`namespaces`),
- tworzenie identyfikatorów i kluczy.

Dzięki XML Schema łatwiej jest:

- opisać dopuszczalną zawartość dokumentu,
- sprawdzić poprawność danych,
- pracować z danymi z bazy danych (definicje typów danych są podobne),
- określić aspekty danych (ograniczenia w typach danych),
- określić wzorce danych (format danych),
- dokonać konwersji pomiędzy różnymi typami danych.

3.5.1. Struktura schematu

Korzeniem w XML Schema jest element `schema`:

```
1 <?xml version="1.0"?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3    targetNamespace="http://www.example.com"
4    xmlns="http://www.example.com"
5    elementFormDefault="qualified"
6    attributeFormDefault="unqualified" >
7
```

```
 8   <!-- definicja elementow -->
 9
10 </xs:schema>
```

Element `schema`:

- jest zdefiniowany w przestrzeni `http://www.w3.org/2001/XMLSchema` (linia 2),
- definiuje przestrzeń nazw dla definiowanego schematu, za pomocą atrybutu `targetNamespace` (linia 3),
- określa czy lokalne elementy i atrybuty muszą być poprzedzone prefixem schematu (linia 5).

3.5.2. Definiowanie elementów

Elementy definiuje się za pomocą znacznika `element` wewnątrz korzenia `schema`:

```
1 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
2            targetNamespace="urn:books"
3            xmlns:bks="urn:books">
4   <xs:element name="books" type="bks:BooksForm"/>
5 </xs:schema>
```

Elementy zdefiniowane jako bezpośredni potomkowie korzenia `schema` nazywamy **elementami globalnymi**. Każdy z tych elementów może być korzeniem dokumentu zbudowanego w zgodzie z tym schematem.

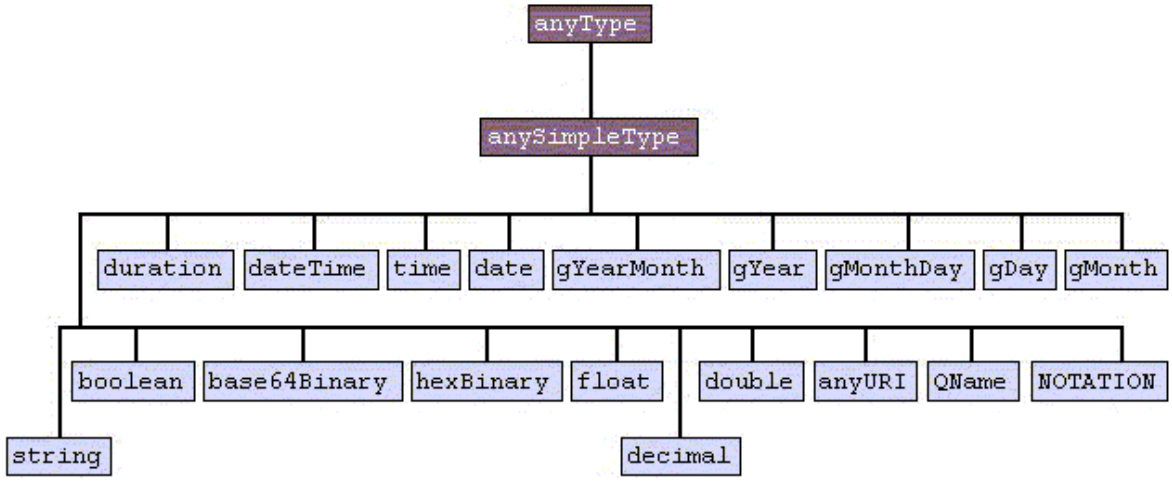
Atrybuty znacznika `element`:

Atrybut	Opis
`name`	nazwa elementu (będzie to nazwa znacznika w instancji dokumentu)
`type`	typ elementu — jeden z typów wbudowanych lub zdefiniowanych przez użytkownika
`minOccurs` / `maxOccurs`	definiuje mnogość danego elementu; domyślnie obie wartości ustawione są na 1
`ref`	wskazuje, że definiowany element jest tego samego typu co inny element; stosowany zamiennie z `type`

3.5.2.1. Typy wbudowane

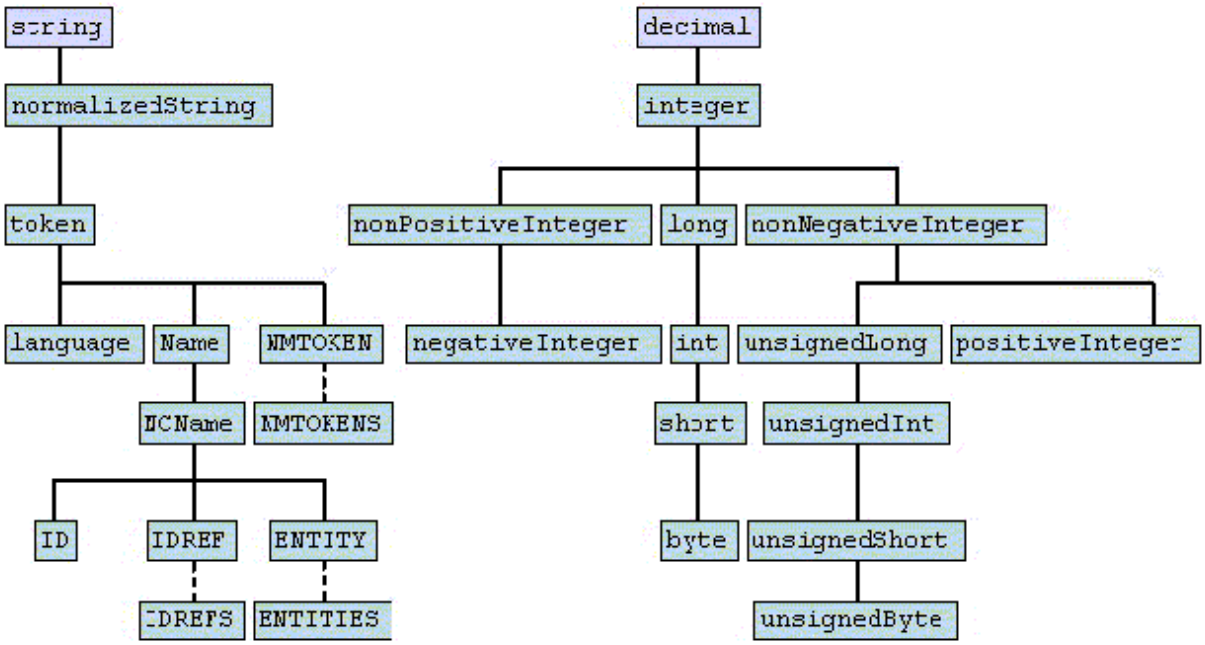
Typami wbudowanymi nazywamy takie typy, które zostały zdefiniowane w drugiej części specyfikacji XML Schema. Dzielimy je na dwie podgrupy:

Pierwotne — najprostsze typy, takie jak liczby, ciągi znaków czy daty:



Rysunek 6. Typy pierwotne XMLSchema

Pochodne — powstały na skutek rozszerzeń lub zawężeń typów `string` lub `decimal`, np. liczby całkowite lub dodatnie:



Rysunek 7. Typy pochodne XMLSchema

Poniżej kilka przykładów deklaracji elementów typów prostych:

```
1 <xs:element name="price"    type="xs:float" />
2 <xs:element name="pub_date" type="xs:date"   minOccurs="0" />
3 <xs:element name="review"   type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
```

`pub_date` jest opcjonalne, a elementy `review` mogą wystąpić w dowolnej liczbie.

Fragment dokumentu opisany takim schematem:

```
<price>44.95</price>
<!-- pominiety element pub_date -->
<review>An amazing story of nothing.</review>
<review>And cheep.</review>
```

3.5.2.2. Wartości domyślne, stałe i elementy puste

Domyślna wartość elementu typu prostego:

```
<xs:element name="color"   type="xs:string" default="red"/>
```

Stała wartość elementu (tylko jedna dozwolona wartość):

```
<xs:element name="version" type="xs:string" fixed="1.1"/>
```

Wartość pusta dla typów liczbowych lub dat (wymaga `nillable="true"` w schemacie):

```
<!-- w schemacie: -->
<xs:element name="shipdate" type="xs:date" nillable="true"/>

<!-- w dokumencie: -->
<shipdate xsi:nil="true"/>
```

3.5.2.3. Aspekty — ograniczenia typów prostych

XML Schema pozwala tworzyć własne typy pochodne na bazie typów prostych poprzez nakładanie ograniczeń. Ograniczenie typu odbywa się w elemencie `restriction`.

`enumeration` — lista dopuszczalnych wartości:

```
<xs:element name="car">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="Audi"/>
      <xs:enumeration value="Golf"/>
      <xs:enumeration value="BMW"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

`length` — dokładna liczba znaków:

```
<xs:element name="password">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:length value="8"/>
```

```
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Zakresy wartości liczbowych:

```
<xs:element name="age">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minExclusive value="0"/>
      <xs:maxExclusive value="120"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

`pattern` — wyrażenie regularne ograniczające wartości:

```
<xs:element name="password">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[a-zA-Z0-9]{8}"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

`whiteSpace` — sposób traktowania białych znaków (`preserve`, `replace`, `collapse`):

```
<xs:element name="address">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:whiteSpace value="preserve"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Dostępne aspekty ograniczeń:

Aspekt	Opis
`enumeration`	lista dozwolonych wartości
`length`	dokładna liczba znaków / elementów listy
`minLength`	minimalna liczba znaków / elementów
`maxLength`	maksymalna liczba znaków / elementów
`minInclusive`	dolna granica wartości (włącznie)
`maxInclusive`	górna granica wartości (włącznie)
`minExclusive`	dolna granica wartości (wyłącznie)

Aspekt	Opis
maxExclusive	górna granica wartości (wyłącznie)
pattern	wyrażenie regularne ograniczające wartości
totalDigits	całkowita liczba cyfr
fractionDigits	maksymalna liczba cyfr po przecinku
whiteSpace	sposób traktowania białych znaków: preserve, replace, collapse

3.5.2.4. Typy nazwane i anonimowe

Typ anonimowy jest zdefiniowany bezpośrednio w deklaracji elementu:

```
<xs:element name="address">
  <xs:simpleType>
    .....
  </xs:simpleType>
</xs:element>
```

Typ nazwany jest zdefiniowany poza deklaracją elementu i posiada przypisaną do niego nazwę:

```
<xs:simpleType name="AdressType">
   .....
</xs:simpleType>
<xs:element name="address"     type="AdressType"/>
<xs:element name="homeAddress" type="AdressType"/>
```

Typy nazwane deklarowane są raz i przeznaczone są do wielokrotnego użycia w ramach schematu. Możliwe jest również wielokrotne wykorzystanie elementów z typami anonimowymi, używając do tego celu atrybutu ref zamiast type.

3.5.2.5. Typy złożone

Użytkownik na bazie typów wbudowanych i innych typów złożonych może definiować swoje własne typy danych, które będą posiadały określone atrybuty i elementy. Takie typy definiuje się w elemencie complexType:

```
<xs:complexType name="BookForm">
  <xs:sequence>
    <xs:element name="author"   type="xs:string"/>
    <xs:element name="title"    type="xs:string"/>
    <xs:element name="genre"    type="xs:string"/>
    <xs:element name="price"    type="xs:float" />
    <xs:element name="pub_date" type="xs:date"   minOccurs="0" />
    <xs:element name="review"   type="xs:string"/>
  </xs:sequence>
```

```
10   <xs:attribute name="id" type="xs:string"/>
11 </xs:complexType>
```

Wewnątrz definicji typu znajduje się opis jego zawartości. W pierwszej kolejności znajdują się potomkowie elementu w elemencie wskaźnika porządku `sequence`, za nim znajduje się lista atrybutów.

3.5.2.6. Wskaźniki porządku

Wskaźniki porządku pozwalają na określenie w jaki sposób elementy potomne są rozłożone wewnątrz definiowanego typu. Wyróżniamy trzy wskaźniki porządku:

`sequence`
: Elementy muszą wystąpić dokładnie w kolejności zdefiniowanej w schemacie. Każdy element może być opcjonalny (jeśli `minOccurs` wynosi 0) lub powtarzalny (jeśli `maxOccurs` jest większe od 1), lecz kolejność grup elementów nie może być zmieniona.

`all`
: Oznacza, że elementy wymienione na liście mogą wystąpić co najwyżej raz i mogą być w dowolnej kolejności. Aby uczynić element opcjonalnym należy ustawić jego wartość `minOccurs` na 0. Wartość `maxOccurs` nie może być większa od 1.

`choice`
: Oznacza, że tylko jeden typ elementu z danej listy może pojawić się jako potomek. Jeśli jego `maxOccurs` jest większe od 1 może on się pojawić wielokrotnie.

3.5.2.7. Atrybuty

Po elementach potomnych definiowane są atrybuty. Wszystkie atrybuty są typu prostego:

```
1 <xs:complexType name="BookForm">
2   .....
3   <xs:attribute name="id"        type="xs:string" use="required"/>
4   <xs:attribute name="publisher" type="xs:string"/>
5 </xs:complexType>
```

Atrybuty nie mogą posiadać znaczników wielokrotności `minOccurs` i `maxOccurs` i domyślnie są opcjonalne. Można wymusić obligatoryjność wystąpienia atrybutu używając atrybutu `use` z wartością `required`. Atrybuty mogą mieć określoną wartość domyślną (`default`) lub stałą (`fixed`). Kolejność atrybutów w elemencie jest dowolna.

3.5.2.8. Rozszerzenia i ograniczenia typów złożonych

Możliwe jest stosowanie rozszerzeń i ograniczeń za pomocą elementu `complexContent` z

elementem `restriction` lub `extension`:

Rozszerzenie — `extension`

```
<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname"  type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="fullpersoninfo">
  <xs:complexContent>
    <xs:extension base="personinfo">
      <xs:sequence>
        <xs:element name="address" type="xs:string"/>
        <xs:element name="city"    type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Nowy typ *fullpersoninfo* ma wszystkie elementy z *personinfo* i następnie informacje o adresie. Dodatkowe elementy można umieścić tylko na końcu rozszerzanego elementu.

Ograniczenie — `restriction`

```
<xs:complexType name="NorwegianCustomer">
  <xs:complexContent>
    <xs:restriction base="customer">
      <xs:sequence>
        <xs:element name="firstname" type="xs:string"/>
        <xs:element name="lastname"  type="xs:string"/>
        <xs:element name="country"   type="xs:string" fixed="Norway"/>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
```

Użycie `restriction` wymusza powtórzenie wszystkich elementów, które chcemy zachować. Nie możemy dodać nowych elementów.

3.5.2.9. Elementy o mieszanej zawartości

Elementy z mieszaną zawartością mogą posiadać zarówno tekst jak i inne elementy. Element mieszany oznacza się za pomocą atrybutu `mixed` z wartością `true`:

```
<xs:element name="letter">
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:element name="name"    type="xs:string"/>
      <xs:element name="orderid" type="xs:positiveInteger"/>
      <xs:element name="shipdate" type="xs:date"/>
    </xs:sequence>
```

```
  </xs:complexType>
</xs:element>
```

```
<letter>
  Dear Mr.<name>John Smith</name>.
  Your order <orderid>1032</orderid>
  will be shipped on <shipdate>2013-07-13</shipdate>.
</letter>
```

3.5.3. Rozszerzalność za pomocą dowolnych elementów

Elementy XML mogą posiadać tylko taką zawartość jaka jest zadeklarowana przez ich typ. XMLSchema pozwala zaznaczyć, że dany element może posiadać elementy i atrybuty dowolnego typu za pomocą `any` i `anyAttribute`:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname"  type="xs:string"/>
      <xs:any minOccurs="0"
              processContents="lax"
              namespace="##any"/>
    </xs:sequence>
    <xs:anyAttribute processContents="strict"/>
  </xs:complexType>
</xs:element>
```

Atrybut `namespace` określa do jakiej przestrzeni nazw mają należeć elementy lub atrybuty:

- `##any` — z każdej przestrzeni nazw (domyślna wartość),
- `##other` — z każdej innej przestrzeni niż aktualnie definiowana,
- `##local` — elementy nie posiadające przestrzeni nazw,
- `##targetNamespace` — z aktualnej przestrzeni nazw,
- listy konkretnych przestrzeni nazw.

Atrybut `processContents` wskazuje jak walidacja powinna traktować obce elementy:

- `strict` — schematy dla dodatkowych elementów muszą być dostępne i musi nastąpić ich walidacja,
- `lax` — jeśli schemat jest dostępny, dodatkowe obiekty będą sprawdzane,
- `skip` — pomiń sprawdzanie.

3.5.4. Identyfikacja węzłów i tworzenie odnośników

3.5.4.1. Identyfikatory i referencje

Jednoznaczna identyfikacja węzłów w dokumencie XML może zostać wykonana za pomocą elementów lub atrybutów o typie `ID`. Żadne dwa elementy w obrębie jednego dokumentu nie mogą mieć tej samej wartości umieszczonej w elemencie lub atrybucie typu `ID`. Do referencji do innych elementów służy typ `IDREF` (jedna referencja) lub `IDREFS` (lista):

```
<xs:element name="employee">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname"   type="xs:string"/>
      <xs:element name="lastname"    type="xs:string"/>
      <xs:element name="employeeID"  type="xs:ID"/>
      <xs:element name="division"    type="xs:IDREF"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="division">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name"      type="xs:string"/>
      <xs:element name="employees" type="xs:IDREFS"/>
    </xs:sequence>
    <xs:attribute name="divisionID" type="xs:ID"/>
  </xs:complexType>
</xs:element>
```

Przykład dokumentu zgodnego z takim schematem:

```
<employee>
  <firstname>Jan</firstname>
  <lastname>Kowalski</lastname>
  <employeeID>E1</employeeID>
  <division>RD</division>
</employee>
<employee>
  <firstname>Julia</firstname>
  <lastname>Kowalska</lastname>
  <employeeID>E2</employeeID>
  <division>RD</division>
</employee>
<division divisionID="RD">
  <name>RaD</name>
  <employees>E1 E2</employees>
</division>
```

3.5.4.2. Klucze i operator unikalności

Mechanizm kluczy pozwala deklarować unikalność węzłów ze względu na wartości w określonych polach lub kombinacjach pól i atrybutów. Mechanizm ten jest bardzo podobny do tego spotykanego w relacyjnych bazach danych.

Definicja klucza:

```
<xs:key name="nazwaKlucza">
  <xs:selector xpath="pole"/>
  <xs:field    xpath="pole"/>
</xs:key>
```

Klucz definiuje się wewnątrz elementów i ma on zastosowanie tylko do podelementów tego konkretnego elementu. Podobnym operatorem do klucza jest operator `unique` — wymusza unikalność wartości w obrębie wskazanych pól, ale nie wymusza ich istnienia w każdym elemencie.

Do wskazania, że wartość w polu musi odpowiadać istniejącemu kluczowi, służy `keyref`:

```
<xs:keyref name="nazwaReferencji" refer="nazwaKlucza">
  <xs:selector xpath="pole"/>
  <xs:field    xpath="pole"/>
</xs:keyref>
```

Pełny przykład ze schematem firmy

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="employee">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="firstname" type="xs:string"/>
        <xs:element name="lastname"  type="xs:string"/>
      </xs:sequence>
      <xs:attribute name="ID" type="xs:integer"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="division">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name"       type="xs:string"/>
        <xs:element name="city"       type="xs:string"/>
        <xs:element name="employeeID" type="xs:integer"
                    maxOccurs="unbounded" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
    <xs:unique name="employeeUnique">
      <xs:selector xpath="employeeID"/>
      <xs:field    xpath="."/>
    </xs:unique>
  </xs:element>

  <xs:element name="company">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="employee" maxOccurs="unbounded"/>
        <xs:element ref="division" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
```

```
35     <xs:key name="employeeKey">
36       <xs:selector xpath="employee"/>
37       <xs:field    xpath="@ID"/>
38     </xs:key>
39     <xs:key name="divisionKey">
40       <xs:selector xpath="division"/>
41       <xs:field    xpath="name"/>
42       <xs:field    xpath="city"/>
43     </xs:key>
44     <xs:keyref name="employeeKeyRef" refer="employeeKey">
45       <xs:selector xpath="division/employeeID"/>
46       <xs:field    xpath="."/>
47     </xs:keyref>
48   </xs:element>
49
50 </xs:schema>
```

Przykładowy dokument zgodny z powyższym schematem

```
1 <company>
2   <employee ID="1">
3     <firstname>Jan</firstname>
4     <lastname>Kowalski</lastname>
5   </employee>
6   <employee ID="2">
7     <firstname>Artur</firstname>
8     <lastname>Dabrowski</lastname>
9   </employee>
10  <employee ID="3">
11    <firstname>Julia</firstname>
12    <lastname>Kowalska</lastname>
13  </employee>
14  <division>
15    <name>RaD</name>
16    <city>Krakow</city>
17    <employeeID>1</employeeID>
18    <employeeID>2</employeeID>
19  </division>
20  <division>
21    <name>RaD</name>
22    <city>Warszawa</city>
23    <employeeID>3</employeeID>
24  </division>
25  <division>
26    <name>HR</name>
27    <city>Warszawa</city>
28    <employeeID>3</employeeID>
29  </division>
30 </company>
```

Klucz `employeeKey` określa, że w obrębie elementu `company` nie może być dwóch pracowników o takim samym atrybucie `ID`. Klucz `divisionKey` nakłada ograniczenie na oddziały — w jednym mieście nie mogą istnieć oddziały o takiej samej nazwie. `employeeUnique` w elemencie `division` zapewnia, że ten sam pracownik nie zostanie dodany do oddziału dwa razy, ale pozwala przypisać go do dwóch różnych oddziałów. `keyref` (`employeeKeyRef`) gwarantuje, że w polach `employeeID` nie wstawimy identyfikatora nieistniejącego pracownika.

3.6. XPath — język zapytań do dokumentów XML

XML Path Language (XPath) jest językiem wyrażeń do adresowania fragmentów dokumentu XML. Jest używany przez XSLT, klauzule `selector`/`field` w kluczach XMLSchema, wyrażenia `SignedElements`/`EncryptedElements` w WS-SecurityPolicy oraz filtry StAX.

Dokument XML traktowany jest jako drzewo **węzłów** (*nodes*): elementów, atrybutów, tekstu, przestrzeni nazw, komentarzy i instrukcji procesowych.

3.6.1. Podstawowe ścieżki

Wyrażenie XPath zwraca zbiór węzłów, wartość logiczną, liczbę lub ciąg znaków. Ścieżka lokalizacyjna (*location path*) składa się z kroków oddzielonych `/`:

```
1 /company/employee          <!-- elementy employee bezpośrednio pod company (od korzenia) -->
2 company/employee           <!-- to samo, ścieżka względna -->
3 //employee                 <!-- wszystkie elementy employee w dokumencie -->
4 /company/employee[1]       <!-- pierwszy element employee -->
5 /company/employee[@ID=1]   <!-- employee z atrybutem ID równym 1 -->
6 /company/division/name     <!-- element name wewnątrz division -->
```

3.6.2. Predykaty

Predykat filtruje węzły i umieszczany jest w nawiasach kwadratowych `[]`:

Predykat	Znaczenie
`[1]`	Pierwszy węzeł w zbiorze (XPath indeksuje od 1)
`[last()]`	Ostatni węzeł
`[@id]`	Węzły posiadające atrybut `id`
`[@id='E1']`	Węzły z atrybutem `id` o wartości `E1`
`[price>10]`	Węzły, których podelementwy `price` mają wartość > 10

3.6.3. Osie (Axes)

Oś określa relację między węzłem kontekstu a adresowanymi węzłami:

Oś	Znaczenie
`child::` (domyślna)	Bezpośredni potomkowie
`attribute::` (skrót `@`)	Atrybuty węzła
`parent::`	Węzeł nadrzędny

Oś	Znaczenie
`ancestor::`	Wszyscy przodkowie
`descendant::`	Wszyscy potomkowie
`following-sibling::`	Rodzeństwo po węźle kontekstu
`self::` (skrót `.`)	Sam węzeł kontekstu
`text()`	Węzły tekstowe

Skrótowe notacje: `@attr` zamiast `attribute::attr`, `.` zamiast `self::node()`, `..` zamiast `parent::node()`, `//` zamiast `/descendant-or-self::node()/`.

3.6.4. Funkcje wbudowane

Wybrane funkcje używane w schematach i politykach:

```
count(//employee)        <!-- liczba elementów employee -->
string(@ID)              <!-- wartość atrybutu ID jako tekst -->
normalize-space(name)    <!-- tekst bez zbędnych białych znaków -->
contains(text(), 'Jan')  <!-- true jeśli tekst zawiera 'Jan' -->
not(@optional)           <!-- true jeśli atrybut optional nie istnieje -->
```

3.6.5. XPath w XMLSchema — klucze i selektory

W deklaracjach `key`/`keyref`/`unique` (opisanych wyżej) wyrażenia XPath wskazują zakres i pola klucza:

```
<xs:key name="employeeKey">
    <xs:selector xpath="employee"/>     <!-- wybiera elementy employee -->
    <xs:field    xpath="@ID"/>          <!-- wartość klucza: atrybut ID -->
</xs:key>
```

`selector` może używać wyłącznie uproszczonego podzbioru XPath (bez osi `ancestor`, funkcji itp.).

3.7. XML a stos technologiczny web serwisów

Wiedza zdobyta w niniejszym rozdziale jest bezpośrednio wykorzystywana we wszystkich kolejnych:

- **SOAP** — koperta, nagłówek i ciało wiadomości to elementy XML; przestrzenie nazw (m.in. `http://schemas.xmlsoap.org/soap/envelope/`) oddzielają elementy protokołu od treści,
- **WSDL** — dokument XML opisujący interfejs usługi; typy danych parametrów i odpowiedzi definiuje XMLSchema osadzona wewnątrz WSDL,

- **JAXB** — mechanizm wiązania dokumentów XML ze strukturami Java oparty na XMLSchema; generuje klasy Javy odpowiadające typom zdefiniowanym w schemacie,
- **JAXP** — niskopoziomowe API do parsowania i transformacji dokumentów XML, używane wewnątrz handlerów JAX-WS oraz w interfejsie SAAJ.

Kolejne rozdziały zakładają swobodne posługiwanie się pojęciami *well-formed*, *valid*, przestrzenie nazw oraz typy proste i złożone XMLSchema.

Część II: REST

4. Usługi sieciowe w stylu REST

REST (REpresentational State Transfer) jest lżejszym z dwóch głównych stylów web serwisów omówionych we wcześniejszym rozdziale. Podczas gdy SOAP definiuje szczegółowy protokół komunikacyjny nałożony na HTTP, REST opiera się bezpośrednio na semantyce HTTP — bez dodatkowej warstwy protokołu. Zasób może być serwowany w dowolnym formacie (XML, JSON, HTML) — wybór odbywa się przez negocjację reprezentacji za pomocą nagłówków HTTP.

4.1. Ograniczenia architektoniczne REST

Skrót REST pochodzi od *REpresentational State Transfer* — terminu użytego przez Roya Fieldinga w jego pracy doktorskiej z 2000 r. Fielding zdefiniował REST jako zestaw sześciu ograniczeń architektonicznych (*constraints*), których łączne spełnienie nadaje systemowi pożądane cechy: skalowalność, niezawodność i możliwość niezależnej ewolucji komponentów:

Klient-serwer (*client-server*)

Wyraźne rozdzielenie odpowiedzialności między klientem (logika prezentacji, UI) a serwerem (zasoby, dane, logika biznesowa). Obie strony mogą ewoluować niezależnie, o ile respektują kontrakt interfejsu.

Bezstanowość (*statelessness*)

Każde żądanie musi zawierać wszystkie informacje niezbędne do jego obsługi. Serwer nie przechowuje kontekstu sesji klienta między żądaniami. Zwalnia to serwer z zarządzania stanem sesji, poprawia skalowalność i umożliwia kierowanie żądań do dowolnego węzła klastra.

Cache'owalność (*cacheability*)

Każda odpowiedź musi jawnie określać, czy może być zapamiętana przez klienta lub pośrednika. Poprawna kontrola cachowania ogranicza obciążenie serwera i poprawia czas odpowiedzi.

Jednolity interfejs (*uniform interface*)

Zasoby identyfikowane przez URI; operacje wyrażone standardowymi metodami HTTP; wiadomości samoopisujące się (nagłówki `Content-Type`, `Accept`); HATEOAS — odpowiedź zawiera linki do możliwych następnych operacji. To ograniczenie jest centralną, wyróżniającą cechą REST.

System warstwowy (*layered system*)

Klient nie wie, czy łączy się bezpośrednio z serwerem końcowym, czy z pośrednikiem (proxy, load balancer, CDN, brama API). Pośrednicy działają transparentnie i mogą realizować cachowanie, kompresję lub uwierzytelnianie.

Kod na żądanie (*code-on-demand*) — opcjonalne

Serwer może przesyłać wykonywalny kod do klienta (JavaScript, aplety). Jedyne nieobowiązkowe ograniczenie w modelu Fieldinga.

Choć REST nie jest jawnie związany z konkretnym protokołem, w praktyce realizowany jest wyłącznie z użyciem HTTP[1] — HTTP implementuje wszystkie wymienione ograniczenia bezpośrednio w swoim protokole.

4.2. Podstawowe elementy REST

> *A resource should have an associated URI if another party might reasonably want to create a hypertext link to it, make or refute assertions about it, retrieve or cache a representation of it, include all or part of it by reference into another representation, annotate it, or perform other operations on it.*
>
> — The Architecture of the World Wide Web, W3C Working Group

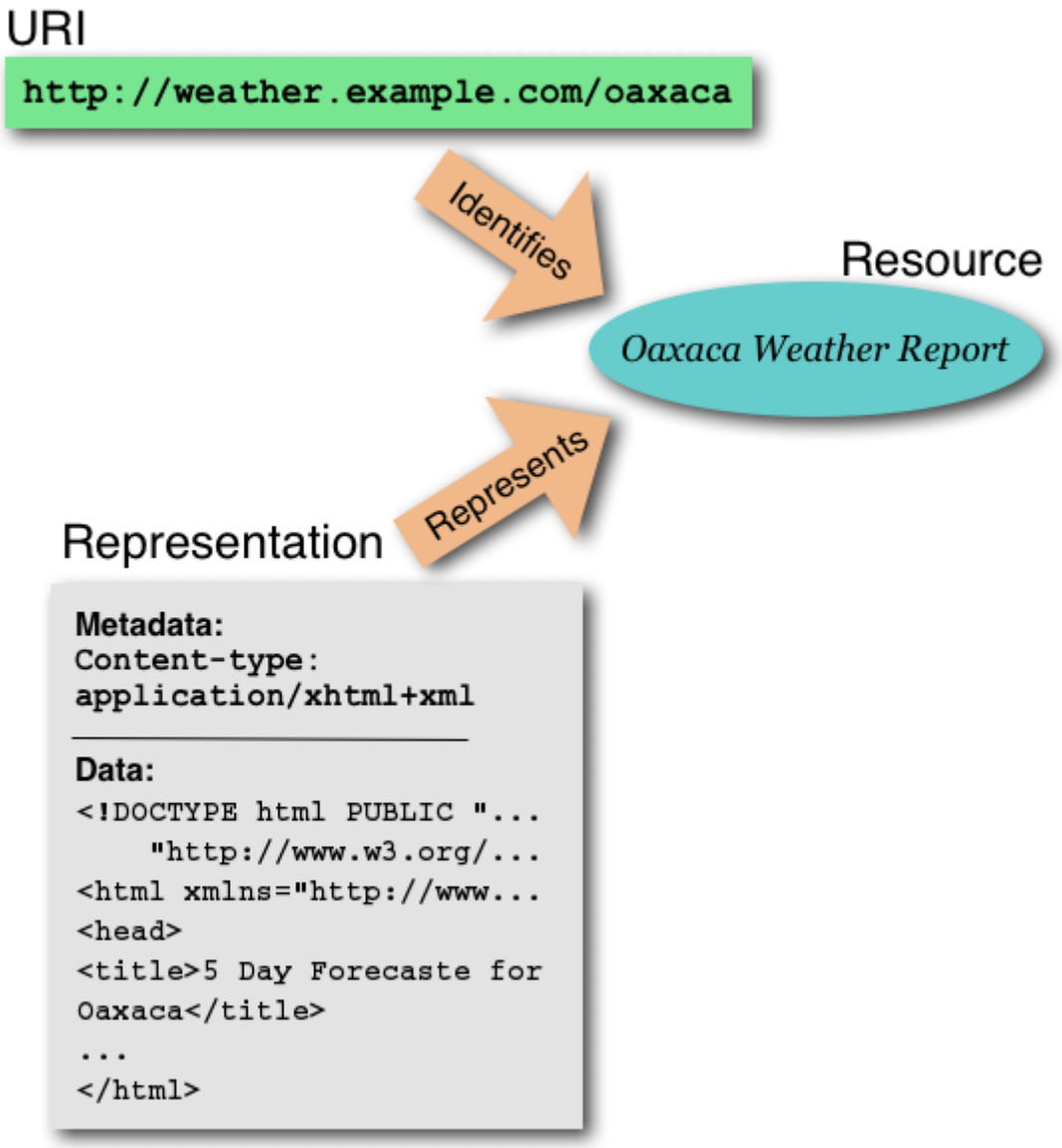
4.2.1. Zasób (Resource)

Podstawową jednostką w REST jest **zasób** — abstrakcyjny twór wskazujący pewną informację, która może być nazwana i umieszczona pod własnym adresem (URI). Zasobem może być wszystko: dokument, obraz, osoba lub pogoda w Warszawie.

Informacja, na którą dany zasób wskazuje nie musi być stała w czasie, może nawet nie istnieć. Przykład: „pogoda na jutro w Warszawie" jednego dnia wskazuje na palące słońce, innego na rzęsisty deszcz — w obu przypadkach jest to ten sam zasób pod tym samym adresem.

4.2.2. Reprezentacja

Fizycznym obrazem zasobu jest **reprezentacja**. Jeden zasób może posiadać wiele reprezentacji, np. XML, HTML, BMP, PDF. Reprezentacja jest zbiorem informacji wskazywanym przez zasób w konkretnym czasie i zapisanych w konkretnym formacie. Dodatkowym elementem reprezentacji są metadane: opis formatu, czas utworzenia, nagłówki HTTP.

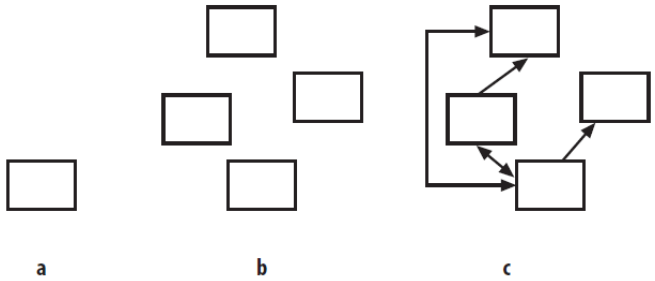


Rysunek 8. URI, zasób i reprezentacja

4.2.3. Bezstanowość — podział stanu

Resource state (serwer) — zadaniem serwera jest obsługa jego własnych zasobów, a nie interesowanie się „stanem umysłu" jego klientów.

Application state (klient) — klient jest odpowiedzialny za swój stan. On ma wiedzieć gdzie jest i czego chce. Każde żądanie do serwera musi zawierać wszystkie informacje konieczne do wykonania żądania.



Rysunek 9. Adresowalność i połączeniowość komponentów

(a) RPC-style — wszystkie operacje pod jednym adresem; nie jest adresowalny ani połączony.
(b) Adresowalny, lecz nie połączony — brak informacji o wzajemnych relacjach zasobów.
(c) Adresowalny i połączony — prawdziwy REST.

4.2.4. Budowa URI

```
1 scheme://host:port/path?queryString#fragment
```

- `scheme` — protokół komunikacyjny, np. `http`, `ftp`, `ldap`, `mailto`,
- `host` — adres docelowy (nazwa DNS lub IP),
- `port` — konkretny proces na docelowej maszynie,
- `path` — ścieżka lokalizująca zasób, segmenty oddzielone `/`.

URL jest podzbiorem adresów URI — poza identyfikacją zasobu określają jego lokalizację.

4.3. REST w połączeniu z HTTP

Przy budowaniu serwisów REST opartych na protokole HTTP otrzymujemy system:

- **skalowalny** — dziedziczy skalowalność po drzewiastej strukturze URL,
- **z ogólnym interfejsem** — metody HTTP działające na obiektach pod adresami URL,
- **warstwowy** — dzięki bramom, proxy i firewallom łatwo budujemy hierarchiczne systemy,
- **wydajny** — cache pozwalają na zapamiętanie i ponowne użycie niezmieniających się elementów.

4.3.1. HATEOAS

Hypermedia As The Engine Of Application State (HATEOAS) — zasada architektoniczna mówiąca, że serwis wraz z reprezentacją dostarcza informacji, w jaki sposób można kontynuować interakcję z nim. Klient nie musi znać struktury URL z góry — odkrywa dostępne operacje z odpowiedzi serwera. Przykładem implementacji

HATEOAS jest standard *Atom link*, opisany w dalszej części rozdziału.

4.3.2. Cachowanie

Nagłówek `Cache-Control` kontroluje cachowanie przez klienta i pośredników:

Wartość	Opis
`private` / `public`	Wiadomości publiczne mogą być przechowywane przez wszystkich pośredników
`no-cache`	Wiadomość nie powinna być cachowana
`no-store`	Zabrania zapisywania na dyskach
`no-transform`	Dane nie powinny być przekształcane przez pośredników
`max-age`	Czas (w sekundach) ważności przesłanych danych

```
1 HTTP/1.1 200 OK
2 Content-Type: application/xml
3 Cache-Control: private, no-store, max-age=300
4 <customers>...</customers>
```

4.3.3. Serwisy RESTful

Serwis spełniający zasady REST (*RESTful*):

- identyfikuje każdy zasób za pomocą unikalnego URI,
- realizuje operacje na zasobach metodami HTTP: `GET`, `POST`, `PUT`, `DELETE`, `PATCH`,
- opisuje interfejs za pomocą WADL (*Web Application Description Language*) lub RSDL,
- może być opisany w WSDL 2.0, jeśli używa formatu POX (*Plain Old XML*).

4.3.4. Jednolity interfejs — Akcje HTTP

W serwisach RESTful wszystko jest zasobem reprezentowanym przez rzeczowniki w URLach:

```
http://example.com/users/michal
```

Prawidłowo zaprojektowany REST udostępnia operacje poprzez zasoby, np. zamiast: `http://www.bookstore.com/action/query?t=books&subject=computers/eclipse` mamy „listę książek o danej tematyce": `http://www.bookstore.com/books/?subject=computers/eclipse`

Operacje reprezentowane przez czasowniki modelu **CRUD**[2]:

Metoda	Typ	Opis
GET	safe, idempotent	Pobieranie reprezentacji zasobu — nie może zmieniać stanu
PUT	idempotent	Aktualizacja zasobu (lub utworzenie pod podanym URI)
POST	—	Tworzenie zasobu podrzędnego lub wywoływanie akcji
DELETE	idempotent	Usunięcie zasobu
PATCH	—	Częściowa aktualizacja zasobu

Metoda GET — przykład

```
1 GET /students/Michal HTTP/1.1
2 Host: example.com
3 Date: Sat, 28 Feb 2013
4 Accept: text/xml
```

```
 1 HTTP/1.1 200 OK
 2 Date: Sat, 28 Feb 2013
 3 Content-Type: text/xml; charset=UTF-8
 4
 5 <xml>
 6     <student>
 7         <name>Michal</name>
 8         <age>25</age>
 9     </student>
10 </xml>
```

Metoda PUT — przykład

```
 1 PUT /students/Michal HTTP/1.1
 2 Host: example.com
 3 Accept: text/xml
 4 Content-Type: text/xml
 5
 6 <xml>
 7    <student>
 8       <name>Michal</name>
 9       <age>30</age>
10    </student>
11 </xml>
```

```
1 HTTP/1.1 204 No Content
2 Date: Sat, 28 Feb 2013
3 Server: Apache/1.3.3.7
4 Connection: close
```

Nagłówek `Content-Type` w żądaniu wskazuje format przesyłanego payloadu. Pomimo, że aktualizowany jest tylko wiek, PUT wymaga przesłania całej reprezentacji.

Metoda PATCH — przykład (częściowa aktualizacja)

```
1 PATCH /students/Michal HTTP/1.1
2 Host: example.com
3 Content-Type: text/xml
4
5 <xml>
6    <student>
7       <age>30</age>
8    </student>
9 </xml>
```

W przeciwieństwie do PUT, PATCH pozwala przesłać tylko te pola, które mają zostać zaktualizowane. W ciele mogą też wystąpić polecenia:

```
1 PATCH /students/Michal HTTP/1.1
2 Host: example.com
3 Content-Type: text/xml
4
5 <update-age>30</update-age>
```

Nagłówek `Accept` w żądaniu i `Content-Type` w odpowiedzi realizują **negocjację reprezentacji** (*content negotiation*) — klient deklaruje akceptowane formaty (np. `text/xml`, `application/json`), serwer wybiera i informuje o wybranym. Ten sam zasób `/students/Michal` może być zwracany jako XML lub JSON zależnie od preferencji klienta.

Tabela 2. Porównanie POST i PUT

Zasób	PUT (nowy adres)	PUT (istniejący adres)	POST
`/weblogs`	zabronione	brak efektu	tworzy nowy blog
`/weblogs/myweblog`	tworzy myweblog	modyfikuje myweblog	tworzy nowy wpis
`/weblogs/myweblog/entries/1`	zabronione	modyfikuje wpis	dodaje komentarz

Metoda DELETE

```
1 DELETE /students/Michal HTTP/1.1
2 Host: example.com
```

Usunięcie zasobu może zostać odroczone w czasie. Możliwe odpowiedzi: `200 OK` (z opcjonalną reprezentacją usuniętego zasobu), `204 No Content` (bez ciała) lub `202 Accepted`

(operacja przyjęta, realizowana asynchronicznie).

4.3.5. Atom link

Popularnym standardem dodawania odnośników w dokumentach XML jest typ *Atom link* ze standardu *Atom* (następcy RSS):

```
<customers>
    <link rel="next"
          href="http://example.com/customers?start=2&amp;size=1"
          type="application/xml"/>
    <customer id="123">
        <name>Bill Burke</name>
    </customer>
</customers>
```

Element `link` zawiera atrybuty:

- `rel` — relacja linku w stosunku do dokumentu (opis linku),
- `href` — URI pod którym można wykonać operację,
- `type` — typ MIME dokumentu wskazywanego przez link,
- `hreflang` — (opcjonalny) język dokumentu.

4.3.6. Rozszerzenia REST

Przykłady styli budowanych na bazie REST:

- **Resource-oriented architecture (ROA)** — 4 pojęcia: zasoby, nazwy (URI), reprezentacje, połączenia; 4 właściwości: adresowalność, bezstanowość, spójność, jednolitość interfejsu,
- **RESTful Objects** — serwisy do pracy na Domain Object Model: reprezentacja oparta na JSON, pełna standaryzacja URLi i struktury wiadomości; implementacje: Apache Isis, Restful Objects for .NET,
- **Resource-oriented computing (ROC)** — architektura oparta o zasoby niezależna od protokołu.

4.4. Zalety i wady podejścia REST

Zalety

Prostota

klienci nie używają specjalizowanego API, używają standardu HTTP, REST jest łatwy w zrozumieniu i przetwarzaniu.

Spójność

URI jest spójną strategią identyfikowania zasobów, HTTP to spójny interfejs operacji

dozwolonych na zasobach.

Skalowalność

serwisy REST w łatwy sposób mogą zostać rozproszone na różne komponenty, systemy i maszyny.

Wady

- brak jednego spójnego standardu definiującego serwisy REST — implementacje mogą się różnić,
- przesyłanie pełnych reprezentacji przy każdym żądaniu może być nieoptymalne dla dużych zasobów,
- brakuje wbudowanych standardów biznesowych: transakcji, bezpieczeństwa na poziomie wiadomości, niezawodnego dostarczania, koordynacji rozproszonych operacji,
- klient może być zmuszony do implementowania części logiki biznesowej po swojej stronie.

4.5. REST a implementacja w Javie

REST jest stylem architektonicznym — opisuje zasady, nie narzuca implementacji. Na platformie Java EE realizację tych zasad zapewnia specyfikacja **JAX-RS** (*Java API for RESTful Web Services*), omówiona w następnym rozdziale. JAX-RS dostarcza adnotacje pozwalające deklaratywnie odwzorować klasy Javy na zasoby REST (`@Path`, `@GET`, `@POST`, `@Produces`, `@Consumes`) oraz mechanizmy automatycznej konwersji między reprezentacjami (XML, JSON) a obiektami Javy — bez ręcznego parsowania.

[1] Roy Fielding pracował nad binarnym protokołem *Waka*, który miał zastępować HTTP i spełniać założenia REST.

[2] Create, Retrieve, Update, Delete

5. Java API for RESTful Web Services — JAX-RS

JAX-RS jest odpowiedzią platformy Java EE na potrzebę standaryzacji tworzenia serwisów REST. Zamiast ręcznego parsowania żądań HTTP w serwletach, JAX-RS pozwala deklaratywnie odwzorować klasy Javy na zasoby REST — za pomocą adnotacji, bez implementowania interfejsów i bez rozbudowanej konfiguracji XML.

5.1. Wprowadzenie do JAX-RS

JAX-RS jest specyfikacją narzędzi oraz API dostępnych w języku Java przeznaczonych do tworzenia web serwisów zgodnych ze stylem *REST*. Wersja 1.0 rozwijana była jako *Java Community Process* pod numerem *JSR-311*, wersja 1.1 stała się częścią Javy EE 6. Wersja 2.0 (JSR-339) jest dostępna od 2013 roku.

Podstawowe założenia API:

Użycie POJO

JAX-RS dostarcza zestawu adnotacji i interfejsów przeznaczonych do użycia z czystymi klasami Javy, czyniąc z nich web serwisy.

Ukierunkowanie na HTTP

Specyfikacja zakłada HTTP(S) jako protokół komunikacyjny. Adnotacje są ściśle powiązane z HTTP i URI.

Dowolność formatu

API dostarcza mechanizmu wtyczek (providerów) pozwalających na obsługę dowolnych formatów.

Część Java EE

Pozwala na integrację z innymi funkcjami Java EE.

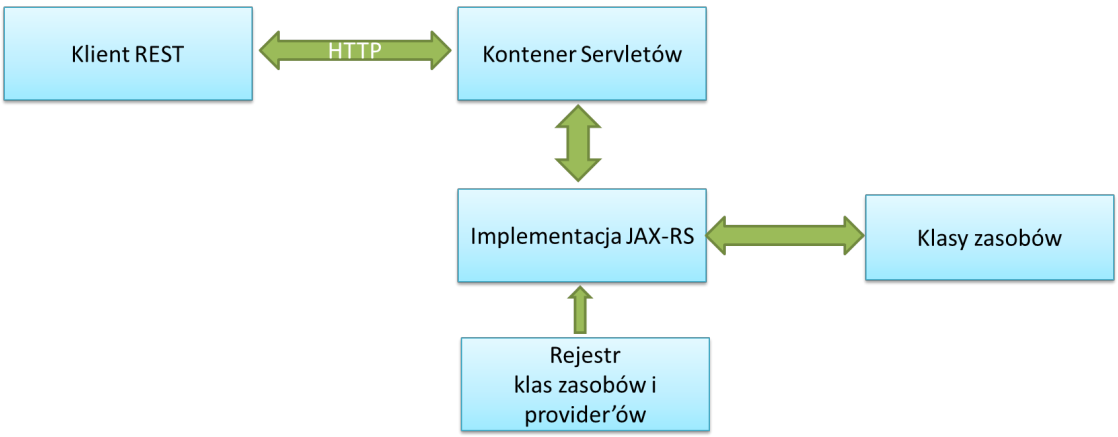
Niezależność od kontenera

Aplikacja JAX-RS może być umieszczona na serwerach z kontenerem serwletów lub jako ziarna EJB.

Wersja 1.x nie definiuje API strony klienckiej — zostało ono wprowadzone dopiero w wersji 2.0.

5.2. Ogólna architektura

Aplikacja w JAX-RS składa się z klas zasobów i klas providerów. Konfiguracji dokonuje się poprzez rozszerzenie klasy `javax.ws.rs.core.Application` i ustawienie jej jako wartość parametru `javax.ws.rs.Application` serwletu dostawcy. Domyślna implementacja skanuje wskazane pakiety w poszukiwaniu adnotowanych klas.



Rysunek 10. Architektura JAX-RS

Klient komunikuje się z serwerem przez HTTP. Żądanie obsługiwane jest przez kontener webowy, który przesyła je do serwletu dostawcy. Następnie zostaje odnaleziona konkretna klasa i metoda zasobu. Jeśli to konieczne, provider dokonuje konwersji payloadu do klasy Javy i następuje wywołanie metody. Wartość zwrócona zostaje przekonwertowana do wymaganej reprezentacji i zwrócona klientowi.

5.3. Klasa zasobu

W JAX-RS zasoby REST implementowane są jako **klasy zasobów**, a konkretne żądania obsługiwane są przez metody tych klas. Jedna klasa zasobu może obsługiwać wiele zasobów REST.

POJO staje się klasą zasobu jeśli:

- klasa lub którakolwiek z metod adnotowana jest za pomocą `@Path`, LUB
- przynajmniej jedna metoda ma adnotację określającą metodę HTTP (`@GET`, `@PUT`, `@POST`, ...).

```
1 import javax.ws.rs.GET;
2 import javax.ws.rs.Path;
3 import javax.ws.rs.core.Response;
4
5 @Path("/message")
6 public class MessageRestService {
7
8     @GET
9     @Path("/hello")
10    public Response printMessage() {
11        String result = "Hello World";
12        return Response.status(200).entity(result).build();
13    }
14 }
```

Klasa `MessageRestService` obsługuje zasoby, których URI zaczyna się od `/message`. Metoda `printMessage` obsługuje żądania `GET` pod adresem `/message/hello`.

5.3.1. Cykl życia zasobu w JAX-RS

Domyślnie każda klasa zasobu tworzona jest do obsługi pojedynczego żądania (*per-request*). Po przyjęciu żądania JAX-RS:

1. Wywołuje konstruktor,
2. Wstrzykuje zależności,
3. Wywołuje metodę,
4. Odsyła odpowiedź,
5. Zwalnia klasę zasobu.

W kontenerze webowym dostępny jest tylko tryb *per-request*. Użycie kontenera EJB pozwala na `@Singleton`, `@Stateless`, `@Stateful`. Alternatywą jest rejestracja w klasie `Application`.

5.3.2. Metody zasobu

Metoda zasobu to publiczna metoda posiadająca adnotację wskaźnika metody HTTP (*request method designator*) — adnotację z adnotacją `@HttpMethod`. Pojedyncza metoda może posiadać tylko jedną taką adnotację.

JAX-RS 1.x definiuje: `@GET`, `@POST`, `@PUT`, `@DELETE`, `@HEAD`, `@OPTIONS`. JAX-RS 2.0 dodaje `@PATCH`.

Egzamin 1Z0-897 oparty jest na Java EE 6 (JAX-RS 1.1), w którym adnotacja `@PATCH` **nie istnieje**. Aby obsłużyć metodę HTTP PATCH w JAX-RS 1.1, należy zdefiniować własną adnotację używając meta-adnotacji `@HttpMethod`:

```
1 import javax.ws.rs.HttpMethod;
2 import java.lang.annotation.*;
3
4 @Target({ElementType.METHOD})
5 @Retention(RetentionPolicy.RUNTIME)
6 @HttpMethod("PATCH")
7 public @interface PATCH { }
```

Po takiej deklaracji adnotacja `@PATCH` może być stosowana jak wbudowane `@GET`/`@POST`:

```
1 @PATCH
2 @Path("{id}")
3 @Consumes(MediaType.APPLICATION_XML)
4 public Response partialUpdate(@PathParam("id") int id, Student partial) { ... }
```

Używając `@HttpMethod` programista może definiować własne wskaźniki dla dowolnych metod HTTP.

Co najwyżej jeden parametr (*entity parameter*) może nie posiadać adnotacji — jest mapowany na *payload* żądania. Pozostałe muszą mieć jedną z adnotacji:

Adnotacja	Opis
`@PathParam`	Wartość segmentu URI wskazanego przez `{placeholder}`
`@QueryParam`	Wartość z *query string*
`@HeaderParam`	Wartość konkretnego nagłówka HTTP
`@MatrixParam`	Wartości macierzy z ostatniego segmentu URI
`@CookieParam`	Wartość z ciasteczka
`@FormParam`	Wartość z formularza (`application/x-www-form-urlencoded`)
`@Context`	Wstrzyknięcie obiektów JAX-RS (`UriInfo`, `HttpHeaders`, `Request`, ...)

Jeśli klasa zasobu ma zasięg *per-request*, powyższe adnotacje można stosować również na parametrach konstruktora i polach klasy — zostaną wstrzyknięte przed wywołaniem metody.

5.3.3. Typy obsługiwane przez adnotacje parametrów

Adnotacje `@PathParam`, `@QueryParam`, `@HeaderParam`, `@MatrixParam`, `@CookieParam` i `@FormParam` obsługują następujące typy Javy:

- typy prymitywne i `String`,
- typy posiadające konstruktor przyjmujący jeden `String`,
- typy posiadające statyczną metodę `valueOf` lub `fromString` przyjmującą jeden `String` (np. `enum`),
- `List<T>`, `Set<T>` lub `SortedSet<T>`, gdzie `T` jest jednym z powyższych.

Błąd podczas konwersji (np. do typu prymitywnego) traktowany jest jako błąd po stronie klienta (`400 Bad Request`).

Jako typ `@CookieParam` można użyć `javax.ws.rs.core.Cookie`, który poza wartością daje dostęp do kontekstowych informacji:

```
1 public class Cookie {
2     public String getName()    { ... }
3     public String getValue()   { ... }
4     public int    getVersion() { ... }
5     public String getDomain()  { ... }
6     public String getPath()    { ... }
7 }
```

5.3.4. Typy parametru ciała żądania

Co najwyżej jeden parametr metody może być *entity parameter* (brak adnotacji) — mapuje się na payload żądania. Obsługiwane typy:

Typ	Zastosowanie
`java.io.InputStream`, `java.io.Reader`	Ogólne strumienie — wszystkie typy danych, w tym binarne
`byte[]`	Dane binarne mniejszych rozmiarów
`String`, `char[]`	Dane tekstowe
`MultivaluedMap<String,String>`	Formularze (`application/x-www-form-urlencoded`), alternatywa dla `@FormParam`
`javax.xml.transform.Source`	Niskopoziomowe przetwarzanie XML
`java.io.File`	JAX-RS zapisuje payload jako plik tymczasowy i przekazuje referencję

5.3.5. Typy wartości zwracanej przez metodę zasobu

Typ	Zachowanie
`void`	Brak ciała odpowiedzi; JAX-RS ustawia `204 No Content`
`javax.ws.rs.core.Response`	Pełna kontrola: status, nagłówki, ciało; brak ciała → `204`, z ciałem → `200`
`javax.ws.rs.core.GenericEntity<T>`	Holder zachowujący informację o typie generycznym (zazwyczaj traconą przez erasure)
`javax.ws.rs.core.StreamingOutput`	Callback z metodą `write(OutputStream)` — pisze bezpośrednio do strumienia odpowiedzi
dowolny obiekt	Konwertowany przez `MessageBodyWriter`; brak pasującego writera → `500`

`Response` tworzony jest wzorcem budowniczego. Wybrane statyczne metody fabrykujące klasy `Response`:

```
public abstract class Response {
    public static ResponseBuilder ok();
    public static ResponseBuilder ok(Object entity);
    public static ResponseBuilder ok(Object entity, MediaType type);
    public static ResponseBuilder status(int status);
    public static ResponseBuilder status(Status status);
    public static ResponseBuilder created(URI location);   // 201
    public static ResponseBuilder noContent();             // 204
    public static ResponseBuilder fromResponse(Response response);
    public static ResponseBuilder seeOther(URI location);      // 303
```

```
11     public static ResponseBuilder temporaryRedirect(URI location); // 307
12 }
```

Wybrane metody `Response.ResponseBuilder`:

```
1 public static class ResponseBuilder {
2     public ResponseBuilder status(int i);
3     public ResponseBuilder entity(Object o);
4     public ResponseBuilder type(MediaType mt);
5     public ResponseBuilder type(String string);
6     public ResponseBuilder language(String string);
7     public ResponseBuilder location(URI uri);
8     public ResponseBuilder contentLocation(URI uri);
9     public ResponseBuilder tag(EntityTag et);
10    public ResponseBuilder lastModified(Date date);
11    public ResponseBuilder cacheControl(CacheControl cc);
12    public ResponseBuilder expires(Date date);
13    public ResponseBuilder header(String name, Object value);
14    public ResponseBuilder cookie(NewCookie... cookies);
15    public Response build();
16 }
```

5.3.6. `@Path` i `@PathParam`

Adnotacja `@Path` zawiera schemat URI jako „wyrażenie ścieżkowe" z nazwanymi parametrami w nawiasach `{}`:

```
1 @Path("/message")
2 public class MessageRestService {
3
4     @GET
5     @Path("/{imie}")
6     public Response printMessage(@PathParam("imie") String name) {
7         return Response.ok("Hello " + name).build();
8     }
9 }
```

Schematy mogą:

- posiadać część stałą i zmienną: `@Path("/sygnatura/IC{id}")` → `GET /sygnatura/IC0123` → `id = 0123`
- mapować wiele segmentów: `@Path("/cars/{marka}/{model}")` → `GET /VW/Golf`
- używać wyrażeń regularnych: `@Path("/users/abc{id:\\d+}def")` → `GET /abc123def`

`@PathParam` może być typu `PathSegment`, który zawiera nazwę segmentu oraz parametry macierzy:

```
1 @GET
2 @Path("cars/{model}/{year}")
3 public String getString(@PathParam("model") PathSegment car,
```

```
4                       @PathParam("year") String year) { ... }
```

Żądanie `GET /cars/e55;color=black/2006` → `car.getPath() = "e55"`, `car.getMatrixParameters() = {color: "black"}`.

5.3.7. `@QueryParam` i `@DefaultValue`

```
1 @GET
2 public String getOrders(@QueryParam("size") @DefaultValue("25") int size) { ... }
```

Żądanie `GET /orders?size=30` → `size = 30`. Bez parametru → `size = 25`.

5.3.8. Subresource locators

Metoda zasobu może zwrócić nie odpowiedź (`Response`), lecz obiekt — JAX-RS używa go wówczas jako *pod-zasób* i dalej dopasowuje pozostałą część ścieżki URI do metod tego obiektu. Umożliwia to budowanie hierarchicznych, dynamicznie tworzonych zasobów:

```
1  @Path("/customers")
2  public class CustomerService {
3
4      @Path("{id}")                                  // brak @GET — to locator, nie metoda zasobu
5      public CustomerResource getCustomer(@PathParam("id") int id) {
6          return new CustomerResource(id);
7      }
8  }
9
10 public class CustomerResource {
11
12     @GET
13     @Produces("application/xml")
14     public Customer get() { ... }                  // GET /customers/123
15
16     @Path("orders")
17     public OrderResource getOrders() { ... }   // GET /customers/123/orders → kolejny poziom
18 }
```

Subresource locators są szczególnie użyteczne, gdy konkretna klasa obsługująca zasób zależy od danych (np. od wersji API w nagłówku lub od identyfikatora w URI).

5.4. Klasy providerów

JAX-RS posiada mechanizm rozszerzeń oparty na providerach z adnotacją `@Provider`.

5.4.1. ExceptionMapper

Mapuje wyjątki na odpowiedzi HTTP:

```
@Provider
public class UserNotFoundMapper implements ExceptionMapper<UserNotFoundException> {
    @Override
    public Response toResponse(UserNotFoundException ex) {
        return Response.status(404)
                       .entity(ex.getMessage())
                       .type("text/plain")
                       .build();
    }
}
```

Specjalna klasa `WebApplicationException` pozwala na tworzenie odpowiedzi błędu bez konieczności definiowania providera.

5.4.2. MessageBodyReader i MessageBodyWriter

`MessageBodyReader<T>` — konwersja wiadomości przychodzących na obiekty Javy:

```
public interface MessageBodyReader<T extends Object> {
    public boolean isReadable(Class<?> type, Type genericType,
                              Annotation[] annotations, MediaType mt);
    public T readFrom(Class<T> type, Type genericType, Annotation[] annotations,
                      MediaType mt, MultivaluedMap<String,String> httpHeaders,
                      InputStream entityStream) throws IOException;
}
```

`MessageBodyWriter<T>` — konwersja obiektów Javy na reprezentacje:

```
public interface MessageBodyWriter<T extends Object> {
    public boolean isWriteable(Class<?> type, Type genericType,
                               Annotation[] annotations, MediaType mt);
    public long getSize(T t, Class<?> type, Type genericType,
                        Annotation[] annotations, MediaType mt);
    public void writeTo(T t, Class<?> type, Type genericType, ...);
}
```

5.4.3. ContextResolver

`javax.ws.rs.ext.ContextResolver<T>` — fabryka obiektów pomocniczych (np. `JAXBContext`). Posiada jedną metodę:

```
public interface ContextResolver<T extends Object> {
    public T getContext(Class<?> type);
}
```

Typowym zastosowaniem jest dostarczenie niestandardowego `JAXBContext` dla wskazanych klas:

```
@Provider
public final class JAXBContextResolver implements ContextResolver<JAXBContext> {

    private final JAXBContext context;
    private final Set<Class> types;
    private final Class[] cTypes = {Flights.class, FlightType.class, AircraftType.class};

    public JAXBContextResolver() throws Exception {
        this.types = new HashSet(Arrays.asList(cTypes));
        this.context = JAXBContext.newInstance(cTypes);
    }

    public JAXBContext getContext(Class<?> objectType) {
        return types.contains(objectType) ? context : null;
    }
}
```

JAX-RS posiada domyślny kontekst JAXB, tworzony ze znalezionych klas z `@XmlType` i `@XmlRootElement`. `ContextResolver` umożliwia rejestrację niestandardowego kontekstu dla wybranych klas — przydatne m.in. gdy schemat XML wymaga specjalnych adapterów.

Adnotacje `@Consumes` i `@Produces` można stosować również na klasach providerów:

- `@Produces` nad `MessageBodyWriter<T>` — provider brany pod uwagę tylko dla wskazanego MIME,
- `@Consumes` nad `MessageBodyReader<T>` — analogicznie dla parsowania.

5.5. Negocjacja reprezentacji

Klient HTTP może określić oczekiwany format za pomocą nagłówków `Accept`:

```
Accept: application/xml,text/xml;q=0.7,text/html;q=0.5
Accept-Language: pl, en-us;q=0.7
```

JAX-RS wspiera negocjację przez adnotacje:

`@Consumes`
: Określa format wejściowy akceptowany przez metodę/klasę.

`@Produces`
: Określa format wyjściowy generowany przez metodę/klasę.

```
@Path("/myResource")
@Produces("text/plain")
public class SomeResource {

    @GET
    public String doGetAsPlainText() { ... }

```

```
    @GET
    @Produces("text/html")
    public String doGetAsHtml() { ... }

    @GET
    @Produces({"application/xml", "application/json"})
    public Response doGetAsXmlOrJson() { ... }

    @POST
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    public String doPost2(Form formData) { ... }
}
```

Serwer zwróci `406 Not Acceptable` jeśli nie może zbudować odpowiedzi w żadnym formacie, lub `415 Unsupported Media Type` jeśli format przesłanego payloadu jest nieobsługiwany.

5.5.1. JSON w JAX-RS z JAXB

Klasa adnotowana `@XmlRootElement` może być automatycznie serializowana do JSON, gdy implementacja JAX-RS zawiera dostawcę JSON (np. Jersey z modułem `jersey-media-json-jackson` lub RESTEasy z `resteasy-jackson-provider`):

```
@XmlRootElement
public class Product {
    public int    id;
    public String name;
    public double price;

    public Product() {}              // wymagany przez JAXB
    public Product(int id, String name, double price) {
        this.id = id; this.name = name; this.price = price;
    }
}
```

```
@Path("/products")
public class ProductResource {

    @GET
    @Path("{id}")
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Product getProduct(@PathParam("id") int id) {
        return new Product(id, "Effective Java", 59.90);
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Response createProduct(Product product) {
        // ... zapis do bazy danych
        URI location = uriInfo.getAbsolutePathBuilder()
                              .path(String.valueOf(product.id)).build();
        return Response.created(location).entity(product).build();
    }
```

```
20 }
```

Żądanie `GET /products/1` z nagłówkiem `Accept: application/json` zwraca:

```
1 {"id":1,"name":"Effective Java","price":59.90}
```

Żądanie `GET /products/1` z nagłówkiem `Accept: application/xml` zwraca:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <product><id>1</id><name>Effective Java</name><price>59.9</price></product>
```

Ten sam zasób, ta sama implementacja — JAX-RS i JAXB automatycznie wybierają format na podstawie nagłówka `Accept` klienta.

5.6. Wsparcie JAX-RS dla HATEOAS i cachowania

5.6.1. Tworzenie odnośników — UriBuilder

`javax.ws.rs.core.UriBuilder` pozwala na programowe budowanie URI:

```
1 @Path("/customers")
2 public class CustomerService {
3     @GET
4     @Produces("application/xml")
5     public String getCustomers(@Context UriInfo uriInfo) {
6         UriBuilder nextLinkBuilder = uriInfo.getAbsolutePathBuilder();
7         nextLinkBuilder.queryParam("start", 5);
8         nextLinkBuilder.queryParam("size", 10);
9         URI next = nextLinkBuilder.build();
10        // ... dołącz link do odpowiedzi
11    }
12 }
```

5.6.2. Żądania warunkowe

JAX-RS wspiera `Cache-Control` przez `javax.ws.rs.core.CacheControl`:

```
1 CacheControl cc = new CacheControl();
2 cc.setMaxAge(300);
3 cc.setPrivate(true);
4 cc.setNoStore(true);
5 ResponseBuilder builder = Response.ok(customer, "application/xml");
6 builder.cacheControl(cc);
7 Response response = builder.build();
```

Żądania warunkowe: **Last-Modified** / **If-Modified-Since** oraz **ETag** / **If-None-Match**:

```
public Response getCustomer(@PathParam("id") int id,
                            @Context Request request) {
    Customer cust = ...;
    EntityTag tag = new EntityTag(Integer.toString(cust.hashCode()));

    ResponseBuilder builder = request.evaluatePreconditions(tag);
    if (builder != null) {
        return builder.build(); // 304 Not Modified
    }
    return Response.ok(cust, "application/xml").tag(tag).build();
}
```

5.7. Deployment serwisów JAX-RS

Web serwisy JAX-RS działają jako komponenty w web kontenerze lub jako ziarna EJB.

`web.xml` — *przykład konfiguracji z Jersey*

```
<web-app>
  <servlet>
    <servlet-name>Jersey Web Application</servlet-name>
    <servlet-class>
      org.glassfish.jersey.servlet.ServletContainer
    </servlet-class>
    <init-param>
      <param-name>javax.ws.rs.Application</param-name>
      <param-value>org.foo.rest.MyApplication</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>Jersey Web Application</servlet-name>
    <url-pattern>/api/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Alternatywnie można wskazać pakiety do skanowania:

```
<init-param>
  <param-name>jersey.config.server.provider.packages</param-name>
  <param-value>org.foo.rest;org.bar.rest</param-value>
</init-param>
```

5.7.1. Konfiguracja bez `web.xml` — `@ApplicationPath`

W Java EE 6 preferowanym sposobem konfiguracji jest adnotacja `@ApplicationPath` na podklasie `Application` — kontener automatycznie rejestruje serwlet dostawcy JAX-RS:

```
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/api")
public class MyApplication extends Application {
```

```
6     // pusta klasa — kontener skanuje wszystkie zasoby i providery w archiwum WAR
7 }
```

Wszystkie zasoby i providery w tym samym archiwum WAR są automatycznie wykrywane. Aby zarejestrować je ręcznie (np. w celu ograniczenia skanowania), należy nadpisać metody `getClasses()` lub `getSingletons()` klasy `Application`.

5.8. Client API (JAX-RS 2.0)

JAX-RS 1.x nie definiował standardowego API po stronie klienta — każda implementacja (Jersey, RESTEasy) miała własne, nieprzenośne rozwiązanie. JAX-RS 2.0 wprowadził fluent Client API umożliwiające wysyłanie żądań HTTP bez użycia niskopoziomowych klas `HttpURLConnection`:

```
Client client = ClientBuilder.newClient();

// Pobranie zasobu
Response response = client.target("http://example.com/api/customers/123")
                          .request(MediaType.APPLICATION_XML)
                          .get();
Customer customer = response.readEntity(Customer.class);

// Utworzenie zasobu
Customer newCustomer = new Customer("Jan", "Kowalski");
Response created = client.target("http://example.com/api/customers")
                         .request()
                         .post(Entity.xml(newCustomer));
// 201 Created + nagłówek Location z URI nowego zasobu

client.close(); // Client zarządza połączeniami — należy go zamknąć
```

Klasa / interfejs	Rola
`ClientBuilder`	Fabryka; punkt wejścia — `ClientBuilder.newClient()`
`Client`	Zarządza połączeniami; należy zamknąć po użyciu (`client.close()`)
`WebTarget`	Reprezentuje URI lub szablon URI; tworzy kolejne `WebTarget` przez `.path()`
`Invocation.Builder`	Konfiguruje żądanie (nagłówki, format); wywołuje `get()`, `post()`, `put()`, `delete()`
`Response`	Odpowiedź HTTP: kod statusu, nagłówki, ciało dostępne przez `readEntity(Class<T>)`

5.9. Podsumowanie

JAX-RS realizuje po stronie Java EE zasady REST opisane w poprzednim rozdziale.

Adnotacje (`@Path`, `@GET`, `@Produces`, `@Consumes`) eliminują większość kodu integracyjnego, a mechanizm providerów (`MessageBodyReader`, `MessageBodyWriter`, `ExceptionMapper`) umożliwia obsługę dowolnych formatów i centralne zarządzanie błędami — bez modyfikacji klas zasobów.

Kolejne rozdziały przechodzą do drugiego stylu web serwisów — protokołu SOAP — opisując kolejno sam protokół, język opisu interfejsu WSDL, niskopoziomowe API przetwarzania XML (JAXP, JAXB) oraz implementację po stronie Java EE w specyfikacji JAX-WS.

Część III: SOAP i JAX-WS

6. Simple Object Access Protocol — SOAP

Simple Object Access Protocol został stworzony przez *DevelopMentor, Inc.* i spopularyzowany przez *Microsoft*. Dzisiaj standard utrzymywany jest przez *W3C*. Najnowsza wersja (z 2007 roku) nosi numer 1.2, w użyciu jednak nadal pozostaje wersja 1.1 (z 2000 roku).

W odróżnieniu od REST, który opiera się bezpośrednio na semantyce HTTP, SOAP definiuje własny protokół wiadomości niezależny od warstwy transportowej. Ta niezależność jest zarówno siłą SOAP — może działać przez HTTP, JMS, SMTP lub TCP — jak i źródłem jego złożoności w porównaniu do REST.

6.1. Podstawowe cechy

SOAP jest:

Tekstowym protokołem komunikacyjnym
: W przeciwieństwie do binarnych protokołów jak IIOP (CORBA) lub JRMP (Java RMI).

Opartym na standardzie XML
: Czyni go niezależnym od platformy, rozszerzalnym, czytelnym dla ludzi i maszyn.

Bezstanowy
: Nie przechowuje stanu między wywołaniami.

Jednokierunkowy
: Nie ma formalnego rozróżnienia żądania i odpowiedzi. Obie wiadomości mają identyczną strukturę.

Niezależny od protokołu transportowego
: Wiadomości mogą być przesyłane przez HTTP, JMS, SMTP.

Niezależny od modelu aplikacji
: Może być użyty zarówno do przesyłania wiadomości, jak i do klasycznych serwisów RPC.

SOAP nie definiuje wsparcia dla:

- odzyskiwania pamięci (*garbage collection*),
- referencji do obiektów i aktywacji,
- grupowania wiadomości (*message batching*).

6.2. Model komunikacyjny w SOAP

SOAP bazuje na modelu jednokierunkowym przesyłania wiadomości. Ewentualne powiązanie wiadomości musi odbyć się za pomocą mechanizmów protokołu komunikacyjnego.

Wiadomości przesyłane są pomiędzy węzłami:

Nadawcy (SOAP senders)

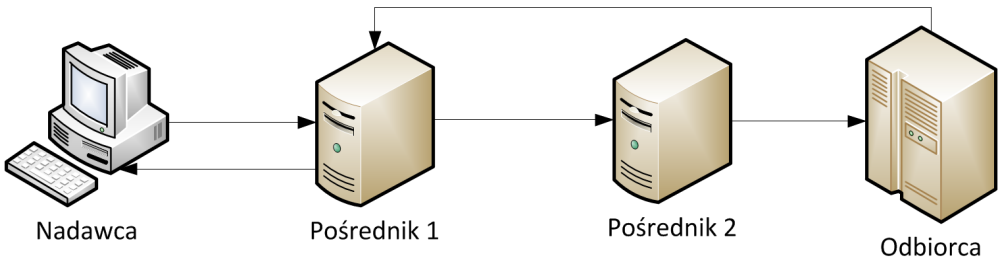
Węzeł tworzący i wysyłający wiadomość.

Pośrednicy (SOAP intermediaries)

Węzeł będący jednocześnie odbiorcą i nadawcą. Pośredniczy w wymianie komunikatu. Może wiadomość czytać i przetwarzać, ale nie powinien zmieniać elementów ciała. Na trasie wiadomości może istnieć dowolna liczba pośredników.

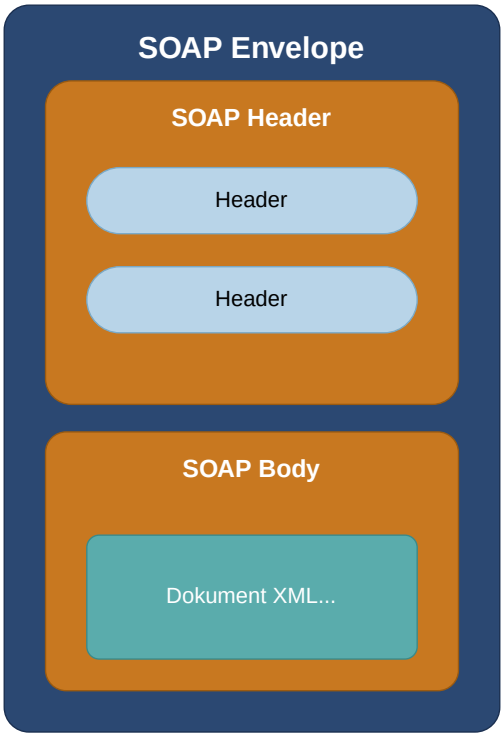
Odbiorcy końcowi (SOAP receivers)

Węzeł, do którego kierowana jest wiadomość.



Rysunek 11. Model komunikacyjny SOAP

6.3. Struktura wiadomości



Rysunek 12. Struktura wiadomości SOAP

6.3.1. SOAP Envelope

Kontener dla całej wiadomości. W reprezentacji XML jest korzeniem drzewa. Nazwa elementu to `Envelope`.

Przestrzenie nazw koperty:

- SOAP 1.1: `http://schemas.xmlsoap.org/soap/envelope/`
- SOAP 1.2: `http://www.w3.org/2003/05/soap-envelope`

W wiadomości mogą pojawić się elementy z obcych przestrzeni nazw. Zabronione jest używanie deklaracji DTD i PI. Wewnątrz koperty może pojawić się element `Header` — tylko jako pierwszy potomek `Envelope`. Obowiązkowy element `Body` musi być obecny i może być poprzedzony tylko nagłówkiem.

6.3.2. SOAP Header

Opcjonalny element wewnątrz koperty. Zawiera zbiór wpisów nagłówkowych (*header entries*). Używany do uwierzytelniania, adresowania, transakcji. Wszystkie wpisy muszą być *namespace qualified*.

Atrybuty bloku nagłówka:

`actor`
: URI oznaczające węzeł, do którego kierowany jest nagłówek. Specjalna wartość:

`http://schemas.xmlsoap.org/soap/actor/next`. Brak atrybutu oznacza, że nagłówek przeznaczony jest dla odbiorcy końcowego. Węzeł, który przetworzył blok, usuwa go przed przesłaniem dalej.

`mustUnderstand`

Ustawiony na `"1"` oznacza, że nagłówek musi być zrozumiany przez odbiorcę wskazywanego przez `actor`. Jeśli tak nie jest, odbiorca musi odpowiedzieć błędem. Nagłówki z tym atrybutem muszą być przetwarzane w pierwszej kolejności.

6.3.3. SOAP Body

Obowiązkowy element koperty — właściwa wiadomość dla odbiorcy końcowego. Może przenosić dokumenty XML, polecenia RPC i informacje o błędach. Bloki ciała mogą być *namespace qualified*.

6.3.4. Przykład wiadomości SOAP 1.1

```
<!-- Koperta wiadomości -->
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <!-- Nagłówek -->
    <SOAP-ENV:Header>
        <person:mail xmlns:person="http://example.com/Header/">
            xyz@example.com
        </person:mail>
    </SOAP-ENV:Header>

    <!-- Ciało wiadomości -->
    <SOAP-ENV:Body>
        <m:GetBookPrice xmlns:m="http://www.example.com/priceList">
            <bookname xsi:type='xsd:string'>
                Developing Java Web Services
            </bookname>
        </m:GetBookPrice>
    </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

6.3.5. SOAP Fault

Jedynym standardowym blokiem ciała jest *SOAP Fault* — przenosi informacje o błędach. Może istnieć tylko jeden taki element w ciele wiadomości.

Podelementy:

Element	Wymagany	Opis
`faultcode`	TAK	URI identyfikujące błąd. Standardowe kody: `VersionMismatch`, `MustUnderstand`, `Client`, `Server`
`faultstring`	TAK	Komunikat tekstowy dla człowieka
`faultactor`	NIE	Węzeł, który wygenerował błąd (nieobowiązkowy gdy błąd generuje odbiorca końcowy)
`detail`	TAK*	Szczegóły błędu (*obowiązkowy gdy błąd dotyczy ciała, niedozwolony gdy nagłówka)

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Body>
        <SOAP-ENV:Fault>
            <faultcode>SOAP-ENV:Server</faultcode>
            <faultstring>Server Error</faultstring>
            <detail>
                <e:myfaultdetails xmlns:e="Some-URI">
                    <message>My application did not work</message>
                    <errorcode>1001</errorcode>
                </e:myfaultdetails>
            </detail>
        </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

6.4. SOAP Encoding

SOAP Encoding — styl zaczerpnięty z języków programowania do definiowania i rozpoznawania typów. Stworzony gdy nie istniała jeszcze XML Schema (używany dla RPC).

Styl ten wprowadza: tablice, odnośniki (`href` i `ID`), polimorficzny akcesor (`<cost xsi:type="xsd:float">29.95</cost>`).

Styl ten jest wspierany tylko ze względów historycznych i nie jest zalecany dla nowych serwisów. Zabrania go WS-I Basic Profile.

6.5. Wiązanie HTTP dla SOAP

SOAP naturalnie łączy się z HTTP. Zasady użycia SOAP przez HTTP:

- `Content-Type` wiadomości musi być ustawiony na `text/xml` (SOAP 1.1) lub `application/soap+xml` (SOAP 1.2),

- do przesyłania żądań używana jest wyłącznie metoda HTTP `POST`,
- klient musi użyć nagłówka HTTP `SOAPAction` jako identyfikator akcji (pusta wartość `""` — gdy akcja nie jest wyspecyfikowana),
- wiadomości pozytywne zwracane są ze statusem `200 OK`,
- SOAP Fault zwracany jest ze statusem `500 Internal Server Error`.

6.6. Specyfikacja WS-I Basic Profile

Wydana przez *Web Services Interoperability Industry Consortium* — doprecyzowuje specyfikacje web serwisów w celu zapewnienia zgodności między implementacjami na różnych platformach. Wersje: 1.0 (2004), 1.1 (2006), 1.2 (2010), 2.0 (2010, SOAP 1.2).

Główne zalecenia:

- element `Fault` może zawierać tylko standardowe elementy,
- pierwsi potomkowie `Fault` muszą być *namespace unqualified*,
- `faultstring` może zawierać atrybut `xml:lang`,
- niestandardowe kody błędów definiowane w przestrzeni aplikacji — bez *dot notation*,
- zabrania się umieszczania elementów za `Body`,
- wiadomość w UTF-8 lub UTF-16,
- nagłówki z `mustUnderstand="1"` sprawdzane w pierwszej kolejności,
- `SOAPAction` musi być ujęty w cudzysłów,
- status 2xx dla wiadomości pozytywnych (zalecane 200 lub 202),
- przekierowanie tylko przez `307 Temporary Redirect`,
- `400 Bad Request` gdy wiadomość jest niepoprawnie sformatowana.

6.7. Style wiadomości SOAP

Sposób odwzorowania operacji i parametrów na strukturę XML wiadomości określony jest dwiema właściwościami definiowanymi w dokumencie WSDL: **styl** (*style*) i **kodowanie** (*use*).

6.7.1. Styl (style)

`rpc`

: Ciało wiadomości zawiera element o nazwie wywoływanej operacji. Jego podelementy są parametrami wywołania. Odpowiedź zawiera element `nazwaOperacjiResponse`.

`document`

: Ciało wiadomości może zawierać dowolny dokument XML zdefiniowany przez XMLSchema. Brak narzuconej nazwy odpowiadającej operacji — struktura

zdefiniowana wyłącznie schematem.

6.7.2. Kodowanie (use)

`encoded`

Typy parametrów opisane atrybutem `xsi:type` zgodnie z regułami kodowania SOAP — mechanizm stosowany przed upowszechnieniem XMLSchema.

Kodowanie `encoded` jest sprzeczne z WS-I Basic Profile. Nie wolno go stosować w nowych serwisach.

`literal`

Typy parametrów opisane przez XMLSchema. Wartości przesyłane bezpośrednio — bez dodatkowych metadanych typów w wiadomości.

6.7.3. Zalecane kombinacje

style	use	Uwagi
`rpc`	`encoded`	Przestarzały; zakazany przez WS-I Basic Profile. Nie używać.
`rpc`	`literal`	Obsługiwany, lecz rzadko stosowany; problemy z przestrzeniami nazw.
`document`	`encoded`	Niespójna kombinacja; praktycznie nieużywana.
`document`	`literal`	**Zalecany**; zgodny z WS-I BP; możliwa walidacja XMLSchema.

Najpopularniejszą odmianą `document/literal` jest konwencja **wrapped** — ciało zawiera jeden element-opakowanie o nazwie operacji, którego podelementy są parametrami. Jest to domyślny styl generowany przez JAX-WS (`@SOAPBinding(style=DOCUMENT, use=LITERAL, parameterStyle=WRAPPED)`).

6.8. Zastosowanie SOAP do RPC

SOAP mapuje wywołanie metody na XML:

- nazwa bloku w ciele odpowiada nazwie metody,
- parametry metody są podelementami tego bloku, z określonymi typami,
- odpowiedź to nazwa metody + `"Response"`,
- w tym elemencie zawarty jest element z wartością zwracaną.

Przykład żądania i odpowiedzi w stylu `rpc/literal`:

Żądanie

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:m="http://www.example.com/bookstore">
  <SOAP-ENV:Body>
    <m:GetBookPrice>                    <!-- nazwa operacji -->
      <bookname>Effective Java</bookname>
    </m:GetBookPrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Odpowiedź

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:m="http://www.example.com/bookstore">
  <SOAP-ENV:Body>
    <m:GetBookPriceResponse>            <!-- nazwa operacji + "Response" -->
      <return>49.95</return>
    </m:GetBookPriceResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

6.9. SOAP 1.2

Cele nowej wersji:

- uściślenie modelu i przetwarzania SOAP,
- oparcie o *XML Information Set* — abstrakcyjny model XML umożliwiający optymalizację (np. dla danych binarnych),
- definicja wiązania niezależna od protokołu transportowego,
- lepsza integracja z HTTP (wiązanie pozwala na użycie `GET`).

6.9.1. Różnice SOAP 1.1 vs 1.2

Aspekt	SOAP 1.1	SOAP 1.2
Przestrzeń nazw	`schemas.xmlsoap.org/soap/envelope/`	`www.w3.org/2003/05/soap-envelope`
Atrybut węzła docelowego	`actor`	`role` (z predefiniowanymi rolami: `none`, `next`, `ultimateReceiver`)
`mustUnderstand`	`"0"` lub `"1"`	`true` lub `false`
Nowy atrybut	—	`relay` (boolean)
Nowy nagłówek	—	`NotUnderstood`
Dot notation w błędach	dozwolona	zabroniona

Aspekt	SOAP 1.1	SOAP 1.2
SOAPAction	nagłówek HTTP	parametr `action` w `Content-Type`
`Content-Type` dla HTTP	`text/xml`	`application/soap+xml`

6.9.2. SOAP Fault w wersji 1.2

```
<env:Fault>
    <env:Code>
        <env:Value>env:Sender</env:Value>
        <env:Subcode>
            <env:Value>m:MessageTimeout</env:Value>
        </env:Subcode>
    </env:Code>
    <env:Reason>
        <env:Text xml:lang="en">Sender Timeout</env:Text>
    </env:Reason>
    <env:Detail>
        <m:MaxTime>P5M</m:MaxTime>
    </env:Detail>
</env:Fault>
```

Rozwinięty kod błędu: `Code/Value` i `Code/SubCode/Value`. `faultstring` zastąpiony przez `Reason` z wieloma `Text` dla różnych języków.

6.10. SOAP a stos technologiczny Java EE

SOAP jest protokołem wiadomości — określa strukturę koperty i zasady przetwarzania węzłów. Praktyczne tworzenie serwisów SOAP wymaga kolejnych elementów stosu:

- **WSDL** — kontrakt serwisu opisujący operacje, typy danych i adresy endpointów; omówiony w następnym rozdziale,
- **XMLSchema** — system typów osadzony w WSDL (poznany w rozdziale XML),
- **JAX-WS** — specyfikacja Java EE do budowania i konsumowania serwisów SOAP,
- **SAAJ** — niskopoziomowy dostęp do wiadomości SOAP bezpośrednio z kodu Java.

Następny rozdział opisuje WSDL — dokument XML stanowiący kontrakt serwisu SOAP, który definiuje co serwis udostępnia i jak się z nim komunikować.

7. Web Services Description Language

Web Services Description Language (WSDL) jest językiem bazującym na XML służącym do opisu usług sieciowych — pełni rolę kontraktu serwisu SOAP. Stworzony przez Microsoft i IBM, następnie przekazany do W3C do standaryzacji. Powszechnie wspierana wersja standardu nosi numer 1.1. Wersja 2.0 wprowadza pełne wsparcie dla serwisów REST[1].

Znajomość WSDL jest niezbędna do zrozumienia JAX-WS — specyfikacja ta w całości opiera się na WSDL jako kontrakcie między klientem a serwisem. Na podstawie dokumentu WSDL narzędzie `wsimport` (JAX-WS RI / Metro) lub `wsdl2java` (Apache CXF) automatycznie generuje interfejsy Javy, klasy stub po stronie klienta i szkielet implementacji serwisu.

W WSDL znajdują się informacje o:

- strukturze wiadomości (XML Schema),
- możliwych do wykonania operacjach,
- szczegółach protokołu komunikacyjnego i transportowego,
- konkretnych adresach usług sieciowych (*endpoints*).

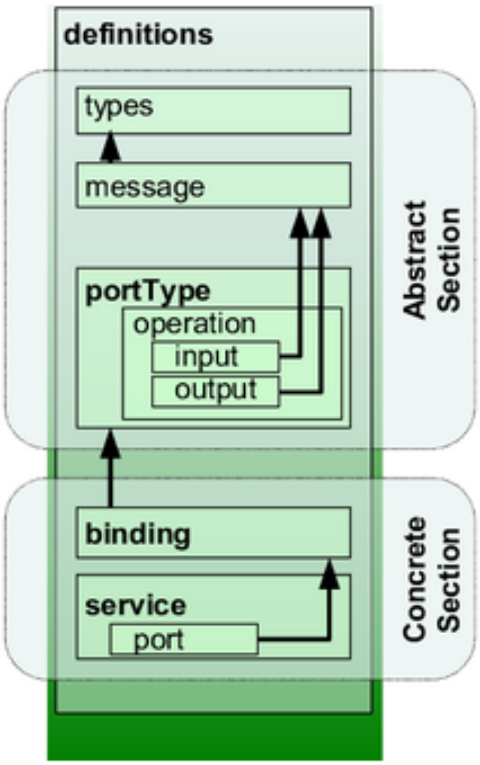
WSDL opisuje usługę sieciową w sposób abstrakcyjny i niezależny od języka programowania. Na jego podstawie możliwe jest automatyczne generowanie szkieletów klientów i serwisów.

7.1. WSDL 1.1

7.1.1. Struktura dokumentu WSDL

Opis serwisu w WSDL dzieli się na dwie części:

- **abstrakcyjną** — opis wiadomości i możliwych operacji (schemat wymiany wiadomości),
- **konkretną** — wiązanie (*binding*) z konkretnym protokołem komunikacyjnym i formatem wiadomości.



Rysunek 13. Struktura WSDL

7.1.1.1. Elementy abstrakcyjne

Element	Podelement	Opis
`types`	—	Typy danych używane przy wymianie wiadomości (zazwyczaj XMLSchema)
`message`	`part`	Abstrakcyjna definicja wiadomości złożonej z części wskazujących na typy
`portType`	`operation`	„Interfejs" usługi — zestaw akcji jakie można wykonać na serwisie
`operation`	`input` / `output` / `fault`	Łączy wiadomości w możliwe akcje; definiuje żądanie, odpowiedź i błędy

Typy operacji:

Jednokierunkowa (*one-way*) — wiadomość wysyłana do serwera, bez odpowiedzi:

```
1 <wsdl:operation name="nmtoken">
2   <wsdl:input name="nmtoken"? message="qname"/>
3 </wsdl:operation>
```

Żądanie-odpowiedź (*request-response*) — żądanie i odpowiedź z opcjonalnymi błędami:

```
<wsdl:operation name="nmtoken" parameterOrder="nmtokens">
  <wsdl:input  name="nmtoken"? message="qname"/>
  <wsdl:output name="nmtoken"? message="qname"/>
  <wsdl:fault  name="nmtoken" message="qname"/>*
</wsdl:operation>
```

Sygnał-odpowiedź (*solicit-response*) — serwer inicjuje, klient odpowiada:

```
<wsdl:operation name="nmtoken" parameterOrder="nmtokens">
  <wsdl:output name="nmtoken"? message="qname"/>
  <wsdl:input  name="nmtoken"? message="qname"/>
  <wsdl:fault  name="nmtoken" message="qname"/>*
</wsdl:operation>
```

Powiadomienie (*notification*) — serwer wysyła, klient nie odpowiada:

```
<wsdl:operation name="nmtoken">
  <wsdl:output name="nmtoken"? message="qname"/>
</wsdl:operation>
```

WSDL dostarcza wiązania tylko dla pierwszych dwóch typów.

7.1.1.2. Elementy konkretne

Element	Podelement	Opis
`binding`	`operation`	Sposób użycia operacji z konkretnym protokołem (SOAP, HTTP)
`service`	`port`	Nazwa serwisu i lista portów z adresami (endpoints)

WSDL 1.1 definiuje wiązania dla protokołów: SOAP 1.1, HTTP (GET i POST) oraz MIME (dla załączników).

Element `binding` odwołuje się do `portType` przez atrybut `type` i dodaje szczegóły protokołu dla każdej operacji. Element `service` wskazuje `binding` przez atrybut `binding` w elemencie `port`. Podział abstrakcja (`portType`) / konkretyzacja (`binding` + `service`) umożliwia opisanie jednego interfejsu logicznego za pomocą wielu protokołów lub wielu endpointów.

Ogólny schemat elementu `binding` z elementami rozszerzającymi (placeholderami protokołu):

```
<wsdl:definitions .... >
```

```
  <wsdl:binding name="nmtoken" type="qname"> *
    <!-- extensibility element (1) --> *
    <wsdl:operation name="nmtoken"> *
      <!-- extensibility element (2) --> *
      <wsdl:input name="nmtoken"? > ?
        <!-- extensibility element (3) -->
      </wsdl:input>
      <wsdl:output name="nmtoken"? > ?
        <!-- extensibility element (4) --> *
      </wsdl:output>
      <wsdl:fault name="nmtoken"> *
        <!-- extensibility element (5) --> *
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>
</wsdl:definitions>
```

Ogólny schemat elementu `service`:

```
<wsdl:service name="nmtoken"> *
  <wsdl:documentation .... />?
  <wsdl:port name="nmtoken" binding="qname"> *
    <wsdl:documentation .... /> ?
    <!-- extensibility element -->
  </wsdl:port>
  <!-- extensibility element -->
</wsdl:service>
```

7.1.2. Przykład dokumentu WSDL — StockQuoteService

```
<?xml version="1.0"?>
<definitions name="StockQuote"
    targetNamespace="http://example.com/stockquote.wsdl"
    xmlns:tns="http://example.com/stockquote.wsdl"
    xmlns:xsd1="http://example.com/stockquote.xsd"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

    <types>
        <schema targetNamespace="http://example.com/stockquote.xsd"
                xmlns="http://www.w3.org/2001/XMLSchema">
            <element name="TradePriceRequest">
                <complexType>
                    <all>
                        <element name="tickerSymbol" type="string"/>
                    </all>
                </complexType>
            </element>
            <element name="TradePrice">
                <complexType>
                    <all>
                        <element name="price" type="float"/>
                    </all>
                </complexType>
            </element>
        </schema>
    </types>
```

```
28
29      <message name="GetLastTradePriceInput">
30          <part name="body" element="xsd1:TradePriceRequest"/>
31      </message>
32      <message name="GetLastTradePriceOutput">
33          <part name="body" element="xsd1:TradePrice"/>
34      </message>
35
36      <portType name="StockQuotePortType">
37          <operation name="GetLastTradePrice">
38              <input  message="tns:GetLastTradePriceInput"/>
39              <output message="tns:GetLastTradePriceOutput"/>
40          </operation>
41      </portType>
42
43      <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
44          <soap:binding style="document"
45                        transport="http://schemas.xmlsoap.org/soap/http"/>
46          <operation name="GetLastTradePrice">
47              <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
48              <input>  <soap:body use="literal"/> </input>
49              <output> <soap:body use="literal"/> </output>
50          </operation>
51      </binding>
52
53      <service name="StockQuoteService">
54          <documentation>My first service</documentation>
55          <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
56              <soap:address location="http://example.com/stockquote"/>
57          </port>
58      </service>
59
60  </definitions>
```

Przykład definiuje serwis zwracający cenę akcji z jedną operacją `GetLastTradePrice`:

`types`

Sekcja abstrakcyjna — zawiera schemat XMLSchema z definicją dwóch elementów: `TradePriceRequest` (żądanie z symbolem akcji) i `TradePrice` (odpowiedź z ceną).

`message`

Dwa elementy wiążą nazwy wiadomości z elementami zdefiniowanymi w schemacie. `GetLastTradePriceInput` wskazuje na `TradePriceRequest`, a `GetLastTradePriceOutput` — na `TradePrice`.

`portType`

Logiczny interfejs serwisu — grupuje operację `GetLastTradePrice` z wiadomością wejściową i wyjściową. Odpowiada interfejsowi Java: określa co serwis *robi*, bez wskazywania jak i gdzie.

`binding`

Konkretyzuje `portType` dla protokołu SOAP 1.1 przez HTTP w stylu `document/literal`. Każda operacja otrzymuje wartość `soapAction` używaną w nagłówku HTTP.

`service`

Przypisuje wiązaniu konkretny adres URL endpointu (`http://example.com/stockquote`). Jeden serwis może mieć wiele portów (np. ten sam `portType` wystawiony przez HTTP i przez JMS).

7.1.3. Importowanie zewnętrznych schematów i dokumentów WSDL

Duże serwisy często dzielą definicje na osobne pliki aby ułatwić zarządzanie i ponowne użycie. WSDL 1.1 dostarcza do tego celu dwa mechanizmy.

7.1.3.1. `<wsdl:import>` — import definicji WSDL

`<wsdl:import>` importuje definicje z zewnętrznego dokumentu WSDL (typy, wiadomości, portType):

```
<definitions name="StockQuote"
    targetNamespace="http://example.com/stockquote.wsdl"
    xmlns:tns="http://example.com/stockquote.wsdl"
    xmlns:sqt="http://example.com/stockquote-types.wsdl"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

    <import namespace="http://example.com/stockquote-types.wsdl"
            location="stockquote-types.wsdl"/>

    <binding name="StockQuoteSoapBinding" type="sqt:StockQuotePortType">
        ...
    </binding>
    <service name="StockQuoteService">
        ...
    </service>
</definitions>
```

Importowany plik `stockquote-types.wsdl` zawiera abstrakcyjne definicje (`types`, `message`, `portType`). Taka separacja pozwala definiować wiele powiązań (`binding`) i usług (`service`) do tego samego interfejsu w osobnych plikach — np. wiązanie SOAP 1.1 i SOAP 1.2 w oddzielnych dokumentach.

7.1.3.2. `<xs:import>` — import zewnętrznego schematu XSD

W sekcji `<types>` schema WSDL można importować zewnętrzny plik `.xsd`:

```
<definitions ...>
    <types>
        <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                   targetNamespace="http://example.com/stockquote.wsdl">
            <xs:import namespace="http://example.com/stockquote.xsd"
                       schemaLocation="stockquote.xsd"/>
        </xs:schema>
    </types>
    ...
```

```
10 </definitions>
```

Zewnętrzny plik `stockquote.xsd` zawiera definicje elementów i typów danych. Taka struktura jest standardem w środowiskach produkcyjnych — `wsimport` automatycznie pobiera pliki wskazane atrybutem `schemaLocation` i generuje odpowiednie klasy JAXB.

Narzędzie `wsimport` obsługuje importowanie wielopoziomowe — jeśli schemat importuje kolejne schematy, wszystkie zostaną pobrane i skompilowane. Przy zasobach lokalnych (bez serwera HTTP) należy użyć opcji `wsimport -catalog catalog.xml` lub pliku `jax-ws-catalog.xml` (OASIS XML Catalogs), aby przekierować adresy URL na lokalne pliki.

7.1.4. Wiązanie SOAP 1.1

Wiązanie SOAP 1.1 wskazuje powiązanie z protokołem SOAP, pozwala określić adres endpointu i URI nagłówka `SOAPAction`, oraz specyfikuje nagłówki SOAP powiązane z operacją.

Pełny schemat rozszerzenia WSDL dla SOAP:

```
<definitions .... >
  <binding .... >
    <soap:binding style="rpc|document" transport="uri"/>
    <operation .... >
      <soap:operation soapAction="uri"? style="rpc|document"?/>?
      <input>
        <soap:body parts="nmtokens"? use="literal|encoded"
              encodingStyle="uri-list"? namespace="uri"?/>
        <soap:header message="qname" part="nmtoken"
              use="literal|encoded" encodingStyle="uri-list"?
              namespace="uri"?>*
          <soap:headerfault message="qname" part="nmtoken"
              use="literal|encoded" encodingStyle="uri-list"?
              namespace="uri"?/>*
        </soap:header>
      </input>
      <output>
        <soap:body parts="nmtokens"? use="literal|encoded"
              encodingStyle="uri-list"? namespace="uri"?/>
        <soap:header message="qname" part="nmtoken"
              use="literal|encoded" encodingStyle="uri-list"?
              namespace="uri"?>*
          <soap:headerfault message="qname" part="nmtoken"
              use="literal|encoded" encodingStyle="uri-list"?
              namespace="uri"?/>*
        </soap:header>
      </output>
      <fault>*
        <soap:fault name="nmtoken" use="literal|encoded"
            encodingStyle="uri-list"? namespace="uri"?/>
      </fault>
    </operation>
  </binding>
```

```
34   <port .... >
35     <soap:address location="uri"/>
36   </port>
37 </definitions>
```

7.1.4.1. Elementy rozszerzenia WSDL dla SOAP

`soap:binding`

Wskazuje, że wiązanie jest do protokołu SOAP. Poszczególne części wiadomości będą przypisywane elementom koperty (`Header` lub `Body`).

```
1 <binding .... >
2   <soap:binding style="rpc|document" transport="uri"/>
3   <operation .... >
4   </operation>
5 </binding>
```

Atrybuty:

- `style` — styl wiązania: `rpc` lub `document`, domyślnie `document`
- `transport` — obowiązkowe URI protokołu transportowego, np. `http://schemas.xmlsoap.org/soap/http`

`soap:operation`

Definiuje parametry dla całej operacji.

```
1 <binding .... >
2   <operation .... >
3     <soap:operation soapAction="uri"? style="rpc|document"?/>?
4   </operation>
5 </binding>
```

Atrybuty:

- `soapAction` — obowiązkowy dla wiązania HTTP; URI wpisywane w nagłówek `SOAPAction`
- `style` — nieobowiązkowy, nadpisuje styl z `soap:binding`

`soap:body`

Określa jak zbudowane będzie ciało koperty. Części z atrybutu `parts` mogą być zarówno częściami (`part`) z elementów `message`, jak i elementami schematu.

```
1 <operation .... >
2   <input>
3     <soap:body parts="nmtokens"? use="literal|encoded"?
4           encodingStyle="uri-list"? namespace="uri"?/>
5   </input>
6   <output>
7     <soap:body parts="nmtokens"? use="literal|encoded"?
```

```
          encodingStyle="uri-list"? namespace="uri"?/>
  </output>
</operation>
```

Atrybuty:

- `parts` — części składające się na ciało koperty; domyślnie wszystkie z wiadomości
- `use` — `literal` (bezpośrednio) lub `encoded` (kodowanie SOAP Encoding)
- `encodingStyle` — lista URI określających zasady kodowania
- `namespace` — przestrzeń nazw elementu opakowującego (dla stylu `rpc`)

`soap:fault`

Mapuje części wiadomości na element `detail` w SOAP Fault.

```
<operation .... >
  <fault name="nmtoken">*
    <soap:fault name="nmtoken" use="literal|encoded"
         encodingStyle="uri-list"? namespace="uri"?/>
  </fault>
</operation>
```

Atrybuty: `name` (część stanowiąca ciało dla elementu `Fault`), `use`, `encodingStyle`, `namespace`.

`soap:header` i `soap:headerfault`

Pozwalają na mapowanie nagłówków SOAP i błędów skojarzonych z tymi nagłówkami.

```
<operation .... >
  <input|output>
    <soap:header message="qname" part="nmtoken"
        use="literal|encoded" encodingStyle="uri-list"?
        namespace="uri"?>*
      <soap:headerfault message="qname" part="nmtoken"
          use="literal|encoded" encodingStyle="uri-list"?
          namespace="uri"?/>*
    </soap:header>
  </input|output>
</operation>
```

Atrybuty: `message` (wiadomość, z której pochodzi część), `part` (konkretna część wiadomości).

`soap:address`

Lokalizacja endpointu, umieszczana w elemencie `port`.

7.1.4.2. Style i metody serializacji SOAP

Styl	Opis	Status
`document/literal`	Zalecany dla nowych serwisów. Jako ciało koperty przesyłany jest pełny dokument XML.	✅ Zalecany
`document/literal wrapped`	Elementy opakowywane elementem o nazwie odpowiadającej operacji.	✅ Zalecany
`rpc/literal`	Styl RPC — nazwa bloku odpowiada nazwie metody.	Akceptowalny
`document/encoded`	Możliwy, ale bez uzasadnienia.	⚠ Unikać
`rpc/encoded`	Historyczny — popularny przed XMLSchema. Zabrania go WS-I BP.	❌ Zabroniony

7.1.4.3. Przykład stylu rpc/encoded

Styl historyczny — definicja wiązania z `use="encoded"` i wynikowa wiadomość SOAP:

```
<message ....>
  <part name="message" type="xsd:string"/>
  <part name="repeat"  type="xsd:int"/>
</message>

<binding ....>
  <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="sayHello">
    <input>
      <soap:body use="encoded"
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
  </operation>
</binding>
```

Wynikowa wiadomość SOAP z typami wpisanymi inline przez `xsi:type`:

```
<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soap:Body>
    <sayHello>
      <message xsi:type="xsd:string">MyRpcMessage</message>
      <count    xsi:type="xsd:int">11</count>
    </sayHello>
  </soap:Body>
</soap:Envelope>
```

Styl `rpc/encoded` jest zakazany przez WS-I Basic Profile. Nie stosować w nowych serwisach.

7.1.5. Wiązanie SOAP 1.2

Definicja wiązania SOAP 1.2 jest oddzielną rekomendacją W3C. Elementy wiązania znajdują się w przestrzeni `http://schemas.xmlsoap.org/wsdl/soap12/`.

Różnice w stosunku do SOAP 1.1:

- `encodingStyle` przyjmuje tylko pojedyncze URI (nie listę),
- nowy atrybut `soapActionRequired` — wskazuje, czy serwer wymaga parametru `action`,
- `Content-Type` dla HTTP to `application/soap+xml` (zamiast `text/xml`).

Przykład wiązania SOAP 1.2:

```
<wsdl:binding ...
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/">
  <soap12:binding
    transport="http://schemas.xmlsoap.org/soap/http"
    style="document" />
  <wsdl:operation name="HelloWorld">
    <soap12:operation
      soapAction="http://example.com/HelloWorldRequest"
      soapActionRequired="true" />
    <wsdl:input>
      <soap12:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap12:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
```

Porównanie nagłówków HTTP dla SOAP 1.1 i 1.2 (gdy `soapActionRequired="true"`):

SOAP 1.1

```
Content-Type: text/xml
```

```
2 SOAPAction: "http://example.com/HelloWorldRequest"
```

SOAP 1.2

```
1 Content-Type: application/soap+xml; action=http://example.com/HelloWorldRequest
```

7.1.6. Wiązanie MIME

Wiązanie MIME może być używane razem z innymi wiązaniami (np. SOAP). Pozwala na podział wiadomości na różne części kontenera MIME — dla załączników binarnych, do których referencje znajdują się w kopercie SOAP.

Elementy rozszerzenia:

```
1 <mime:content part="nmtoken"? type="string"?/>
2
3 <mime:multipartRelated>
4   <mime:part> *
5     <!-- mime element -->
6   </mime:part>
7 </mime:multipartRelated>
8
9 <mime:mimeXml part="nmtoken"?/>
```

`mime:multipartRelated`

Agreguje poszczególne części MIME w jednym kontenerze.

`mime:content`

Łączy część z fragmentem MIME; pozwala określić typ MIME.

`mime:mimeXml`

Działa z wiązaniem SOAP; dołącza element XML zdefiniowany w schemacie.

`soap:body`

Dozwolone jest użycie elementu SOAP body jako fragmentu MIME.

Przykład operacji z odpowiedzią wieloczęściową (ciało SOAP + dokumentacja HTML + logo):

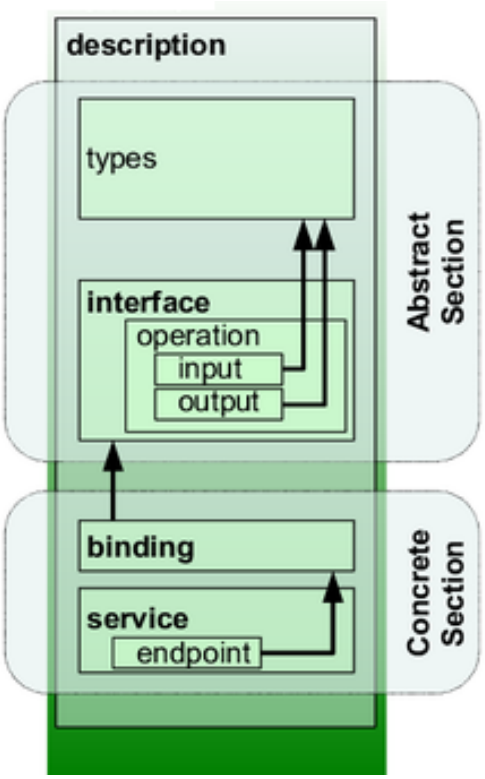
```
1 <operation name="GetCompanyInfo">
2   <soap:operation soapAction="http://example.com/GetCompanyInfo"/>
3   <input>
4     <soap:body use="literal"/>
5   </input>
6   <output>
7     <mime:multipartRelated>
8       <mime:part>
9         <soap:body parts="body" use="literal"/>
```

```
10       </mime:part>
11       <mime:part>
12         <mime:content part="docs" type="text/html"/>
13       </mime:part>
14       <mime:part>
15         <mime:content part="logo" type="image/gif"/>
16         <mime:content part="logo" type="image/jpeg"/>
17       </mime:part>
18     </mime:multipartRelated>
19   </output>
20 </operation>
```

7.2. Wprowadzenie do WSDL 2.0

Nowa wersja W3C znacznie poprawia strukturę i elastyczność. Główne zmiany:

WSDL 1.1	**WSDL 2.0**
Korzeń: `definitions`	Korzeń: `description`
`message` — pośredni opis wiadomości	Bezpośrednie wskazanie elementów ze schematu
`portType`	`interface` (obsługuje dziedziczenie interfejsów)
`port`	`endpoint`
4 typy operacji (one-way, request-response…)	8 wzorców MEP z pełną specyfikacją
Brak natywnego wiązania REST	Wiązanie HTTP z pełną obsługą metod GET/POST/PUT/DELETE
Brak dziedziczenia interfejsów	`interface` może rozszerzać inne interfejsy

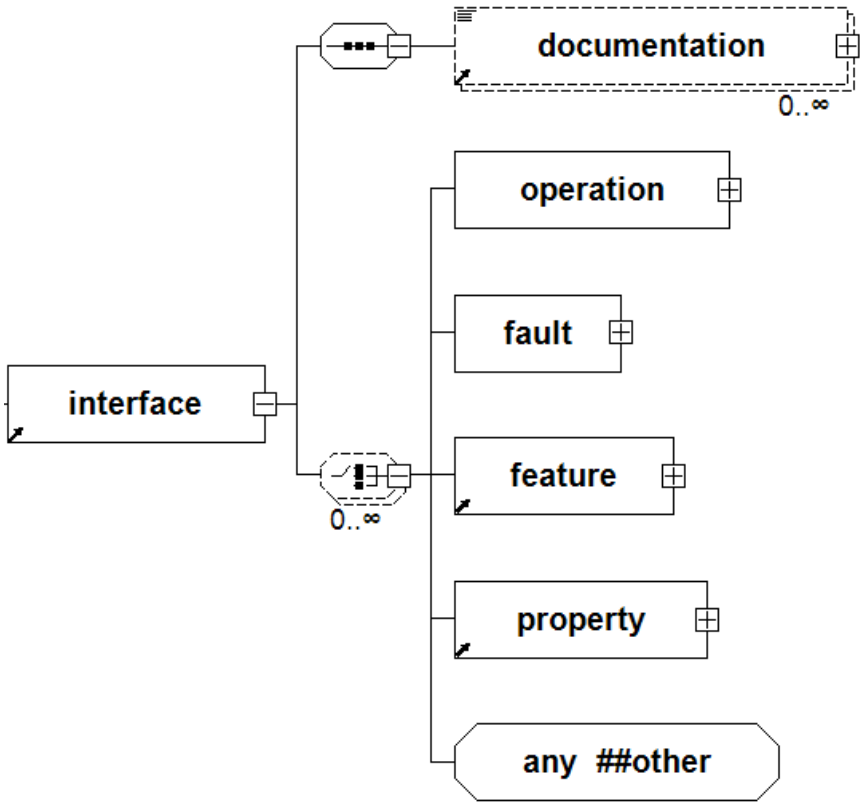


Rysunek 14. Struktura WSDL 2.0

W praktyce WSDL 2.0 nie zdobył szerokiego wsparcia w narzędziach Java EE. JAX-WS, CXF i Axis2 w wersji 1.x operują na WSDL 1.1. Na egzaminie 1Z0-897 obowiązuje wyłącznie WSDL 1.1.

7.2.1. Komponent interface

Podstawową zmianą jest element `interface`, zastępujący `portType` z WSDL 1.x. Stanowi zbiór abstrakcyjnych operacji i wiadomości. Nowością jest możliwość dziedziczenia po innych interfejsach. Błędy (`fault`) definiowane wewnątrz interfejsu, łączone z operacjami przez referencje.



Rysunek 15. Komponent interface WSDL 2.0

Ogólna struktura elementu `interface`:

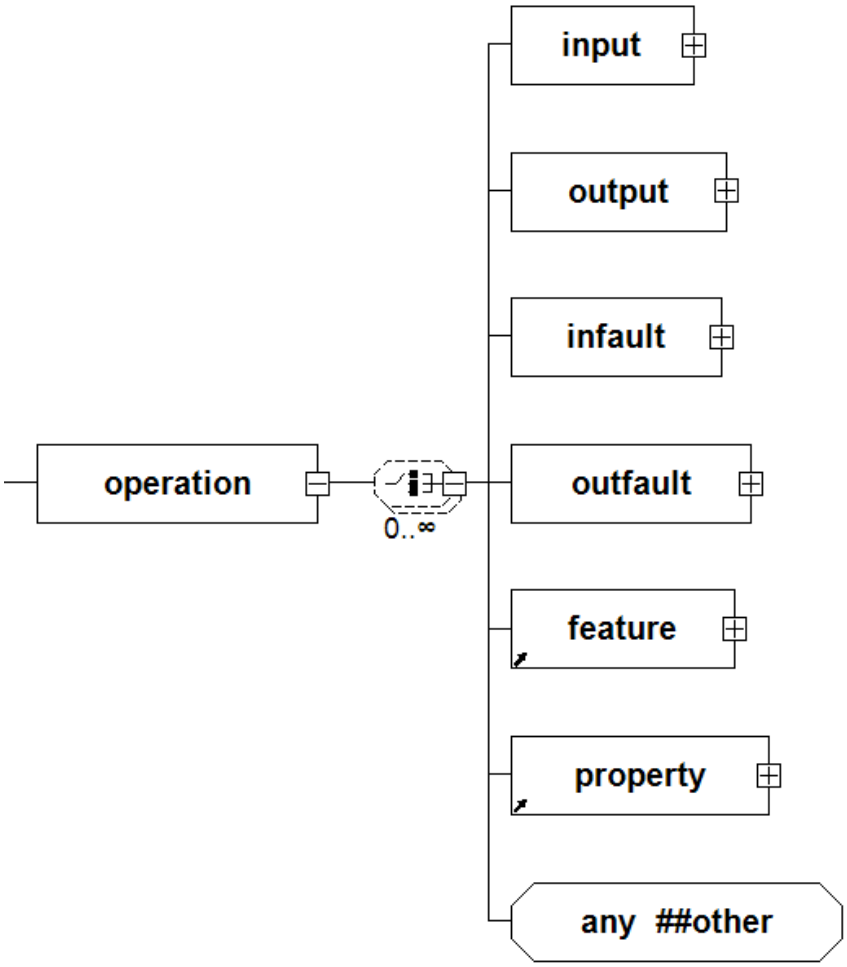
```
<description targetNamespace="xs:anyURI" >
  . . .
  <interface name="xs:NCName"
          extends="list of xs:QName"?
          styleDefault="list of xs:anyURI"? >

    <fault name="xs:NCName"
            element="xs:QName"? >
    </fault>*
    <operation name="xs:NCName"
            pattern="xs:anyURI"
            style="list of xs:anyURI"?
            wsdlx:safe="xs:boolean"? >

      <input  messageLabel="xs:NCName"?
            element="union of xs:QName, xs:Token"? >
      </input>*
      <output messageLabel="xs:NCName"?
            element="union of xs:QName, xs:Token"? >
      </output>*
      <infault  ref="xs:QName" messageLabel="xs:NCName"? >
      </infault>*
      <outfault ref="xs:QName" messageLabel="xs:NCName"? >
      </outfault>*
    </operation>*
  </interface>*
```

```
27   . . .
28 </description>
```

Atrybut `wsdlx:safe` wskazuje czy operacja powoduje zobowiązanie (np. `false` dla złożenia zamówienia, `true` dla zapytania o cenę).

7.2.2. Wzorce wymiany wiadomości (MEP) w WSDL 2.0

MEP	Opis
`in-only`	Jedna wiadomość od klienta, brak odpowiedzi (nawet błędu)
`robust-in-only`	Klient wysyła, nie oczekuje odpowiedzi pozytywnej (może być Fault)
`in-out`	Standardowy request-response
`in-opt-out`	Odpowiedź opcjonalna
`out-only`	Model push — serwer wysyła do klienta
`out-in`	Sekwencja odwrócona



Rysunek 16. Operacje WSDL 2.0

Przykład dziedziczenia interfejsu — `reservationInterface` rozszerza `messageLogInterface` i dziedziczy operację `opLogMessage`:

```
<description ...>
  ...
  <interface name="messageLogInterface">
    <operation name="opLogMessage"
               pattern="http://www.w3.org/ns/wsdl/out-only">
      <output messageLabel="out"
              element="ghns:messageLog" />
    </operation>
  </interface>

  <interface name="reservationInterface"
      extends="tns:messageLogInterface">
    <operation name="opCheckAvailability"
               pattern="http://www.w3.org/ns/wsdl/in-out"
               style="http://www.w3.org/ns/wsdl/style/iri"
               wsdlx:safe="true">
      <input  messageLabel="In"  element="ghns:checkAvailability" />
      <output messageLabel="Out" element="ghns:checkAvailabilityResponse" />
      <outfault ref="tns:invalidDataFault" messageLabel="Out"/>
    </operation>
  </interface>
  ...
```

```
23 </description>
```

7.2.3. Wiązanie SOAP w WSDL 2.0

Typ wiązania `http://www.w3.org/ns/wsdl/soap`. Atrybut `protocol` określa protokół komunikacyjny, `mep` wskazuje wzorzec wymiany wiadomości, `code` przypisuje kod błędu do SOAP Fault.

```
1 <binding name="reservationSOAPBinding"
2     interface="tns:reservationInterface"
3     type="http://www.w3.org/ns/wsdl/soap"
4     wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/">
5
6   <operation ref="tns:opCheckAvailability"
7     wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"/>
8
9   <fault ref="tns:invalidDataFault"
10    wsoap:code="soap:Sender"/>
11
12 </binding>
```

`soap-response` oznacza zapytanie przez HTTP GET i odpowiedź SOAP. Klasyczny wzorzec z POST: `http://www.w3.org/2003/05/soap/mep/request-response`.

7.2.4. Wiązanie HTTP w WSDL 2.0

Typ `http://www.w3.org/ns/wsdl/http` — atrybut `method` określa metodę HTTP:

```
1 <binding name="reservationHTTPBinding"
2          interface="tns:reservationInterface"
3          type="http://www.w3.org/ns/wsdl/http">
4     <operation ref="tns:opCheckAvailability"
5                whttp:location="/hotel/"
6                whttp:method="GET"/>
7     <operation ref="..." whttp:method="POST"/>
8 </binding>
```

7.3. WSDL a JAX-WS

JAX-WS wspiera dwa podejścia do tworzenia serwisów SOAP, w obu przypadkach WSDL jest centralnym elementem:

Contract-first (*WSDL-first*)

Najpierw tworzony jest dokument WSDL, następnie `wsimport` generuje interfejsy Javy, klasy SEI (*Service Endpoint Interface*) i klasy pośredniczące po stronie klienta. Podejście daje pełną kontrolę nad kontraktem — zalecane dla integracji z zewnętrznymi systemami.

Code-first (*Java-first*)

Klasy Javy adnotowane są adnotacjami JAX-WS (`@WebService`, `@WebMethod`, `@SOAPBinding`), a WSDL generowany jest automatycznie przez kontener przy wdrożeniu. Prostsze, lecz trudniejsze do kontrolowania dokładnego kształtu kontraktu.

Kolejne rozdziały opisują niskopoziomowe API przetwarzania XML (JAXP i JAXB), które stanowią fundament, na którym JAX-WS opiera generowanie i parsowanie wiadomości SOAP.

[1] Wsparcia dla WSDL 2.0 nie deklaruje JAX-WS ani CXF. Jedynym z popularnych frameworków dla Javy jest Axis2.

8. Niskopoziomowe przetwarzanie XML

Protokół SOAP i format WSDL opierają się na dokumentach XML. Choć JAX-WS ukrywa większość szczegółów przetwarzania XML za wygenerowanymi klasami i JAXB, znajomość niskopoziomowych API jest niezbędna w trzech sytuacjach: gdy piszemy własne handlery JAX-WS operujące bezpośrednio na wiadomości, gdy używamy interfejsu SAAJ (który bazuje bezpośrednio na modelu DOM), oraz gdy JAX-WS nie wystarcza i sami budujemy logikę parsowania lub serializacji.

Java nie wspierała parsowania i przetwarzania XML na początku swojego istnienia. W celu usunięcia tej luki powstały takie projekty jak *Sun Project X*, *Xerces* czy *Xalan*. Popularność tych rozwiązań zaowocowała potrzebą ich standaryzacji.

Sun zebrał najlepsze rozwiązania tworząc API, które wprowadził jako standardowe biblioteki do Javy SE. Konkretna implementacja dostarczana jest w postaci wtyczek, umożliwiając łatwe przełączanie się między dostawcami bez zmian w kodzie źródłowym.

Standardowymi API dostępnymi w języku Java są:

- *Simple API for XML* — SAX
- *Document Object Model* — DOM
- *Transformation API for XML* — TrAX (XSLT)
- *Streaming API for XML* — StAX

8.1. Wprowadzenie do JAXP

Jedną z luk jakie pojawiły się w tych ustandaryzowanych API był brak zdefiniowania metod pozyskiwania instancji konkretnych fabryk. Uzupełnia ją specyfikacja *Java API for XML Processing* (JAXP), która w sposób ustandaryzowany określa, jak należy tworzyć obiekty parserów i transformatorów.

Problem: Jak pozyskiwać instancje podstawowych fabryk? Rozwiązanie JAXP standaryzuje mechanizm pozyskiwania instancji fabryk w oparciu o mechanizm wtyczek.

```
1 DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
2 String location = "http://myserver/mycontent.xml";
3 try {
4     DocumentBuilder builder = factory.newDocumentBuilder();
5     Document document = builder.parse(location);
6 } catch (Exception se) {...}
```

Po wywołaniu metody `DocumentBuilderFactory.newInstance()` następuje wybór implementacji na podstawie pierwszej niepustej wartości uzyskanej z następujących

lokalizacji:

- zmienna systemowa `javax.xml.parsers.DocumentBuilderFactory`,
- zmienna o tej samej nazwie w pliku *lib/jaxp.properties* w katalogu domowym JRE,
- plik *META-INF/services/javax.xml.parsers.DocumentBuilderFactory* aplikacji,
- domyślna implementacja platformy (Apache Xerces 2 dla JVM Oracle).

Dla SAX i TrAX sekwencja poszukiwania jest identyczna, zmieniają się tylko nazwy zmiennych:

- DOM — `javax.xml.parsers.DocumentBuilderFactory`
- SAX — `javax.xml.parsers.SAXParserFactory`
- TrAX — `javax.xml.transform.TransformerFactory`

StAX używa kilku fabryk wejściowych:

- `javax.xml.stream.XMLInputFactory`
- `javax.xml.stream.XMLOutputFactory`
- `javax.xml.stream.XMLEventFactory`

8.2. Simple API for XML — SAX

Specyfikacja rozwijana przez *SAX Community_*[1]*. Zarys stworzył _David Megginson* w 1997 roku. Początkowo API było dostępne tylko dla Javy, ale z czasem powstały implementacje w innych językach.

Charakterystyka SAX:

parser strumieniowy (sekwencyjny)
: Dokument XML przetwarzany jest z ciągłego strumienia danych, bez możliwości nawigowania po dokumencie, w szczególności cofania się.

jednokierunkowy
: Możliwy jest tylko odczyt dokumentu. Nie ma metod pozwalających na modyfikację wczytywanego strumienia.

oparty na zdarzeniach (*event-based*)
: Parser podczas odczytywania dokumentu generuje zdarzenia (np. początek i koniec dokumentu lub elementu). Programista definiuje przetwarzanie w postaci obsługi tych zdarzeń.

typu *push*
: Parser odpowiedzialny jest za czytanie dokumentu krok po kroku. W czasie tego procesu wywołuje metody zwrotne (*callbacks*) na obiektach użytkownika (*handlers*), dostarczając informacji o aktualnym zdarzeniu — parser *wpycha* dane do aplikacji

użytkownika.

wydajny

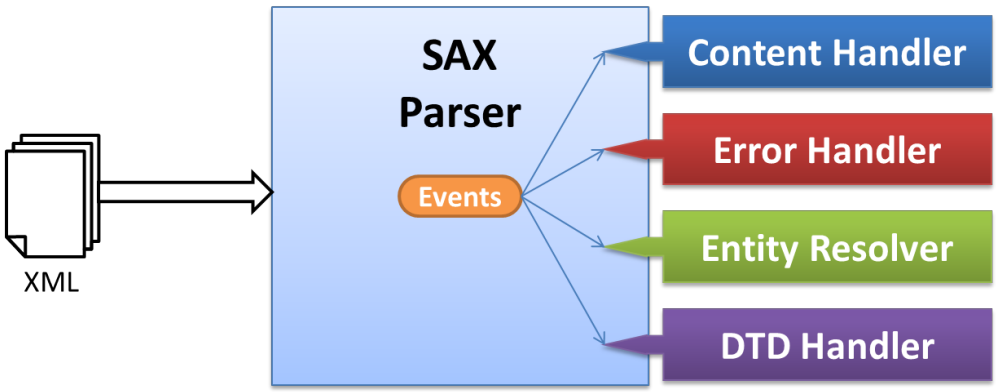
Parser przetwarza dokumenty w czasie liniowo zależnym od wielkości, a zapotrzebowanie na pamięć jest proporcjonalne do głębokości aktualnie przetwarzanego elementu, nie całego pliku.

trudny w użyciu

Model *push* wymaga od aplikacji śledzenia struktury przetwarzanego dokumentu i zapamiętywania potrzebnych informacji.

8.2.1. Schemat działania SAX parsera

Parser SAX sprawuje kontrolę nad aplikacją. Przed przystąpieniem do parsowania użytkownik musi ustawić handlery w obiekcie parsera. W architekturze SAX wyróżniamy następujące typy handlerów:



Rysunek 17. Schemat SAX parsera

Content Handler

Obiekt implementujący `org.xml.sax.ContentHandler` — zawierający zestaw metod zwrotnych otrzymujących informację o logicznej strukturze i zawartości dokumentu. W tym obiekcie zawarta jest logika biznesowa.

Error Handler

Obiekt implementujący `org.xml.sax.ErrorHandler` — zestaw 3 metod wywoływanych w przypadku błędu w parsowanym dokumencie.

Entity Resolver

Obiekty implementujące `org.xml.sax.EntityResolver` — dostarczają zewnętrznych encji DTD. W przypadku podania tylko publicznego identyfikatora, użytkownik musi zaimplementować mechanizm dostarczający treść wymaganej encji.

DTD Handler

Implementacja `org.xml.sax.DTDHandler` — obsługa nierozumianych przez parser encji

DTD, np. danych binarnych.

8.2.2. SAX API

SAX API zdefiniowane jest w pakietach:

Pakiet	Opis
`javax.xml.parsers`	Podstawowe interfejsy: `SAXParserFactory` i `SAXParser`
`org.xml.sax`	Klasy odpowiedzialne za podstawową funkcjonalność SAX
`org.xml.sax.helpers`	Klasy pomocnicze, m.in. domyślne implementacje handlerów (`DefaultHandler`)
`org.xml.sax.ext`	Rozszerzenia SAX API — bardziej szczegółowe handlery

Klasą wejściową API jest `SAXParserFactory` z metody `SAXParserFactory.newInstance()`. Fabryka pozwala na:

- otrzymanie obiektu parsera metodą `newSAXParser()`,
- konfigurację parsera: `setValidating` (domyślnie `false`), `setNamespaceAware` (domyślnie `false`), `setSchema`, `setFeature`.

Parser `SAXParser` może parsować dokumenty z wielu źródeł: `InputStream`, `File`, `URL`, `InputSource`.

Interfejs `org.xml.sax.ContentHandler` definiuje metody zwrotne:

Metoda	Opis
`startDocument()`	Początek dokumentu
`endDocument()`	Koniec dokumentu
`startElement(uri, localName, qName, atts)`	Otwarcie znacznika
`endElement(uri, localName, qName)`	Zamknięcie znacznika
`startPrefixMapping(prefix, uri)`	Pojawienie się przestrzeni nazw
`endPrefixMapping(prefix)`	Odwołanie prefiksu przestrzeni nazw
`characters(ch[], start, length)`	Tekst w dokumencie
`ignorableWhitespace(ch[], start, length)`	Białe znaki między elementami
`processingInstruction(target, data)`	Instrukcja procesowa `<? ?>`

Metoda	Opis
`setDocumentLocator(locator)`	Pozycja w dokumencie (linia, kolumna)
`skippedEntity(name)`	Zewnętrzna encja, której nie udało się zlokalizować

Interfejs `org.xml.sax.ErrorHandler`:

- `error(SAXParseException)` — błąd walidacji schematu lub DTD,
- `fatalError(SAXParseException)` — błąd uniemożliwiający dalsze parsowanie (np. złe formatowanie),
- `warning(SAXParseException)` — ostrzeżenie nie traktowane jako błąd.

Interfejs `org.xml.sax.EntityResolver` posiada jedną metodę `InputSource resolveEntity(String publicId, String systemId)`.

Przykład: jeśli dokument zawiera:

```
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.1.2//EN"
    "file:///n:/docbook/xml/docbookx.dtd">
```

to `publicId = "-//OASIS//DTD DocBook XML V4.1.2//EN"` i `systemId = "file:///n:/docbook/xml/docbookx.dtd"`.

```
public class MyResolver implements EntityResolver {
    public InputSource resolveEntity(String publicId, String systemId) {
        if (systemId.equals("file:///n:/docbook/xml/docbookx.dtd")) {
            MyReader reader = new MyReader();
            return new InputSource(reader);
        } else {
            return null; // wywołaj domyślną akcję
        }
    }
}
```

Użycie tej klasy pozwala na implementację przekierowań, cache'ów, wyciągania informacji z baz danych.

8.2.3. Przykładowa implementacja SAX parsera

Przykład opiera się na przeczytaniu pliku XML z listą książek:

```
<x:books xmlns:x="urn:books"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="urn:books books.xsd">
    <book id="bk001">
        <author>Hightower, Kim</author>
        <title>The First Book</title>
        <genre>Fiction</genre>
```

```
        <price>44.95</price>
        <pub_date>2000-10-01</pub_date>
        <review>An amazing story of nothing.</review>
    </book>
</x:books>
```

Klasa `DefaultHandler` implementuje wszystkie interfejsy handlerów z domyślnymi zachowaniami. Użytkownik nadpisuje tylko potrzebne metody:

```
DefaultHandler handler = new DefaultHandler() {
    @Override
    public void startElement(String uri, String localName, String qName,
                             Attributes attributes) throws SAXException {
        System.out.println();
        System.out.print(localName);
    }

    @Override
    public void characters(char[] ch, int start, int length) throws SAXException {
        if (new String(ch, start, length).trim().isEmpty()) return;
        System.out.print(": ");
        System.out.print(new String(ch, start, length));
    }

    @Override
    public void endElement(String uri, String localName, String qName)
                           throws SAXException {
        System.out.println();
        System.out.print("/" + localName);
    }
};
```

Użycie parsera:

```
SAXParserFactory spFactory = SAXParserFactory.newInstance();
spFactory.setNamespaceAware(true);
try {
    SAXParser parser = spFactory.newSAXParser();
    parser.parse(SAXExample.class.getResourceAsStream("books.xml"), handler);
} catch (ParserConfigurationException | SAXException | IOException e) {
    e.printStackTrace();
}
```

8.3. Document Object Model — DOM

SAX API jest jedną z pierwszych implementacji parserów XML. Choć jest on bardzo wydajny, to nie jest wygodny we wszystkich zastosowaniach. Programiści potrzebują mieć dostęp do reprezentacji XML w sposób swobodny. Z tej potrzeby powstał model obiektowy dokumentu (DOM), który pozwala na zaprezentowanie dokumentu XML jako drzewa obiektów i dostarcza metod poruszania się po nim.

Cechy DOM:

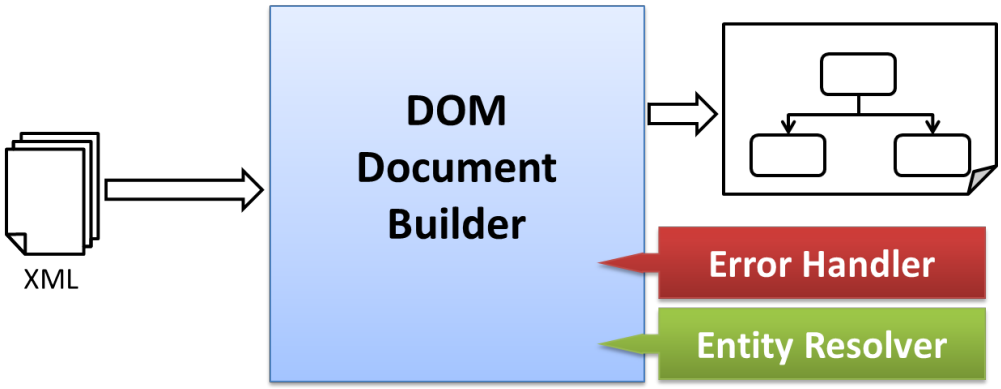
- niezależny od platformy model przedstawiania dokumentów opartych o XML,
- specyfikacja nadzorowana przez W3C,
- reprezentacja za pomocą hierarchii węzłów: elementów, atrybutów, tekstu, komentarzy,
- metody pozwalające na odczyt i modyfikację drzewa.

Specyfikacja DOM podzielona jest na poziomy:

- **DOM Level 1**: Core (obiekty reprezentujące strukturę), HTML
- **DOM Level 2**: Events, Style, Views, Traversal i Range
- **DOM Level 3**: modele zawartości z walidacją, ustandaryzowane ładowanie/zapisywanie plików, XPath

8.3.1. Idea działania parsera DOM

Mechanizm tworzenia reprezentacji DOM jest oparty na parserze SAX z odpowiednio zdefiniowanym `ContentHandler`. W wyniku parsowania otrzymujemy drzewo obiektów. Wadą jest dwukrotne wydłużenie czasu w stosunku do SAX i konieczność przechowywania całego obrazu dokumentu w pamięci.



Rysunek 18. Schemat DOM parsera

8.3.2. DOM API

API DOM zdefiniowane jest w pakietach:

Pakiet	Opis
`javax.xml.parsers`	`DocumentBuilderFactory` i `DocumentBuilder`
`org.w3c.dom`	Interfejsy DOM zgodne z rekomendacją W3C

`DocumentBuilderFactory.newInstance()` — fabryka pozwala na:

- otrzymanie obiektu `DocumentBuilder` metodą `newDocumentBuilder()`,
- konfigurację: `setValidating`, `setNamespaceAware`, `setSchema`, `setExpandEntityReferences`,

`setFeature`.

Interfejs `Document` — reprezentacja całego dokumentu XML, najwyżej położony węzeł (`Node`). Zawiera metody fabrykujące nowe węzły: `createElement`, `createAttribute`, `createTextNode`, `createComment`.

Wybrane metody interfejsu `Node`:

Metoda	Opis
`getLocalName()`	Nazwa węzła bez prefiksu przestrzeni nazw
`getNamespaceURI()`	Przestrzeń nazw węzła
`getNodeName()`	Pełna nazwa (prefiks + `LocalName`)
`getNodeValue()`	Wartość dla `Text`, `Comment`, `Attr`, `CDATASection`
`getNodeType()`	Typ węzła (stałe w interfejsie `Node`)
`getTextContent()`	Tekst zawarty w elemencie i jego potomkach
`appendChild(Node)`	Dodanie nowego potomka
`getAttributes()`	Lista atrybutów
`getChildNodes()`	Lista potomków

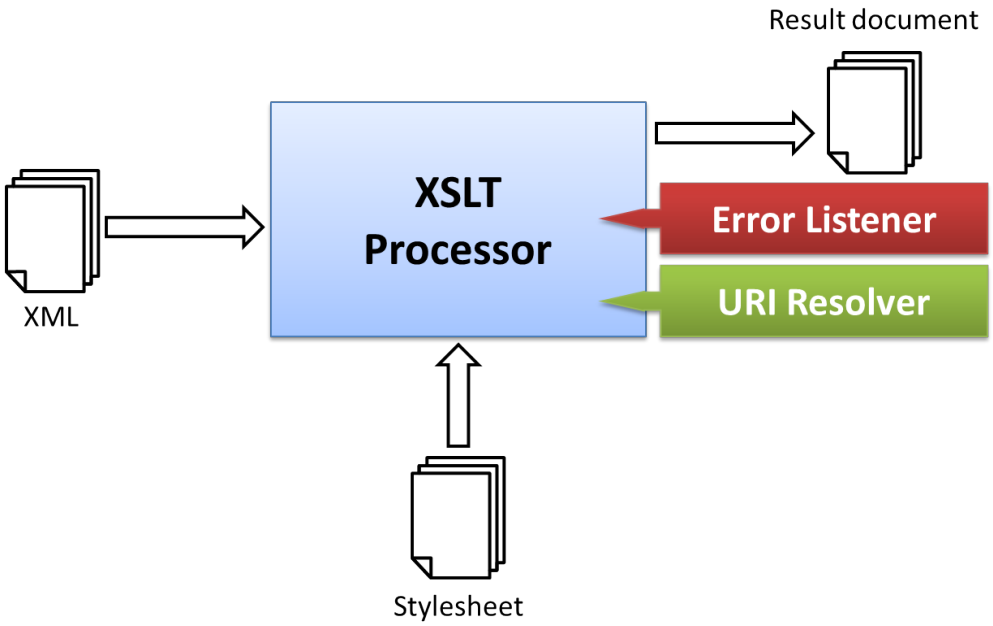
Interfejs `Element` dziedziczy po `Node` i dostarcza metod: `getAttribute`, `getAttributeNS`, `setAttribute`, `removeAttribute`, `getElementsByTagName`, `getElementsByTagNameNS`.

8.3.3. Przetwarzanie dokumentów XML w modelu DOM

```
DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
dbFactory.setNamespaceAware(true);
DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
Document doc = dBuilder.parse(DOMExample.class.getResourceAsStream("books.xml"));

Element root = doc.getDocumentElement();
NodeList nList = root.getElementsByTagName("book");
for (int i = 0; i < nList.getLength(); i++) {
    Node nNode = nList.item(i);
    if (nNode.getNodeType() == Node.ELEMENT_NODE) {
        Element e = (Element) nNode;
        System.out.println("id:     " + e.getAttribute("id"));
        System.out.println("author: " + e.getElementsByTagName("author")
                                          .item(0).getTextContent());
        System.out.println("title:  " + e.getElementsByTagName("title")
                                          .item(0).getTextContent());
    }
}
```

8.4. Transformation API For XML — TrAX

TrAX odpowiada na potrzebę łatwego przekształcania dokumentów XML. Instrukcje transformacyjne zapisane są w XSL (*Extensible Stylesheet Language*) w postaci plików XSLT. Wejściowy plik XML może być przekształcony do dowolnego formatu: XML, HTML, PDF, tekst. TrAX jest używany przez inne API jako mechanizm serializacji.



Rysunek 19. Schemat działania TrAX

8.4.1. TrAX API

TrAX zdefiniowany jest w pakietach:

Pakiet	Opis
`javax.xml.transform`	Główny pakiet — funkcjonalność procesora XSLT
`javax.xml.transform.dom`	Klasy opakowujące `Document` w `Source`/`Result`
`javax.xml.transform.sax`	Implementacje handlerów do bezpośredniej serializacji
`javax.xml.transform.stream`	Klasy opakowujące strumienie I/O (`InputStream`, `OutputStream`, `Reader`, `Writer`) jako `Source`/`Result`

TrAX używa interfejsów `Source` i `Result` jako abstrakcyjnego środka wsparcia wielu źródeł. Konkretne implementacje:

- `DOMResult` / `DOMSource`
- `SAXResult` / `SAXSource`
- `StreamResult` / `StreamSource`

Najważniejsze klasy i metody:

`TransformerFactory.newInstance()` — punkt wejściowy TrAX API:

- `newTransformer(Source)` — obiekt XSLT procesora dla podanego arkusza,
- `newTransformer()` — serializacja bez transformacji,
- `newTemplates(Source)` — fabryka `Transformer'ów (do wielowątkowego użycia),
- `setURIResolver(URIResolver)` — rozwijanie odnośników z `xsl:import`/`xsl:include`.

`Transformer` — przekształca dokument źródłowy na dokument wynikowy (jeden wątek na raz):

- `transform(Source xmlSource, Result outputTarget)` — wykonuje transformację,
- `setParameter(name, value)` — parametry XSLT,
- `setOutputProperty(name, value)` — właściwości wyjścia.

`Templates` — fabryka `Transformer'ów z już skompilowanym arkuszem styli (bezpieczna wielowątkowo).

8.5. Wprowadzenie do XSLT

Dokumenty XML są kontenerami dla danych, ale konkretne znaczniki nie niosą informacji o sposobie prezentacji graficznej. Istnieją dwie metody na dostarczenie tych informacji:

CSS

Ograniczone zastosowanie — pozwala tylko na formatowanie sposobu wyświetlania.

XSLT

Język transformacji — pozwala na modyfikowanie drzewa XML i konwertowanie go do dowolnych formatów. Transformacje opisuje się z użyciem rekurencji i wyrażeń XPath.

Przykład najprostszego arkusza:

```
1 <xsl:stylesheet version="1.0"
2                 xmlns:nn="urn:books"
3                 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4     <xsl:output method="xml"/>
5 </xsl:stylesheet>
```

Reguła przetwarzania — element `template`, atrybut `match` wskazuje element XPath:

```
1 <xsl:template match="book">
2     Book:
3     Author: <xsl:value-of select="author"/>
```

```
4     Title:  <xsl:value-of select="title"/>
5     Genre:  <xsl:value-of select="genre"/>
6 </xsl:template>
```

Element `apply-templates` — zastosowanie innych reguł do potomków:

```
1 <xsl:template match="nn:books">
2     Books list <xsl:apply-templates/>
3     End books of list
4 </xsl:template>
```

Pełny arkusz przetwarzający listę książek:

```
 1 <xsl:stylesheet version="1.0"
 2                 xmlns:nn="urn:books"
 3                 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 4     <xsl:output method="text"/>
 5     <xsl:strip-space elements="nn:books book" />
 6
 7     <xsl:template match="nn:books">
 8         Books list <xsl:apply-templates/>
 9         End books of list
10     </xsl:template>
11
12     <xsl:template match="book">
13         Book:
14         Author:   <xsl:value-of select="author"/>
15         Title:    <xsl:value-of select="title"/>
16         Genre:    <xsl:value-of select="genre"/>
17         Pub_date: <xsl:value-of select="pub_date"/>
18         Price:    <xsl:value-of select="price"/>
19     </xsl:template>
20 </xsl:stylesheet>
```

8.5.1. Instrukcje warunkowe i pętle

Element `xsl:if` realizuje warunkowe przetwarzanie — wyrażenie testowe jest wyrażeniem XPath. Każdy niepusty wynik wyrażenia, który nie jest równy `false()`, uznawany jest za prawdę:

```
1 <xsl:if test="wyrazenie">
2     ...jeśli prawda to...
3 </xsl:if>
```

Nie ma instrukcji `if/else` — zastępuje ją element `xsl:choose`:

```
1 <xsl:choose>
2     <xsl:when test="wyrazenie">
3         ... szablon ...
4     </xsl:when>
5     <xsl:otherwise>
```

```
6         ... szablon ...
7     </xsl:otherwise>
8 </xsl:choose>
```

Pętla `xsl:for-each` iteruje po elementach wskazanych wyrażeniem XPath:

```
1 <xsl:for-each select="wyrazenie">
2     ... szablon ...
3 </xsl:for-each>
```

Przykład — szablon `books` przepisany z użyciem pętli:

```
 1 <xsl:template match="nn:books">
 2     Books list
 3     <xsl:for-each select="book">
 4         Book:
 5         Author: <xsl:value-of select="author"/>
 6         Title:  <xsl:value-of select="title"/>
 7         Genre:  <xsl:value-of select="genre"/>
 8     </xsl:for-each>
 9     End books of list
10 </xsl:template>
```

8.6. Streaming API for XML — StAX

StAX jest najmłodszym ustandaryzowanym API do przetwarzania XML — pomostem między szybkością SAX i swobodą DOM.

Cecha	Opis
Strumieniowy	Dokument przetwarzany sekwencyjnie od początku do końca; cofanie nie jest obsługiwane
Dwukierunkowy i symetryczny	Odczyt i zapis — API wygląda podobnie dla obu operacji
Oparty o zdarzenia	Traktuje dokument jak sekwencję zdarzeń
Typ *pull*	Programista kontroluje, kiedy zażąda kolejnego zdarzenia (w przeciwieństwie do SAX *push*)

StAX dostępny jest w dwóch odmianach:

Cursor API

Prosty kursor poruszający się po dokumencie, wskazujący jeden element na raz. Wszystkie informacje zwraca jako `String`i — minimalne zapotrzebowanie na pamięć.

Event iterator API

Bardziej zaawansowane — zwraca obiekty `XMLEvent` z pełną informacją o zdarzeniu. Może być wykorzystywane do tworzenia potoków (*pipelines*).

8.6.1. Przegląd API

Pakiety:

- `javax.xml.stream` — podstawowe interfejsy, klasy i fabryki,
- `javax.xml.stream.events` — implementacje typów zdarzeń,
- `javax.xml.stream.util` — pomocnicze interfejsy.

Trzy fabryki (każda przez `newInstance()`):

Fabryka	Opis
`XMLInputFactory`	Tworzy `XMLStreamReader` lub `XMLEventReader`
`XMLOutputFactory`	Tworzy `XMLStreamWriter` lub `XMLEventWriter`
`XMLEventFactory`	Tworzy obiekty `XMLEvent` do strumienia wyjściowego

Interfejsy kursor API:

```
// Odczyt – XMLStreamReader
public interface XMLStreamReader {
    public int     next()        throws XMLStreamException;
    public boolean hasNext()     throws XMLStreamException;
    public String  getText();
    public String  getLocalName();
    public String  getNamespaceURI();
    // ...
}

// Zapis – XMLStreamWriter
public interface XMLStreamWriter {
    public void writeStartElement(String localName)  throws XMLStreamException;
    public void writeEndElement()                    throws XMLStreamException;
    public void writeCharacters(String text)         throws XMLStreamException;
    // ...
}
```

Interfejsy event iterator API:

```
// Odczyt – XMLEventReader
public interface XMLEventReader extends Iterator {
    public XMLEvent nextEvent() throws XMLStreamException;
    public boolean  hasNext();
    public XMLEvent peek()      throws XMLStreamException;
    public String   getElementText() throws XMLStreamException;
}

```

```
// Zapis – XMLEventWriter
public interface XMLEventWriter {
    public void flush() throws XMLStreamException;
    public void close() throws XMLStreamException;
    public void add(XMLEvent e) throws XMLStreamException;
}
```

8.6.2. Przykładowe implementacje

Cursor API — odczyt

```
XMLInputFactory inputFactory = XMLInputFactory.newFactory();
XMLStreamReader reader = inputFactory.createXMLStreamReader(
    StAXExample.class.getResourceAsStream("books.xml"));

while (reader.hasNext()) {
    switch (reader.next()) {
        case XMLStreamConstants.START_ELEMENT:
            System.out.print(reader.getLocalName());
            break;
        case XMLStreamConstants.CHARACTERS:
            if (!reader.getText().trim().isEmpty()) {
                System.out.print(": ");
                System.out.print(reader.getText());
            }
            break;
    }
}
```

Cursor API — zapis

```
XMLOutputFactory outputFactory = XMLOutputFactory.newFactory();
XMLStreamWriter writer = outputFactory.createXMLStreamWriter(System.out);

writer.writeStartDocument();
writer.writeStartElement("x", "books", "urn:books");
writer.writeNamespace("x", "urn:books");

writer.writeStartElement("book");
writer.writeAttribute("id", "book1");
writer.writeStartElement("author");
writer.writeCharacters("Hightower, Kim");
writer.writeEndElement();
writer.writeEndElement(); // book
writer.writeEndElement(); // books

writer.flush();
writer.close();
```

Event iterator API — odczyt

```
XMLInputFactory inputFactory = XMLInputFactory.newFactory();
XMLEventReader reader = inputFactory.createXMLEventReader(
    StAXExample.class.getResourceAsStream("books.xml"));

while (reader.hasNext()) {
    XMLEvent event = reader.nextEvent();
```

```
    switch (event.getEventType()) {
        case XMLEvent.START_ELEMENT:
            System.out.print(event.asStartElement().getName());
            break;
        case XMLEvent.CHARACTERS:
            if (!event.asCharacters().getData().trim().isEmpty()) {
                System.out.print(": ");
                System.out.print(event.asCharacters().getData());
            }
            break;
    }
}
```

Event iterator API — zapis (XMLEventWriter + XMLEventFactory)

```
XMLOutputFactory outputFactory = XMLOutputFactory.newFactory();
XMLEventFactory eventFactory = XMLEventFactory.newFactory();
XMLEventWriter writer = outputFactory.createXMLEventWriter(System.out);

writer.add(eventFactory.createStartDocument());
writer.add(eventFactory.createStartElement("x", "urn:books", "books"));
writer.add(eventFactory.createNamespace("x", "urn:books"));

writer.add(eventFactory.createStartElement(new QName("book"), null, null));
writer.add(eventFactory.createAttribute("id", "book1"));
writer.add(eventFactory.createStartElement(new QName("author"), null, null));
writer.add(eventFactory.createCharacters("Hightower, Kim"));
writer.add(eventFactory.createEndElement(new QName("author"), null));
writer.add(eventFactory.createStartElement("", null, "title"));
writer.add(eventFactory.createCharacters("The First Book"));
writer.add(eventFactory.createEndElement("", null, "title"));
writer.add(eventFactory.createEndElement(new QName("book"), null));
writer.add(eventFactory.createEndElement("x", "urn:books", "books"));
writer.add(eventFactory.createEndDocument());

writer.flush();
writer.close();
```

`XMLEventFactory` tworzy instancje zdarzeń dodawanych do strumienia przez `XMLEventWriter`. Fabrykę uzyskuje się z `XMLEventFactory.newFactory()`:

```
public interface XMLEventFactory {
    Attribute     createAttribute(QName name, String value);
    Characters    createCData(String content);
    DTD           createDTD(String dtd);
    Namespace     createNamespace(String namespaceURI);
    StartDocument createStartDocument(String encoding);
    StartElement  createStartElement(QName name, Iterator attributes, Iterator namespaces);
    // ...
}
```

8.6.3. Filtrowanie strumienia

Używając specjalnych metod fabryki `XMLInputFactory`, można przekazać do parsera implementację filtra, który zmodyfikuje otrzymywany strumień zdarzeń. Dostępne są

dwa interfejsy:

- `javax.xml.stream.StreamFilter` — do pracy z `XMLStreamReader`,
- `javax.xml.stream.EventFilter` — do pracy z `XMLEventReader`.

Oba posiadają metodę `accept` przyjmującą zdarzenie i zwracającą `boolean` — wartość `false` oznacza odfiltrowanie zdarzenia.

8.6.4. Raportowanie błędów i rozwiązywanie encji

StAX oferuje dwa interfejsy, których implementacje dołącza się do parsera metodami fabryki:

`javax.xml.stream.XMLResolver`

Posiada jedną metodę `resolveEntity(String publicID, String systemID, String baseURI, String namespace)`, która może zwrócić `InputStream`, `XMLStreamReader`, `XMLEventReader` lub `null`.

`javax.xml.stream.XMLReporter`

Posiada metodę `report(String message, String errorType, Object relatedInformation, Location location)`, do której przekazywane są informacje o błędach.

8.7. Porównanie API przetwarzania XML

Cecha	SAX	DOM	StAX	TrAX
Kierunek	Odczyt	Odczyt i zapis	Odczyt i zapis	Transformacja
Model	Zdarzeniowy (push)	Drzewiasty	Zdarzeniowy (pull)	XSLT / arkusz styli
Pamięć	Minimalna	Cały dokument	Minimalna	Zależy od źródła
Wydajność	Wysoka	Niższa	Wysoka	Umiarkowana
Nawigacja	Brak	Dowolna	Jednokierunkowa	Brak
Modyfikacja	Nie	Tak	Tak (zapis)	Tak (transformacja)
Typowy przypadek użycia	Parsowanie dużych dokumentów	Modyfikacja drzewa	Generowanie/parsowanie XML	Konwersja formatu

8.8. JAXP a stos web serwisów

W praktyce bezpośrednie użycie JAXP przez programistę JAX-WS jest rzadkie — JAX-WS i JAXB przejmują parsowanie wewnętrznie. Jednak znajomość tych API jest niezbędna w trzech sytuacjach:

- **Handlery JAX-WS** (`SOAPHandler`) — operują bezpośrednio na modelu SAAJ opartym na DOM; konieczna znajomość interfejsów `Node`, `Element`, `NodeList`,
- **SAAJ API** — budowanie i parsowanie wiadomości SOAP bez JAX-WS używa DOM,
- **Własna serializacja** — gdy wbudowane wsparcie JAXB dla danego formatu jest niewystarczające.

Kolejny rozdział opisuje JAXB — mechanizm wiązania XML z obiektami Javy, który jest wewnętrznym silnikiem serializacji i deserializacji w JAX-WS.

[1] http://sax.sourceforge.net/

9. Java Architecture for XML Binding — JAXB

JAXB jest wewnętrznym silnikiem JAX-WS — każde wywołanie operacji serwisu SOAP skutkuje automatyczną deserializacją (*unmarshalling*) ciała wiadomości XML na parametry metody Javy, a następnie serializacją (*marshalling*) wartości zwracanej z powrotem do XML. Programista JAX-WS zazwyczaj nie wywołuje JAXB bezpośrednio, ale zrozumienie jego mechanizmu jest niezbędne do diagnozowania problemów z serializacją i do dostosowywania kształtu generowanego XML.

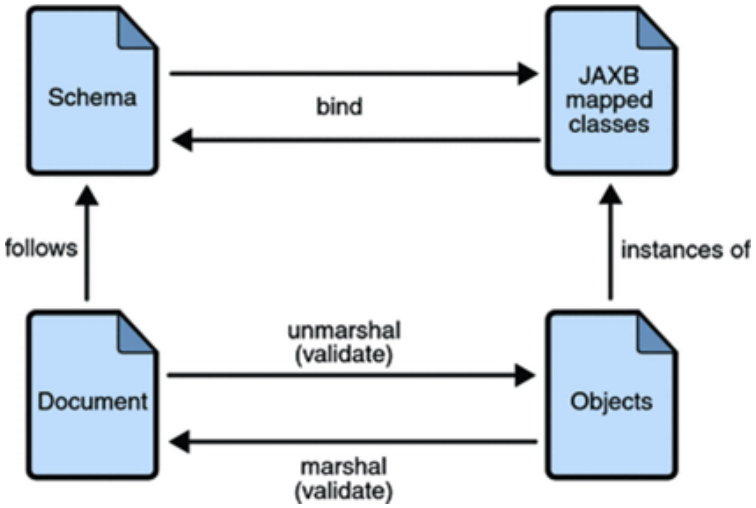
Specyfikacja XML nie nakłada na dokumenty innych ograniczeń poza poprawnością formatu. Jednak do efektywnej komunikacji z użyciem XML systemy potrzebują czegoś więcej — schematów, które definiują składnię i semantykę drzew XML. Używanie parserów DOM czy SAX wraz ze wzrostem liczby możliwych dokumentów staje się uciążliwe. W celu rozwiązania tego problemu powstały systemy takie jak JAXB, wiążące XML z obiektami Javy.

Data Binding oznacza ideę łączenia ze sobą różnych form reprezentacji tych samych danych. Mechanizm ten przyjmuje różne postacie, np:

- model-widok-kontroler,
- łączenie kontrolek GUI ze zmiennymi aplikacji,
- mapowanie struktur XML na klasy.

9.1. Wprowadzenie do JAXB

JAXB jest specyfikacją API stworzoną w *Java Community Process* (wersja 1.0 — JSR 31, 2.0 — JSR 222). Pozwala na dwukierunkową transformację dokumentów XML na obiekty Javy i jest częścią Javy SE. Począwszy od wersji 2.0 konfiguracja wiązania odbywa się poprzez adnotowanie klas, pól i metod. Posiada kompilator, który na podstawie podanego schematu XML generuje gotowy kod źródłowy z adnotacjami, oraz generator, który z adnotowanych klas generuje schematy.



Rysunek 20. Model JAXB

Formalne wiązanie (*bind*) odbywa się na poziomie schematu XML z odpowiednio adnotowanymi klasami Javy. W czasie wykonania aplikacja przyjmuje dokumenty XML i tworzy instancje adnotowanych klas (*unmarshalling*). Możliwa jest również serializacja obiektów Javy na dokumenty XML (*marshalling*). W czasie tych operacji możliwa jest walidacja dokumentu.

9.2. Architektura JAXB

W JAXB wyróżniamy trzy komponenty:

Kompilator schematu (*schema compiler*)

Powiązuje gotowy schemat z elementami (klasami) programu. Dostępny w pakiecie Java SDK jako program `xjc`.

```
1 JDK_HOME\bin\xjc.exe books.xsd
```

Program generuje pakiet z klasami Javy i pomocniczą klasę fabryki `ObjectFactory`. Zachowanie kompilatora można dostosować plikiem XML z *Java Customization Binding Declarations* (parametr `-b`).

Generator schematu (*schema generator*)

Wykonuje odwrotne zadanie — z adnotowanych klas Javy tworzy schemat XML. Dostępny jako program `schemagen` lub operacja w czasie wykonania (schemat generowany dynamicznie).

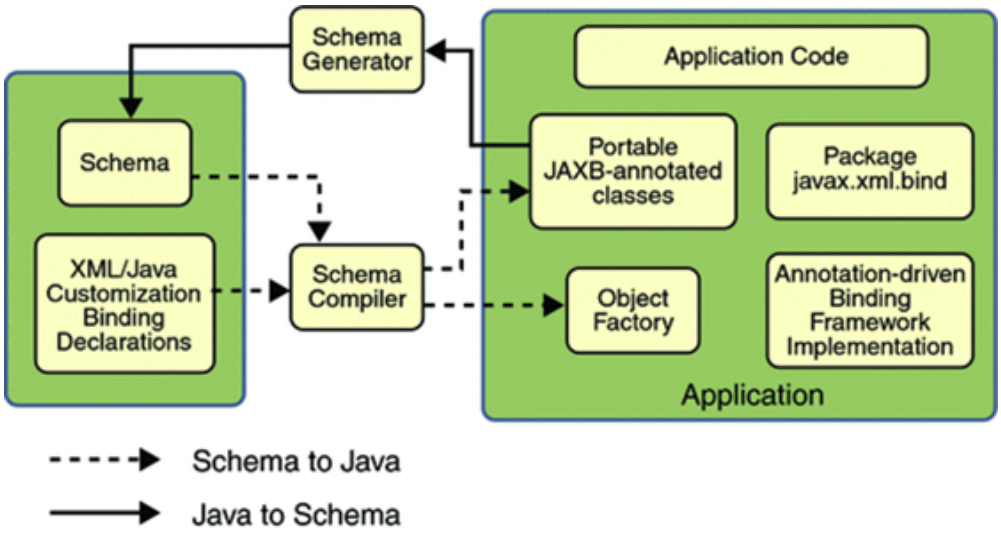
```
1 JDK_HOME\bin\schemagen.exe Foo.java Bar.java
```

Binding runtime framework

Dostarcza głównej funkcjonalności — przeprowadza faktyczną konwersję XML obiekty Javy (*marshalling* / *unmarshalling*) wraz z opcjonalną walidacją zgodności ze schematem.

JAXB wspiera trzy strategie tworzenia wiązania:

- **Rozpoczęcie od schematu** — kompilator schematu generuje klasy Javy.
- **Rozpoczęcie od modelu klas** — programista manualnie dodaje adnotacje JAXB do istniejących klas.
- **Meet in the middle** — dopasowanie adnotacji w klasach do istniejącego schematu.



Rysunek 21. Strategie JAXB

9.2.1. Reprezentacja XML w Javie

Dokument XML reprezentowany jest jako drzewo obiektowe *object content tree*. JAXB używa następujących typów dla wiązań:

Typ	Opis
value class	Klasy reprezentujące złożone typy schematu `complexType` — generowane przez JAXB
property	Proste typy: atrybuty lub pola z wartościami tekstowymi; typ referencyjny dla *value class*
Enum	Typy enumeracyjne ze schematu
collection	Lista obiektów Javy — gdy elementy mogą się powtarzać
DOM Element	Fragment dokumentu używany jako węzeł modelu DOM

Klasa `ObjectFactory` tworzona jest dla każdego schematu i dostarcza metody fabrykujące nowe instancje.

9.2.2. `javax.xml.bind.JAXBContext`

Wejście do JAXB API. Instancję otrzymuje się przez statyczną metodę `newInstance`:

```
// Inicjalizacja na podstawie listy pakietów
JAXBContext jc = JAXBContext.newInstance("com.acme.foo:com.acme.bar");

// Inicjalizacja na podstawie listy klas
JAXBContext jc = JAXBContext.newInstance(com.acme.foo.Foo.class);
```

Tworzenie `JAXBContext` jest kosztowne — specyfikacja wymaga, aby rozszerzenia klasy były *thread-safe* i mogły być cachowane w całej aplikacji.

Interfejs klasy `JAXBContext`:

```
public abstract class JAXBContext {
    static JAXBContext newInstance(String contextPath);
    static JAXBContext newInstance(String contextPath, ClassLoader cl);
    static JAXBContext newInstance(Class... classesToBeBound);
    abstract Unmarshaller createUnmarshaller();
    abstract Marshaller   createMarshaller();
    abstract JAXBIntrospector createJAXBIntrospector();
    <T> Binder<T> createBinder(Class<T> domType);
    Binder<org.w3c.dom.Node> createBinder();
    void generateSchema(SchemaOutputResolver);
}
```

9.2.3. Obiekty `Marshaller` i `Unmarshaller`

9.2.3.1. Unmarshaller

Deserializacja dokumentów XML na drzewo obiektowe JAXB. Metoda `setSchema` włącza walidację.

```
JAXBContext jc = JAXBContext.newInstance("com.example");
Unmarshaller u = jc.createUnmarshaller();
Object o = u.unmarshal(new File("books.xml"));
```

Metoda `unmarshal` obsługuje wiele źródeł: `java.io.File`, `InputStream`, `java.net.URL`, `javax.xml.transform.Source`, `org.w3c.dom.Node`, `SAXSource`, `XMLStreamReader`, `XMLEventReader`.

9.2.3.2. Marshaller

Serializacja drzewa obiektowego na dokumenty XML:

```
JAXBContext jc = JAXBContext.newInstance("com.example");
Unmarshaller u = jc.createUnmarshaller();
Object element = u.unmarshal(new File("books.xml"));
```

```
4
5 Marshaller m = jc.createMarshaller();
6 OutputStream os = new FileOutputStream("newbooks.xml");
7 m.marshal(element, os);
```

Metoda `marshal` obsługuje wiele celów: `File`, `OutputStream`, `Writer`, `org.w3c.dom.Node`, `ContentHandler`, `javax.xml.transform.Result`, `XMLStreamWriter`, `XMLEventWriter`.

9.2.4. `javax.xml.bind.Binder`

`Binder` (dodany w JAXB 2.0) pozwala na synchronizację między załadowanym dokumentem XML a obiektem JAXB — można jednocześnie pracować nad tym samym dokumentem za pomocą DOM API i drzewa obiektowego JAXB, a oba modele pozostaną zsynchronizowane.

Najważniejsze metody:

Metoda	Opis
`unmarshal(XmlNode)`	Tworzy element JAXB z węzła XML
`getJAXBNode(XmlNode)`	Obiekt JAXB skojarzony z danym węzłem XML
`getXMLNode(jaxbObject)`	Węzeł XML skojarzony z danym elementem JAXB
`updateJAXB(XmlNode)`	Propaguje zmiany z XML do JAXB
`updateXML(jaxbObject)`	Propaguje zmiany z JAXB do XML

Przykład użycia — plik `input.xml` z elementami niezwiązanymi i komentarzami:

```
1 <customer>
2     <UNMAPPED_ELEMENT_1/>
3     <name>Jane Doe</name>
4     <!-- COMMENT 1 -->
5     <address>
6         <UNMAPPED_ELEMENT_2/>
7         <street>1 A Street</street>
8         <!-- COMMENT 2 -->
9         <UNMAPPED_ELEMENT_3/>
10        <city>Any Town</city>
11    </address>
12    <!-- COMMENT 3 -->
13    <UNMAPPED_ELEMENT_4/>
14    <phone-number type="home">555-HOME</phone-number>
15    <!-- COMMENT 4 -->
16    <phone-number type="cell">555-CELL</phone-number>
17    <UNMAPPED_ELEMENT_5/>
18    <!-- COMMENT 5 -->
19 </customer>
```

```
1 DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
2 DocumentBuilder db = dbf.newDocumentBuilder();
```

```
Document document = db.parse(new File("input.xml"));

JAXBContext jc = JAXBContext.newInstance(Customer.class);
Binder<Node> binder = jc.createBinder();

Customer customer = (Customer) binder.unmarshal(document);
customer.getAddress().setStreet("2 NEW STREET");

PhoneNumber workPhone = new PhoneNumber();
workPhone.setType("work");
workPhone.setValue("555-WORK");
customer.getPhoneNumbers().add(workPhone);

binder.updateXML(customer); // synchronizacja z drzewem DOM

TransformerFactory tf = TransformerFactory.newInstance();
Transformer t = tf.newTransformer();
t.transform(new DOMSource(document), new StreamResult(System.out));
```

Wynik zachowa wszystkie niezwiązane elementy i komentarze z oryginalnego pliku (zmodyfikowany adres i dodany nowy numer telefonu):

```
<customer>
    <UNMAPPED_ELEMENT_1/>
    <name>Jane Doe</name>
    <!-- COMMENT 1 -->
    <address>
        <UNMAPPED_ELEMENT_2/>
        <street>2 NEW STREET</street>
        <!-- COMMENT 2 -->
        <UNMAPPED_ELEMENT_3/>
        <city>Any Town</city>
    </address>
    <!-- COMMENT 3 -->
    <UNMAPPED_ELEMENT_4/>
    <phone-number type="home">555-HOME</phone-number>
    <!-- COMMENT 4 -->
    <phone-number type="cell">555-CELL</phone-number>
    <phone-number type="work">555-WORK</phone-number>
    <UNMAPPED_ELEMENT_5/>
    <!-- COMMENT 5 -->
</customer>
```

9.2.5. Sprawdzanie poprawności

JAXB wspiera trzy typy walidacji:

(un)marshal-time validation

Wykrywa błędy podczas odczytywania lub serializacji dokumentu. Użytkownik implementuje `javax.xml.bind.ValidationEventHandler` i rejestruje go w `Unmarshaller` lub `Marshaller`.

```
public interface ValidationEventHandler {
    public boolean handleEvent(ValidationEvent event);
```

```
3 }
```

Metoda `handleEvent` zwraca `true` — kontynuuj, lub `false` — zakończ z wyjątkiem. Implementacja musi zwrócić `false` po błędzie krytycznym.

on-demand validation

Przestarzała od JAXB 2.0 — sprawdzanie poprawności drzewa obiektów w dowolnym momencie.

fail-fast validation

Opcjonalna — natychmiastowa reakcja na nieprawidłowe dane (wyjątek przy wprowadzeniu błędnej wartości).

JAXB definiuje dwie implementacje `ValidationEventHandler`:

- domyślna — loguje i przerywa przy pierwszym błędzie,
- `javax.xml.bind.util.ValidationEventCollector` — zbiera wszystkie ostrzeżenia i błędy, przerywa tylko po błędzie krytycznym.

9.2.6. Adnotacje mapujące JAXB

Drzewami obiektowymi JAXB są zwykłe klasy Javy (POJO) z adnotacjami sterującymi wiązaniem. Wszystkie adnotacje zdefiniowane są w pakietach `javax.xml.bind.annotation` i `javax.xml.bind.annotation.adapters`.

Adnotacja	Opis
`@XmlRootElement`	Mapuje klasę na element globalny (korzeń) w XML
`@XmlType`	Definiuje typ XML w schemacie (`complexType` lub `simpleType`)
`@XmlElement`	Domyślne mapowanie — pole na lokalny element XML
`@XmlAttribute`	Mapuje pole na atrybut XML (tylko typy proste lub ich listy)
`@XmlTransient`	Ignoruje pole/właściwość przez JAXB
`@XmlSchema`	Globalne właściwości pakietu (w `package-info.java`)
`@XmlElementWrapper`	Opakowuje listę w dodatkowy element
`@XmlAccessorType`	Określa jakich elementów klasy JAXB używa domyślnie
`@XmlJavaTypeAdapter`	Wskazuje klasę `XmlAdapter<ValueType,BoundType>` do niestandardowej konwersji typów (np. `Date`, `Map`, typy nieobsługiwane domyślnie)

Przykład klasy JAXB

```
@XmlRootElement
public class Customer {
    @XmlElement(name="Name")
    public String name;

    @XmlElement(name="Age")
    public int age;

    @XmlAttribute
    public int id;
}
```

JAXB wymaga, aby każda klasa podlegająca unmarshallingowi posiadała bezargumentowy konstruktor `public` lub `protected` — brak takiego konstruktora skutkuje wyjątkiem `com.sun.xml.bind.v2.runtime.IllegalAnnotationsException` w czasie działania programu. Klasy generowane przez `xjc` zawsze go posiadają; przy ręcznej adnotacji należy pamiętać o jego dodaniu, jeśli jest zdefiniowany jakikolwiek inny konstruktor.

`@XmlRootElement` — *mapowanie na element globalny*

```
@XmlRootElement
class Point {
    int x;
    int y;

    Point() {}                          // wymagany przez JAXB do unmarshallingu
    Point(int _x, int _y) { x = _x; y = _y; }
}
```

Mapuje się na:

```
<point><x>3</x><y>5</y></point>
```

Odpowiada następującemu fragmentowi schematu XML:

```
<xs:element name="point" type="Point"/>
<xs:complexType name="Point">
    <xs:sequence>
        <xs:element name="x" type="xs:int"/>
        <xs:element name="y" type="xs:int"/>
    </xs:sequence>
</xs:complexType>
```

Adnotacja `@XmlRootElement` posiada dwa atrybuty:

- `name` — nazwa lokalnego elementu; domyślnie przyjmowana jest nazwa klasy,
- `namespace` — przestrzeń nazw elementu; domyślnie budowana jest na podstawie nazwy pakietu.

`@XmlType` — *z* `propOrder` *i* `@XmlAccessorType`

```
1 @XmlType(propOrder = {"street", "city", "state", "zip", "name"})
2 @XmlAccessorType(XmlAccessType.FIELD)
3 public class USAddress {
4     public String name;
5     public String street;
6     public String city;
7     public String state;
8     public BigDecimal zip;
9 }
```

Atrybuty `@XmlType`: `name`, `namespace`, `propOrder`.

Wartości `XmlAccessType`:

- `FIELD` — wszystkie niestatyczne pola,
- `NONE` — nic (chyba że ma jawną adnotację),
- `PROPERTY` — getery/setery zgodne z JavaBeans,
- `PUBLIC_MEMBER` — domyślna wartość, wszystkie publiczne pola i właściwości.

Właściwości `@XmlElement`: `name`, `defaultValue`, `nillable`, `required`, `type`. Właściwości `@XmlAttribute`: `name`, `namespace`, `required`.

`@XmlSchema` *w* `package-info.java`

```
1 @javax.xml.bind.annotation.XmlSchema(
2     namespace = "http://www.example.at/hospital",
3     elementFormDefault = javax.xml.bind.annotation.XmlNsForm.QUALIFIED)
4 package hospital;
```

`@XmlElementWrapper` — *opakowanie listy*

```
1 @XmlElementWrapper(name="emails")
2 @XmlElement(name="email")
3 List<String> emails = new ArrayList<String>();
```

Generuje:

```
1 <emails>
2     <email>adam.nowak@a.pl</email>
3     <email>jarek.kowalski@b.pl</email>
4 </emails>
```

9.2.7. Dziedziczenie klas i `@XmlSeeAlso`

JAXB obsługuje hierarchie klas, ale wymaga jawnego poinformowania kontekstu o podtypach używanych polimorficznie. Jeśli metoda serwisu deklaruje typ bazowy jako parametr lub wartość zwracaną, a w czasie działania przekazywany jest obiekt podtypu, JAXB nie może go serializować bez znajomości podtypu w czasie tworzenia `JAXBContext`.

Rozwiązaniem jest adnotacja `@XmlSeeAlso` na klasie bazowej:

```
@XmlRootElement
@XmlSeeAlso({Cat.class, Dog.class})      // poinformuj JAXB o podtypach
public abstract class Animal {
    public String name;
    public Animal() {}
}

@XmlRootElement
public class Cat extends Animal {
    public String indoor;
    public Cat() {}
}

@XmlRootElement
public class Dog extends Animal {
    public String breed;
    public Dog() {}
}
```

Bez `@XmlSeeAlso` marshalling obiektu `Dog` przez `Marshaller` skonfigurowanego tylko dla klasy `Animal` zakończyłby się wyjątkiem `JAXBException`.

`wsimport` automatycznie generuje `@XmlSeeAlso` w klasie bazowej SEI, gdy schemat WSDL używa dziedziczenia przez `xs:extension`.

Adnotacja `@XmlDiscriminatorValue` (dostępna w implementacji MOXy) pozwala na kontrolę znacznika rozróżniającego podtypy w JSON — przydatna gdy serwis używa zarówno XML jak i JSON.

9.2.7.1. Typy anonimowe a dziedziczenie

Przy użyciu strategii `document/literal wrapped` JAX-WS opakowuje parametry w anonimowe elementy XMLSchema. Gdy parametr jest typem bazowym hierarchii, element wygenerowany przez `wsgen` będzie zawierał adnotację `@XmlSeeAlso` wskazującą na znane podtypy — programista musi zadbać, by były one widoczne w tym samym archiwum co SEI.

9.3. JAXB a JAX-WS

JAXB jest silnikiem wiązania wbudowanym w JAX-WS. Typy XMLSchema zdefiniowane

w WSDL są kompilowane przez `wsimport` (który wewnętrznie wywołuje `xjc`) do klas Javy z adnotacjami JAXB. Podczas każdego wywołania operacji serwisu JAX-WS wykonywane są trzy kroki:

1. `Unmarshaller` deserializuje ciało wiadomości SOAP na obiekty Javy — parametry metody,
2. implementacja metody serwisu jest wywoływana z tymi obiektami,
3. `Marshaller` serializuje wartość zwracaną z powrotem do XML w ciele odpowiedzi SOAP.

Programista może dostosować kształt generowanego XML adnotacjami JAXB bezpośrednio na klasach domenowych — bez zmiany WSDL. Adnotacja `@XmlJavaTypeAdapter` jest szczególnie użyteczna dla typów, których JAXB nie obsługuje domyślnie (np. `java.util.Date`, `java.util.Map`, typy z bibliotek zewnętrznych).

Kolejny rozdział opisuje standardy WS-Addressing i MTOM — rozszerzenia stosu SOAP dotyczące adresowania wiadomości i wydajnego przesyłania danych binarnych.

10. Standardy WS — Polityki, Adresowanie i Załączniki Binarne

Stos SOAP definiowany przez SOAP i WSDL jest wystarczający do tworzenia prostych serwisów. Praktyczne wdrożenia wymagają jednak dodatkowych możliwości: deklaratywnej konfiguracji wymagań (WS-Policy), adresowania niezależnego od protokołu transportowego (WS-Addressing) oraz wydajnego przesyłania danych binarnych (MTOM/XOP).

10.1. WS-Policy — Web Services Policy Framework

Specyfikacje WSDL pozwalają na rozszerzenia w wielu miejscach dokumentu. Oprócz wiązań protokołów można deklarować dodatkowe wymagania serwisu — zabezpieczenia, adresowanie czy wymagania jakości usług (QoS). Sposób zapisu takich deklaracji określa rekomendacja W3C *Web Services Policy Framework*.

WS-Policy:

- jest podstawą opisu dodatkowych funkcjonalności w WSDL,
- pozwala na definiowanie, grupowanie i tworzenie alternatyw wybranych opcji,
- takie deklaracje nazywane są **politykami**, zawierają **asercje** (wymagania),
- stosowane tam, gdzie dozwolone są elementy rozszerzające WSDL.

Przykład polityki wskazującej, że serwis obsługuje szyfrowanie symetryczne lub asymetryczne:

```
<wsp:Policy
  xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
  xmlns:wsp="http://www.w3.org/ns/ws-policy">
  <wsp:ExactlyOne>
    <sp:Basic256Rsa15/>
    <sp:TripleDesRsa15/>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Elementy z przestrzeni `http://www.w3.org/ns/ws-policy` (wersja 1.5) zawsze towarzyszą elementom z innych specyfikacji (tu WS-SecurityPolicy) określającym konkretne funkcjonalności.

10.1.1. Łączenie asercji

`wsp:ExactlyOne` **(*policy alternatives*)**

> Lista wzajemnie wykluczających się asercji — klient musi wybrać i używać tylko jednej (np. metody szyfrowania).

`wsp:All`
: Lista asercji wymaganych jednocześnie (np. szyfrowanie wiadomości i WS-Addressing).

Oba elementy można zagnieżdżać:

```
<wsp:Policy
  xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
  xmlns:wsp="http://www.w3.org/ns/ws-policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:Basic256Rsa15/>
    </wsp:All>
    <wsp:All>
      <sp:TripleDesRsa15/>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Powyższa definicja jest równoważna prostszej z jednym `ExactlyOne`. Asercje mogą być konfigurowane przez zagnieżdżone polityki:

```
<wsp:Policy
  xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
  xmlns:wsp="http://www.w3.org/ns/ws-policy" >
  <sp:TransportBinding>
    <wsp:Policy>
      <sp:TransportToken>
        <wsp:Policy>
          <sp:HttpsToken RequireClientCertificate="false" />
        </wsp:Policy>
      </sp:TransportToken>
      <sp:AlgorithmSuite>
        <wsp:Policy>
          <wsp:ExactlyOne>
            <sp:Basic256Rsa15 />
            <sp:TripleDesRsa15 />
          </wsp:ExactlyOne>
        </wsp:Policy>
      </sp:AlgorithmSuite>
    </wsp:Policy>
  </sp:TransportBinding>
</wsp:Policy>
```

Szyfrowanie zdefiniowane jest tu na poziomie protokołu transportowego. Przy wyborze funkcjonalności można wybierać ze wszystkich asercji `ExactlyOne` we wszystkich zagnieżdżonych politykach.

10.1.2. Referencje do polityk

Raz zdefiniowana polityka może być wielokrotnie użyta przez nadanie jej identyfikatora atrybutem `wsu:Id`:

```
<wsp:Policy
  xmlns:wsp="http://www.w3.org/ns/ws-policy"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
  wsu:Id="MyPolicy" >
  ....
</wsp:Policy>
```

Referencja do oznaczonej polityki:

```
<wsp:PolicyReference URI="#MyPolicy"/>
```

Asercja opcjonalna (klient nie musi z niej korzystać) — atrybut `wsp:Optional="true"`:

```
<wsp:Policy
  xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
  xmlns:wsp="http://www.w3.org/ns/ws-policy">
  <sp:Basic256Rsa15 wsp:Optional="true"/>
</wsp:Policy>
```

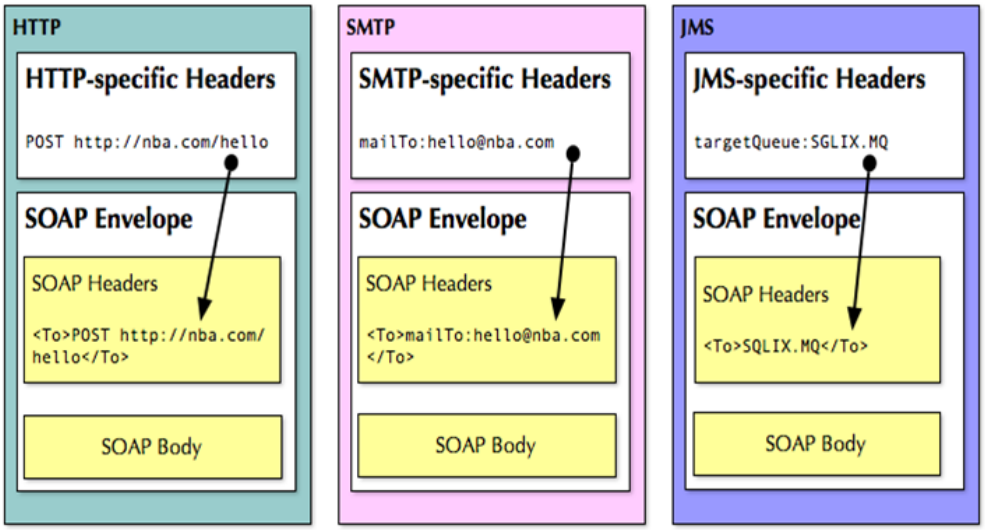
Powyższa polityka jest równoważna rozpisanej formie z `ExactlyOne` i pustym `All` (brak asercji = brak wymagania, czyli opcjonalność):

```
<wsp:Policy
  xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
  xmlns:wsp="http://www.w3.org/ns/ws-policy" >
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:Basic256Rsa15/>
    </wsp:All>
    <wsp:All />
  </wsp:ExactlyOne>
</wsp:Policy>
```

10.2. WS-Addressing

Specyfikacja pozwala w sposób abstrakcyjny opisać nadawców i adresatów wiadomości:

- adresowanie jest niezależne od protokołu transportowego,
- wiadomość może przechodzić przez węzły pośredniczące przed dostarczeniem do właściwego adresata,
- umożliwia komunikację prawdziwie asynchroniczną,
- odpowiedź lub błędy mogą być kierowane do innego odbiorcy niż nadawca,
- EPR (*endpoint reference*) może zawierać dodatkowe parametry referencyjne przekazywane z każdą wiadomością.



Rysunek 22. WS-Addressing

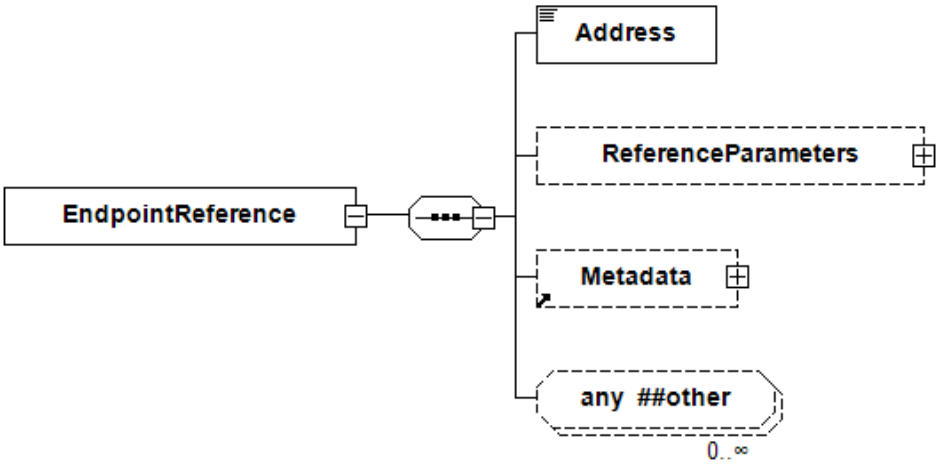
WS-Addressing wprowadza dwie konstrukcje:

- nagłówki adresowe (*addressing headers*),
- *endpoint reference* (EPR).

Elementy specyfikacji zdefiniowane są w przestrzeni nazw `http://www.w3.org/2005/08/addressing`.

10.2.1. Endpoint Reference

Endpoint reference (EPR) umożliwia dokładniejsze adresowanie usług — nie tylko przez fizyczny adres, ale również przez konkretne parametry.



Rysunek 23. Endpoint Reference

Elementy EPR:

Address

URI identyfikujące usługę — adres logiczny lub fizyczny.

ReferenceParameters

Zestaw elementów określanych przez twórcę serwisu.

Metadata

Opis serwisu — może zawierać definicję WSDL, `InterfaceName` (nazwa interfejsu/portType), `ServiceName`.

Specjalne wartości adresów:

- `http://www.w3.org/2005/08/addressing/anonymous` — nie można określić nadawcy lub odbiorcy,
- `http://www.w3.org/2005/08/addressing/none` — używany w `ReplyTo`/`FaultTo`: odpowiedź nie jest oczekiwana.

```
<wsa:EndpointReference
      xmlns:wsa="http://www.w3.org/2005/08/addressing"
      xmlns:wsaw="http://www.w3.org/2006/02/addressing/wsdl"
      xmlns:fabrikam="http://example.com/fabrikam">
    <wsa:Address>http://example.com/fabrikam/acct</wsa:Address>
    <wsa:ReferenceParameters>
        <fabrikam:CustomerKey>123456789</fabrikam:CustomerKey>
        <fabrikam:ShoppingCart>ABCDEFG</fabrikam:ShoppingCart>
    </wsa:ReferenceParameters>
    <wsa:Metadata>
        <wsaw:InterfaceName>fabrikam:Inventory</wsaw:InterfaceName>
    </wsa:Metadata>
</wsa:EndpointReference>
```

10.2.2. Elementy adresowe nagłówka

Element	Wymagany	Opis
`To`	TAK	URI adresata wiadomości
`From`	NIE	EPR nadawcy wiadomości
`ReplyTo`	NIE	EPR adresata odpowiedzi (domyślnie *anonymous*)
`FaultTo`	NIE	EPR adresata błędu, jeśli inny niż `ReplyTo`
`Action`	TAK	URI identyfikujące akcję wiadomości (≈ `SOAPAction`)
`MessageID`	NIE	URI jednoznacznie identyfikujące wiadomość
`RelatesTo`	NIE	`MessageID` powiązanych wiadomości

Element	Wymagany	Opis
`ReferenceParameters`	NIE	Parametry ustalone przez partnerów

```
<SOAP-ENV:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
    <wsa:To>
        http://localhost:6080/myAppServer/services/myService
    </wsa:To>
    <wsa:Action>myOperation1</wsa:Action>
    <wsa:MessageID>uuid:515704D6-0111-4000-E000-82267F000001</wsa:MessageID>
    <wsa:ReplyTo>
        <wsa:Address>http://www.w3.org/2005/08/addressing/anonymous</wsa:Address>
    </wsa:ReplyTo>
    <wsa:FaultTo>
        <wsa:Address>http://www.w3.org/2005/08/addressing/none</wsa:Address>
    </wsa:FaultTo>
</SOAP-ENV:Header>
```

W celu zaznaczenia, że serwis używa WS-Addressing należy w sekcji `binding` dokumentu WSDL dodać element `UsingAddressing` z przestrzeni `http://www.w3.org/2006/05/addressing/wsdl`. Dodatkowo można wymienić akcje wiadomości za pomocą atrybutu `Action`:

```
<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType"
         xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl">
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http" />
    <wsaw:UsingAddressing wsdl:required="true" />
    <operation name="GetLastTradePrice">
        <soap:operation soapAction="http://example.com/GetLastTradePrice" />
        <input  wsaw:Action="http://example.com/GetQuote">
            <soap:body use="literal" />
        </input>
        <output wsaw:Action="http://example.com/Quote">
            <soap:body use="literal" />
        </output>
    </operation>
</binding>
```

Alternatywą jest zdefiniowanie asercji w elemencie `wsp:Policy` w elemencie `port` — daje większą kontrolę (np. wymóg odpowiedzi nie-anonimowych):

```
<wsdl:service name="StockQuoteSoapService"
    xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
  <wsdl:port binding="tns:StockQuoteSoapBinding"
      name="StockQuoteSoapPort">
    <soap:address
        location="http://localhost:9000/StockQuoteSoapService"/>
    <wsp:Policy xmlns:wsp="http://www.w3.org/ns/ws-policy">
      <wsam:Addressing>
        <wsp:Policy>
          <wsam:NonAnonymousResponses/>
        </wsp:Policy>
      </wsam:Addressing>
```

```
    </wsp:Policy>
  </wsdl:port>
</wsdl:service>
```

10.2.3. WS-Addressing w JAX-WS

JAX-WS 2.1 zintegrował obsługę WS-Addressing za pomocą adnotacji `@Addressing` z pakietu `javax.xml.ws.soap`:

```
@WebService
@Addressing(enabled = true, required = true)
public class OrderService {
    // ...
}
```

Parametr `required = true` powoduje, że kontener odrzuci żądania bez nagłówków WS-Addressing. Po stronie klienta nagłówki WS-Addressing dodawane są automatycznie przez środowisko wykonawcze (JAX-WS RI / Metro).

Klasa `W3CEndpointReference` (implementacja interfejsu `EndpointReference`) reprezentuje EPR w API Javy. Klient może uzyskać EPR z istniejącego portu i przekazać go dalej:

```
Service service = Service.create(wsdlURL, serviceName);
OrderServicePortType port = service.getPort(portName,
    OrderServicePortType.class,
    new AddressingFeature(true, true)); // enabled=true, required=true

W3CEndpointReferenceBuilder builder = new W3CEndpointReferenceBuilder();
W3CEndpointReference epr = builder
    .address("http://example.com/orders")
    .build();
```

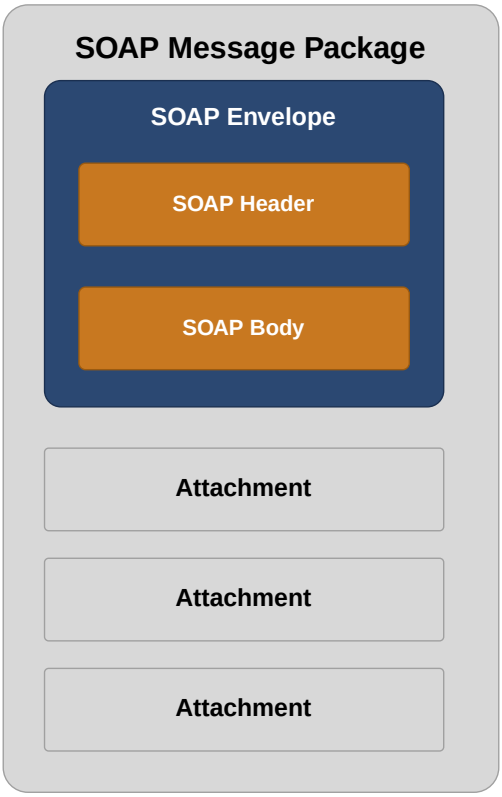
10.3. Przesyłanie załączników binarnych

Dane binarne można zagnieżdżać w XML poprzez kodowanie *base64*, ale zwiększa to rozmiar o ~33%. W Javie obiekty `String` używają UTF-16, więc rozmiar pamięci się podwaja.

Rozwiązaniem jest przesyłanie danych binarnych w postaci załączników. Trzy technologie:

- **SOAP with Attachments (SwA)** — W3C,
- **Attachments Profile (WS-I)**,
- **MTOM/XOP** — W3C (zalecane).

10.3.1. SOAP with Attachments



Rysunek 24. SOAP z załącznikami

Notatka W3C określa schemat *SOAP message package*:

- cała wiadomość w `Multipart/Related`,
- głównym elementem jest koperta SOAP,
- wiadomość posiada odnośniki do załączników,
- odnośniki `CID` wskazują na `Content-ID` części kontenera.

```
MIME-Version: 1.0
Content-Type: Multipart/Related; boundary=MIME_boundary;
              type=text/xml; start="<claim.xml@claiming-it.com>"

--MIME_boundary
Content-Type: text/xml; charset=UTF-8
Content-ID: <claim.xml@claiming-it.com>

<?xml version='1.0' ?>
<SOAP-ENV:Envelope ...>
    <SOAP-ENV:Body>
        <theSignedForm href="cid:claim.tiff@claiming-it.com"/>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

--MIME_boundary
Content-Type: image/tiff
Content-ID: <claim.tiff@claiming-it.com>
...binary TIFF image...
```

```
20 --MIME_boundary--
```

Przykładowy nagłówek HTTP żądania z paczką SOAP z załącznikiem:

```
1 POST /insuranceClaims HTTP/1.1
2 Host: www.risky-stuff.com
3 Content-Type: Multipart/Related; boundary=MIME_boundary;
4     type=text/xml; start="<claim061400a.xml@claiming-it.com>"
5 SOAPAction: http://schemas.risky-stuff.com/Auto-Claim
```

10.3.2. WS-I Attachments Profile

Wprowadza typ `swaRef` (z `http://ws-i.org/profiles/basic/1.1/xsd`) — jednoznacznie wskazuje, że ciało elementu jest transportowane w załączniku:

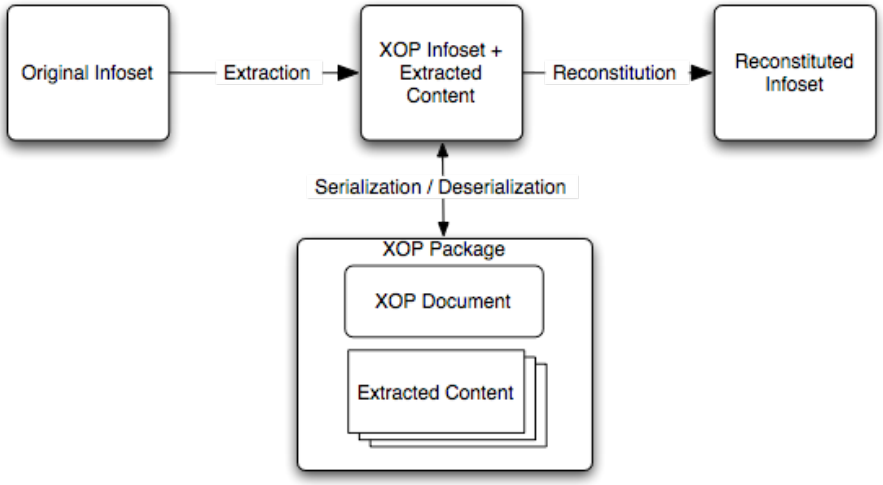
```
1 <xs:element name="person"
2             xmlns:swaRef="http://ws-i.org/profiles/basic/1.1/xsd">
3     <xs:complexType>
4         <xs:sequence>
5             <xs:element name="name"    type="xs:string"/>
6             <xs:element name="surname" type="xs:string"/>
7             <xs:element name="photo"   type="swaRef:swaRef"/>
8         </xs:sequence>
9     </xs:complexType>
10 </xs:element>
```

Instancja:

```
1 <person>
2     <name>John</name>
3     <surname>Doe</surname>
4     <photo>cid:b0a597fd5ef71d25@example.com</photo>
5 </person>
```

10.3.3. MTOM/XOP

XML-binary Optimized Packaging (XOP) — rekomendacja W3C. Dokument XML z danymi zakodowanymi w *base64* jest transformowany do paczki zawierającej dane binarne i oryginalny XML z odnośnikami (`xop:Include`).



Rysunek 25. XOP — transformacja danych binarnych

SOAP Message Transmission Optimization Mechanism (MTOM) zaleca użycie XOP jako mechanizmu serializacji wiadomości SOAP z załącznikami — następca SwA.

Przed XOP:

```
<m:data xmlns:m='http://example.org/stuff'>
    <m:photo>/aWKKapGGyQ=</m:photo>
</m:data>
```

Po XOP — dane zastąpione przez `xop:Include`:

```
<m:data xmlns:m='http://example.org/stuff'>
    <m:photo>
        <xop:Include xmlns:xop='http://www.w3.org/2004/08/xop/include'
                     href='cid:http://example.org/me.png'/>
    </m:photo>
</m:data>
```

Asercja MTOM/XOP w WS-Policy:

```
<Policy xmlns="http://www.w3.org/ns/ws-policy"
        xmlns:wsoma="http://schemas.xmlsoap.org/ws/2004/09/policy/
                     optimizedmimeserialization">
    <wsoma:OptimizedMimeSerialization/>
</Policy>
```

10.3.4. Porównanie mechanizmów załączników

Cecha	SwA (SOAP with Attachments)	WS-I Attachments Profile (swaRef)	MTOM/XOP
Organizacja standardów	W3C Note (2000)	WS-I Basic Profile 1.1 (2006)	W3C Recommendation (2005)
Referencja do załącznika	`href="cid:…"` jako zwykły string	Typ `swaRef` w schemacie	`xop:Include` wewnątrz elementu `base64Binary`
Integracja z XMLSchema	Brak (string)	Częściowa (typ `swaRef`)	Pełna (element zachowuje typ `xs:base64Binary`)
Walidacja schematu wiadomości	Niemożliwa	Utrudniona	Możliwa
Wsparcie JAX-WS	Ograniczone (brak natywnych adnotacji)	`@XmlAttachmentRef` + `DataHandler`	Pełne: `@MTOM`, `MTOMFeature`, `DataHandler`
Zgodność z WS-I Basic Profile	Tylko BP 1.2	BP 1.1 (typ `swaRef`)	MTOM Binding Profile 1.0
Obsługa w WSDL	Wiązanie MIME	Wiązanie MIME + typ `swaRef`	Polityka MTOM w WS-Policy
Wpływ na rozmiar wiadomości	Bez base64 (dane binarne osobno)	Bez base64	Bez base64; próg `threshold` steruje optymalizacją
Zalecenie	❌ Nie stosować w nowych serwisach	⚠️ Tylko gdy wymagana zgodność BP 1.1 bez MTOM	✅ Zalecany dla wszystkich nowych serwisów

Podsumowanie reguły: jeśli serwis wymaga przesyłania danych binarnych, należy wybrać **MTOM/XOP** — jest jedynym mechanizmem w pełni zintegrowanym z XMLSchema, JAX-WS i WS-Policy. SwA i swaRef zachowują znaczenie jedynie przy integracji z systemami legacy, które MTOM nie obsługują.

10.3.5. MTOM w JAX-WS

JAX-WS obsługuje MTOM przez `@BindingType` na klasie serwisu lub przez `MTOMFeature` przy tworzeniu portu po stronie klienta:

```
// Serwis — włączenie MTOM przez typ wiązania
@WebService
@BindingType(SOAPBinding.SOAP11HTTP_MTOM_BINDING)
public class DocumentService {
```

```
5
6     public void upload(
7             @XmlMimeType("application/octet-stream") DataHandler data) {
8         // DataHandler udostępnia InputStream z przesłanymi danymi
9     }
10 }
11
12 // Klient — MTOMFeature z progiem (threshold) 1024 bajtów
13 DocumentServicePortType port = service.getPort(
14     DocumentServicePortType.class,
15     new MTOMFeature(true, 1024));
```

Dane binarne w parametrach metody przekazywane są przez typ `javax.activation.DataHandler` lub `byte[]`. Adnotacja `@XmlMimeType` (z pakietu `javax.xml.bind.annotation`) podpowiada JAXB, jaki typ MIME przypisać elementowi XMLSchema (`base64Binary` z atrybutem `expectedContentTypes`).

Parametr `threshold` w `MTOMFeature` określa minimalny rozmiar danych (w bajtach), poniżej którego MTOM nie jest stosowany — małe wartości pozostają zakodowane w *base64* wewnątrz koperty SOAP.

10.4. WS-Addressing i MTOM w praktyce

WS-Addressing i MTOM/XOP rozszerzają podstawowy stos SOAP o możliwości wymagane w zaawansowanych scenariuszach integracyjnych:

WS-Addressing

Przenosi informacje o routingu wiadomości poza protokół transportowy — niezbędne dla wywołań asynchronicznych (*one-way*) i przekierowania odpowiedzi do innego endpointu. W JAX-WS włączane adnotacją `@Addressing(enabled=true, required=true)`; EPR reprezentuje klasa `W3CEndpointReference`.

MTOM/XOP

Eliminuje narzut kodowania *base64* (~33% wzrostu rozmiaru) przy przesyłaniu danych binarnych (obrazy, pliki PDF, certyfikaty). W JAX-WS włączane przez `@BindingType(SOAPBinding.SOAP11HTTP_MTOM_BINDING)` lub `MTOMFeature`; dane binarne przekazywane przez `DataHandler`.

Oba rozszerzenia obsługiwane są przez stos WSIT (Metro) lub Apache CXF — mechanizmy ich konfiguracji opisuje rozdział WSIT. Następny rozdział opisuje SAAJ (*SOAP with Attachments API for Java*) — niskopoziomowe API umożliwiające bezpośrednią konstrukcję i parsowanie wiadomości SOAP z poziomu kodu Java.

11. SOAP with Attachments API for Java — SAAJ

SOAP with Attachments API for Java (SAAJ) jest niskopoziomowym API do bezpośredniej pracy z wiadomościami SOAP w wersji 1.1 i 1.2 — bez pośrednictwa frameworku JAX-WS. API umożliwia programistyczną konstrukcję kopert SOAP, modyfikację nagłówków i załączników oraz wysyłanie wiadomości przez HTTP. SAAJ tworzy wiadomości zgodne z *WS-I Basic Profile 1.0*.

SAAJ pojawia się w projektach JAX-WS w dwóch kontekstach:

- **handlery protokołu SOAP** — interfejs `SOAPHandler` w łańcuchu handlerów JAX-WS operuje na `SOAPMessageContext`, który udostępnia obiekt `SOAPMessage`; znajomość SAAJ jest niezbędna do pisania handlerów na poziomie protokołu,
- **samodzielny klient SOAP** — gdy serwis nie udostępnia poprawnego WSDL lub wymaga niestandardowej konstrukcji wiadomości.

11.1. Wprowadzenie

API umieszczone w pakiecie `javax.xml.soap`. Oparte o DOM API. Większość klas reprezentuje konkretne elementy koperty: `SOAPEnvelope`, `SOAPBody`, `SOAPHeader`, `SOAPFault`. Klasy te rozszerzają interfejsy DOM (`org.w3c.dom.Document`, `Node`, `Element`).

Klasa / Interfejs	Opis
`SOAPMessage`	Kontener całej wiadomości — `SOAPPart` plus ewentualne `AttachmentPart`
`SOAPPart`	Implementuje `org.w3c.dom.Document`; zawiera kopertę SOAP
`SOAPEnvelope`	Korzeń XML wiadomości; zawiera `SOAPHeader` i `SOAPBody`
`SOAPHeader`	Opcjonalny nagłówek; zawiera obiekty `SOAPHeaderElement`
`SOAPBody`	Obowiązkowe ciało; zawiera `SOAPBodyElement` lub `SOAPFault`
`SOAPFault`	Standardowy element błędu SOAP zgodny ze specyfikacją
`AttachmentPart`	Dane binarne lub tekstowe poza częścią SOAP
`MessageFactory`	Fabryka `SOAPMessage`; tryby: `SOAP_1_1_PROTOCOL` / `SOAP_1_2_PROTOCOL`
`SOAPFactory`	Fabryka elementów XML — użyteczna bez dostępu do instancji koperty
`SOAPConnection`	Uproszczony transport HTTP point-to-point (żądanie-odpowiedź)

11.2. Kreowanie połączenia

```
SOAPConnectionFactory factory = SOAPConnectionFactory.newInstance();
SOAPConnection connection = factory.createConnection();

SOAPMessage request = // ... tworzenie wiadomości SOAP

java.net.URL endpoint = new URL("http://example.com/someendpoint");
SOAPMessage response = connection.call(request, endpoint);
```

11.3. Tworzenie wiadomości

```
MessageFactory factory = MessageFactory.newInstance();
SOAPMessage message = factory.createMessage();
```

Nowa wiadomość zawiera: `SOAPPart` → `SOAPEnvelope` → `SOAPHeader` + `SOAPBody`.

Dostęp do elementów:

```
// przez hierarchię
SOAPPart soapPart    = message.getSOAPPart();
SOAPEnvelope envelope = soapPart.getEnvelope();
SOAPHeader header     = envelope.getHeader();
SOAPBody body         = envelope.getBody();

// lub bezpośrednio
SOAPHeader header = message.getSOAPHeader();
SOAPBody body     = message.getSOAPBody();
```

Usunięcie elementu: `header.detachNode();`

Dodawanie elementów do `SOAPBody`:

```
Name bodyName = envelope.createName("GetLastTradePrice", "m", "http://example.com");
SOAPBodyElement bodyElement = body.addBodyElement(bodyName);
// lub
SOAPFault soapFault = body.addFault();
```

Nie zawsze mamy dostęp do instancji koperty — można użyć fabryki `SOAPFactory`:

```
SOAPFactory soapFactory = SOAPFactory.newInstance();
Name bodyName = soapFactory.createName("GetLastTradePrice", "m", "http://example.com");
SOAPBodyElement bodyElement = body.addBodyElement(bodyName);
```

Kompletny przykład tworzenia wiadomości zamówienia:

```
MessageFactory factory = MessageFactory.newInstance();
SOAPMessage message = factory.createMessage();
SOAPBody body = message.getSOAPBody();

Name bodyName = soapFactory.createName("PurchaseLineItems", "PO",
                                       "http://sonata.fruitsgalore.com");
SOAPBodyElement purchaseLineItems = body.addBodyElement(bodyName);

Name childName = soapFactory.createName("Order");
SOAPElement order = purchaseLineItems.addChildElement(childName);

childName = soapFactory.createName("Product");
SOAPElement product = order.addChildElement(childName);
product.addTextNode("Apple");

childName = soapFactory.createName("Price");
SOAPElement price = order.addChildElement(childName);
price.addTextNode("1.56");

message.writeTo(System.out);
```

Dodawanie nagłówków (np. zgodność z WS-I BP 1.0):

```
SOAPHeader header = message.getSOAPHeader();
Name headerName = soapFactory.createName("Claim", "wsi",
    "http://ws-i.org/schemas/conformanceClaim/");
SOAPHeaderElement headerElement = header.addHeaderElement(headerName);
headerElement.addAttribute(soapFactory.createName("conformsTo"),
    "http://ws-i.org/profiles/basic1.0/");
```

11.4. Dodawanie załączników

```
AttachmentPart attachment = message.createAttachmentPart();
String stringContent = "Mój załącznikowy tekst";
attachment.setContent(stringContent, "text/plain");
attachment.setContentId("zalacznik_1");
message.addAttachmentPart(attachment);
```

Jako treść załącznika można użyć: `String`, `javax.xml.transform.Source`, `javax.activation.DataHandler` (automatycznie ustawia `Content-Type`):

```
URL url = new URL("http://example.com/img.jpg");
DataHandler dataHandler = new DataHandler(url);
AttachmentPart attachment = message.createAttachmentPart(dataHandler);
attachment.setContentId("attached_image");
message.addAttachmentPart(attachment);
```

Dostęp do załączników:

```
java.util.Iterator iterator = message.getAttachments();
```

```
while (iterator.hasNext()) {
    AttachmentPart attachment = (AttachmentPart) iterator.next();
    String id   = attachment.getContentId();
    String type = attachment.getContentType();
    if ("text/plain".equals(type)) {
        Object content = attachment.getContent();
        System.out.println("Attachment " + id + ": " + content);
    }
}
```

11.5. Podsumowanie

Zalety SAAJ

- Bezpośrednia obsługa komunikatów SOAP na poziomie protokołu,
- elastyczność i kontrola nad szczegółami,
- możliwość stworzenia klienta nawet gdy serwis zachowuje się niestandardowo lub brak poprawnego WSDL.

Wady SAAJ

- programowanie jest pracochłonne — wymaga znajomości struktury koperty SOAP,
- programista musi ręcznie zapewnić zgodność z WSDL,
- wiadomość wczytywana jest w całości do pamięci; dla dużych wiadomości należy stosować rozwiązania strumieniowe (SAX lub StAX).

SAAJ i JAX-WS najczęściej współpracują przez interfejs `SOAPHandler` — handler na poziomie protokołu odczytuje i modyfikuje kopertę SOAP za pomocą SAAJ, po czym zwraca `true`, aby pozwolić przetwarzaniu kontynuować łańcuch.

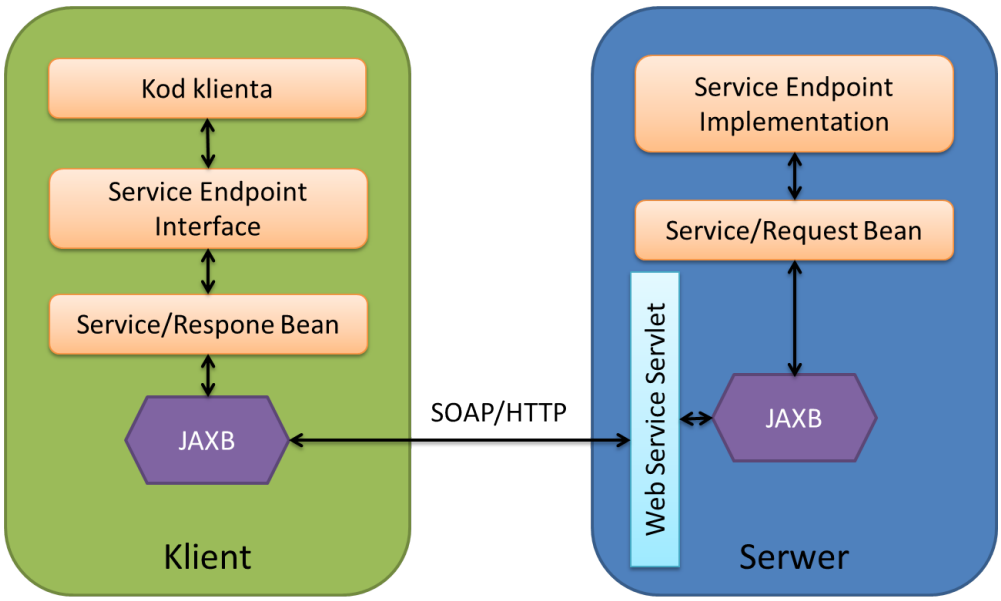
Kolejny rozdział opisuje JAX-WS — główny framework Java EE 6 do tworzenia serwisów SOAP, który integruje SOAP, WSDL, JAXB i SAAJ w spójne środowisko programowania.

12. Java API for XML Web Services — JAX-WS

JAX-WS jest główną specyfikacją Java EE 6 do tworzenia i konsumowania serwisów SOAP — stanowi centrum egzaminu 1Z0-897. API scala wszystkie omówione wcześniej technologie: SOAP i WSDL (protokół i kontrakt), JAXB (wiązanie typów), SAAJ (dostęp przez handlery), WS-Addressing i MTOM (rozszerzenia jakości usług). Wersja 2.2 zastępuje wcześniejszą specyfikację JAX-RPC i wchodzi w skład Java SE oraz Java EE 6. Główne cechy:

- używa JAXB jako mechanizmu wiązania dokumentów XML z Javą,
- wspiera SOAP w wersji 1.1 i 1.2,
- jest zgodne z WS-I Basic Profile 1.1,
- wspiera synchroniczną i asynchroniczną komunikację po stronie klienta,
- jest częścią Java SE,
- pozwala na użycie innego niż HTTP protokołu transportowego,
- oparte głównie na adnotacjach,
- wspiera sesje protokołów HTTP i SOAP,
- definiuje mechanizm *handlerów* uruchamianych przed i po każdej operacji,
- wspiera strategie Java-To-WSDL i WSDL-To-Java,
- wspiera style *RPC/literal* i *document/literal*.

12.1. Architektura JAX-WS



JAX-WS oparty jest o trzy zasadnicze komponenty:

***Service Endpoint Interface* (SEI)**

Interfejs dostępny zarówno po stronie serwera jak i klienta. Stanowi kontrakt pomiędzy klientem a serwerem i jest odbiciem typu portu w WSDL na komponent

Javy. Dzięki SEI, które jest zaimplementowane przez proxy, klient może wywoływać metody serwisu tak, jakby używał lokalnego obiektu.

***Service Endpoint Bean* (SIB)**

Implementacja SEI po stronie serwera. Metody tego obiektu wywoływane są przez JAX-WS jako rezultat żądań przesyłanych przez klienta.

Proxy Instance

Implementacja SEI po stronie klienta. Proxy stanowi namiastkę (*stub*) serwisu po stronie klienta — wywołania metod są transparentnie przekształcane w żądania SOAP i przesyłane do serwera. Może być stworzone statycznie (jeśli SEI jest znane w czasie tworzenia klienta) lub dynamicznie w trakcie wykonywania programu.

12.2. Strategia WSDL-To-Java

Narzędzia JAX-WS pozwalają na utworzenie SEI z udostępnionego WSDL za pomocą `wsimport`:

```
1 wsimport -s src -keep stockquote.wsdl
```

Opcja `-keep` zachowuje kod źródłowy, a `-s` wskazuje katalog dla kodu źródłowego. Przykładowe wygenerowane klasy (dla `StockQuote`):

- `ObjectFactory.java`
- `package-info.java`
- `StockQuotePortType.java` — SEI
- `StockQuoteService.java` — rozszerza `javax.xml.ws.Service`
- `TradePrice.java`, `TradePriceRequest.java` — klasy JAXB

Wygenerowana klasa serwisu:

```
1  @WebServiceClient(name = "StockQuoteService",
2      targetNamespace = "http://example.com/stockquote.wsdl",
3      wsdlLocation = "file:/E:/tst/stockquote.wsdl")
4  public class StockQuoteService extends Service {
5
6      public StockQuoteService() { ... }
7      public StockQuoteService(URL wsdlLocation) { ... }
8      public StockQuoteService(URL wsdlLocation, QName serviceName) { ... }
9      // + warianty z WebServiceFeature...
10
11     @WebEndpoint(name = "StockQuotePort")
12     public StockQuotePortType getStockQuotePort() {
13         return super.getPort(
14             new QName("http://example.com/stockquote.wsdl", "StockQuotePort"),
15             StockQuotePortType.class);
16     }
```

```
17 }
```

Wygenerowany SEI:

```
1 @WebService(name = "StockQuotePortType",
2             targetNamespace = "http://example.com/stockquote.wsdl")
3 @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
4 @XmlSeeAlso({ ObjectFactory.class })
5 public interface StockQuotePortType {
6
7     @WebMethod(operationName = "GetLastTradePrice",
8                action = "http://example.com/GetLastTradePrice")
9     @WebResult(name = "TradePrice",
10                targetNamespace = "http://example.com/stockquote.xsd",
11                partName = "body")
12    public TradePrice getLastTradePrice(
13        @WebParam(name = "TradePriceRequest",
14                  targetNamespace = "http://example.com/stockquote.xsd",
15                  partName = "body")
16        TradePriceRequest body);
17 }
```

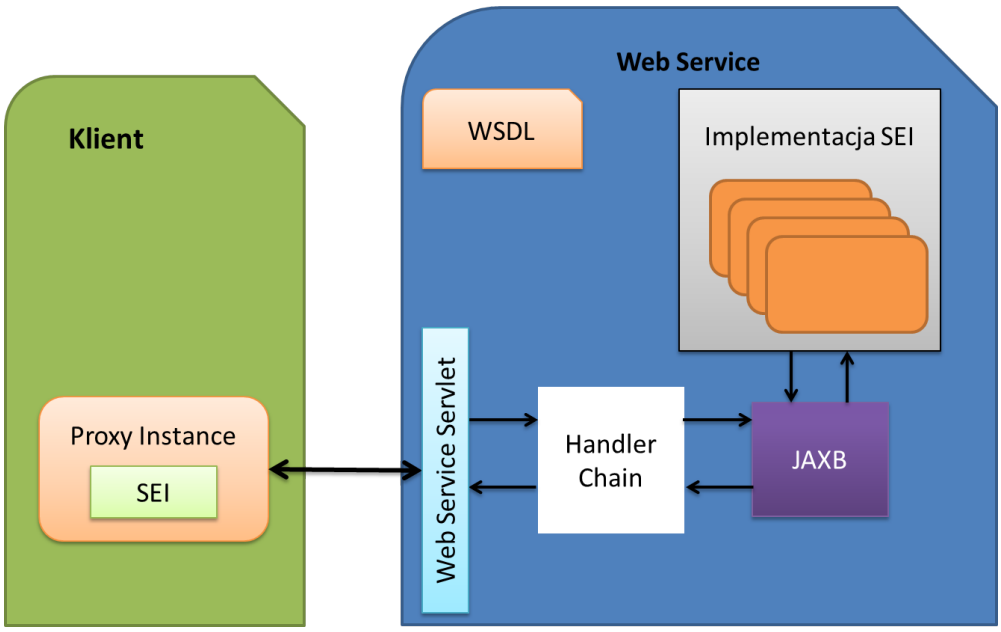
Użycie klienta:

```
1 StockQuoteService service = new StockQuoteService();
2 StockQuotePortType port = service.getStockQuotePort();
3
4 TradePriceRequest request = new TradePriceRequest();
5 request.setTickerSymbol("USDPLN");
6
7 TradePrice tradePrice = port.getLastTradePrice(request);
8 System.out.println("Kwotowanie USDPLN: " + tradePrice.getPrice());
```

Implementacja serwisu:

```
1 @WebService(endpointInterface = "com.example.stockquote.StockQuotePortType")
2 public class StockQuoteServiceImpl {
3     public TradePrice getLastTradePrice(TradePriceRequest priceRequest) {
4         TradePrice result = new TradePrice();
5         float factor = (float)(1.0 + Math.random() / 10.0);
6         if ("USDPLN".equals(priceRequest.getTickerSymbol()))
7             result.setPrice(3.2f * factor);
8         // ...
9         return result;
10    }
11 }
```

Implementacja musi posiadać adnotację `@WebService` z atrybutem `endpointInterface`.



Rysunek 26. Serwer JAX-WS

12.3. Strategia Java-To-WSDL (code-first)

W podejściu *code-first* programista tworzy klasę Javy z adnotacjami JAX-WS, a narzędzie `wsgen` generuje artefakty wymagane przez środowisko wykonawcze JAX-WS.

```
1 @WebService
2 @SOAPBinding(style = SOAPBinding.Style.DOCUMENT)
3 public class HelloService {
4     @WebMethod
5     public String hello(@WebParam(name = "name") String name) {
6         return "Hello, " + name + "!";
7     }
8 }
```

```
1 wsgen -cp . -s src -keep com.example.HelloService
2 wsgen -cp . -wsdl -d wsdl com.example.HelloService   # generuje też WSDL
```

Opcja `-keep` zachowuje kod źródłowy wygenerowanych klas opakowujących (*wrapper classes*). `wsgen` wymaga skompilowanej klasy implementacyjnej — nie interfejsu SEI. Bez flagi `-wsdl` WSDL generowany jest automatycznie przez kontener przy wdrożeniu (pod adresem `http://host/serwis?wsdl`).

12.3.1. Publikacja w Java SE

W środowisku Java SE serwis JAX-WS można uruchomić bez serwera aplikacji za pomocą klasy `javax.xml.ws.Endpoint`:

```
Endpoint.publish("http://localhost:8080/hello", new HelloService());
```

Klasa `Endpoint` uruchamia wbudowany serwer HTTP (Sun HTTP Server w JDK). Stosowana do testów i prototypowania — nie nadaje się do środowisk produkcyjnych.

Tabela 3. Porównanie strategii

Aspekt	WSDL-first (`wsimport`)	Java-first (`wsgen`)
Punkt wyjścia	Dokument WSDL	Adnotowana klasa Java
Kontrola kontraktu	Pełna	Ograniczona (kształt WSDL zależy od implementacji)
Integracja z systemami zewnętrznymi	Preferowane	Trudniejsze
Szybkość prototypowania	Wolniejsza	Szybsza
Zalecenie	Produkcja, integracja	Prototypy, serwisy wewnętrzne

12.4. Adnotacje kontrolujące interfejs serwisu (JSR-181)

`@javax.jws.WebService`

Oznacza klasę implementującą usługę lub SEI.

Atrybuty: `name`, `targetNamespace`, `serviceName`, `endpointInterface`, `portName`, `wsdlLocation`.

`@javax.jws.WebMethod`

Definiuje metodę jako operację serwisu. Atrybuty: `operationName`, `action`, `exclude`.

```
@WebService
public class MyWebService {
    @WebMethod(operationName = "echoString", action = "urn:EchoString")
    public String echo(String input) { return input; }
}
```

Mapuje się na:

```
<definitions>
  <portType name="MyWebService">
    <operation name="echoString">
      <input message="echoString"/>
      <output message="echoStringResponse"/>
    </operation>
  </portType>
  <binding name="MyWebServiceBinding" type="MyWebService">
```

```
    <operation name="echoString">
      <soap:operation soapAction="urn:EchoString"/>
    </operation>
  </binding>
</definitions>
```

`@javax.jws.WebResult`

Opisuje wartość zwracaną przez metodę. Atrybuty: `name`, `partName`, `targetNamespace`, `header`.

`@javax.jws.WebParam`

Opisuje parametr operacji. Atrybuty: `name`, `partName`, `targetNamespace`, `mode` (`IN`/`OUT`/`INOUT`), `header`.

```
@WebMethod
@WebResult(name = "TPrice", partName = "response")
public TradePrice getLastTradePrice(
    @WebParam(name = "TradePriceReq", partName = "priceRequest")
    TradePriceRequest body);
```

Parametr OUT (użycie `Holder`):

```
@WebMethod
public void getLastTradePrice(TradePriceRequest body,
        @WebParam(mode = WebParam.Mode.OUT)
        Holder<TradePrice> response) {
    response.value = new TradePrice();
    // ...
}
```

`@javax.jws.soap.SOAPBinding`

Określa wiązanie z protokołem SOAP.

- `style` — `document` lub `rpc`
- `use` — `literal` lub `encoded` (domyślnie `literal`)
- `parameterStyle` — `WRAPPED` (domyślnie) lub `BARE`

Przykład document/wrapped:

```
@WebService
@SOAPBinding(style = SOAPBinding.Style.DOCUMENT,
             use = SOAPBinding.Use.LITERAL,
             parameterStyle = SOAPBinding.ParameterStyle.WRAPPED)
public interface StockQuotePortType {
    @WebMethod(operationName = "GetLastTradePrice")
    @WebResult(name = "price")
    public float getLastTradePrice(
            @WebParam(name = "tickerSymbol") String tickerSymbol);
```

```
10 }
```

Mapowanie `document/wrapped` na schemat WSDL (`<types>`):

```
1 <types>
2   <xs:schema>
3     <xs:element name="GetLastTradePrice">
4       <xs:complexType>
5         <xs:sequence>
6           <xs:element name="tickerSymbol" type="xs:string"/>
7         </xs:sequence>
8       </xs:complexType>
9     </xs:element>
10    <xs:element name="GetLastTradePriceResponse">
11      <xs:complexType>
12        <xs:sequence>
13          <xs:element name="price" type="xs:float"/>
14        </xs:sequence>
15      </xs:complexType>
16    </xs:element>
17  </xs:schema>
18 </types>
```

Dla porównania — styl `BARE` (jeden element na parametr, bez wrappera):

```
1 <types>
2   <xs:schema>
3     <xs:element name="tickerSymbol" nillable="true" type="xs:string"/>
4     <xs:element name="price" type="xs:float"/>
5   </xs:schema>
6 </types>
```

`@javax.xml.ws.RequestWrapper` / `@ResponseWrapper`

Pozwalają na zmianę domyślnych właściwości elementu opakowującego w stylu WRAPPED. Atrybuty: `localName`, `targetNamespace`, `className`, `partName`.

```
1 @RequestWrapper(localName = "TradePriceRequest",
2                 className = "ocewsd.stockquote.TradePriceRequest")
3 @ResponseWrapper(localName = "TradePrice",
4                  className = "ocewsd.stockquote.TradePrice")
5 public float getLastTradePrice(@WebParam(name = "tickerSymbol") String tickerSymbol);
```

`@javax.jws.Oneway`

Wskazuje, że operacja jest jednokierunkowa (zwraca `void`, brak wyjątków). Mapuje się na `portType` z samym elementem `<input>`:

```
1 <portType name="MyWebService">
2   <operation name="disconnect">
3     <input message="disconnect"/>
4     <!-- brak <output> i <fault> -->
```

```
  </operation>
</portType>
```

12.5. Adnotacje API klienta

`@javax.xml.ws.WebServiceClient`

Umieszczana nad wygenerowanym rozszerzeniem `javax.xml.ws.Service`. Atrybuty: `name`, `targetNamespace`, `wsdlLocation`.

`@javax.xml.ws.WebEndpoint`

Mapuje metodę `getPortName` na konkretną definicję portu w WSDL. Atrybut: `name`.

Jeśli WSDL definiuje dwa porty, `wsimport` wygeneruje dwie metody w klasie serwisu:

```
<service name="StockQuoteService">
  <port name="StockQuotePortSoap11" binding="StockQuotePortSoap11Binding">
    <soap:address location="http://example.com/stocks"/>
  </port>
  <port name="StockQuotePortSoap12" binding="StockQuotePortSoap12Binding">
    <soap12:address location="http://example.com/stocks"/>
  </port>
</service>
```

```
@WebServiceClient(name = "StockQuoteService", ...)
public class StockQuoteService extends javax.xml.ws.Service {

    @WebEndpoint(name = "StockQuotePortSoap11")
    public StockQuotePortType getStockQuotePortSoap11() { ... }

    @WebEndpoint(name = "StockQuotePortSoap12")
    public StockQuotePortType getStockQuotePortSoap12() { ... }
}
```

`@javax.xml.ws.WebServiceRef`

W środowisku Java EE — wstrzykuje referencję do usługi sieciowej. Atrybuty: `name`, `wsdlLocation`, `type`, `value`.

```
// Referencja do klasy serwisu
@WebServiceRef
public StockQuoteService stockQuoteService;

// Referencja do konkretnego proxy
@WebServiceRef(StockQuoteService.class)
private StockQuoteProvider stockQuoteProvider;
```

12.6. Funkcjonalności (WebServiceFeature)

JAX-WS definiuje trzy funkcjonalności:

- `javax.xml.ws.soap.AddressingFeature` — uruchamia WS-Addressing,
- `javax.xml.ws.soap.MTOMFeature` — uruchamia MTOM/XOP,
- `javax.xml.ws.RespectBindingFeature` — wymusza obsługę rozszerzeń z `required=true`.

Po stronie klienta przekazuje się je przy tworzeniu proxy lub obiektu `Dispatch`:

```
1 Hello port = service.getHelloPort(new MTOMFeature());
2 Dispatch d = service.createDispatch(..., new MTOMFeature());
```

Po stronie serwera — adnotacje nad SIB:

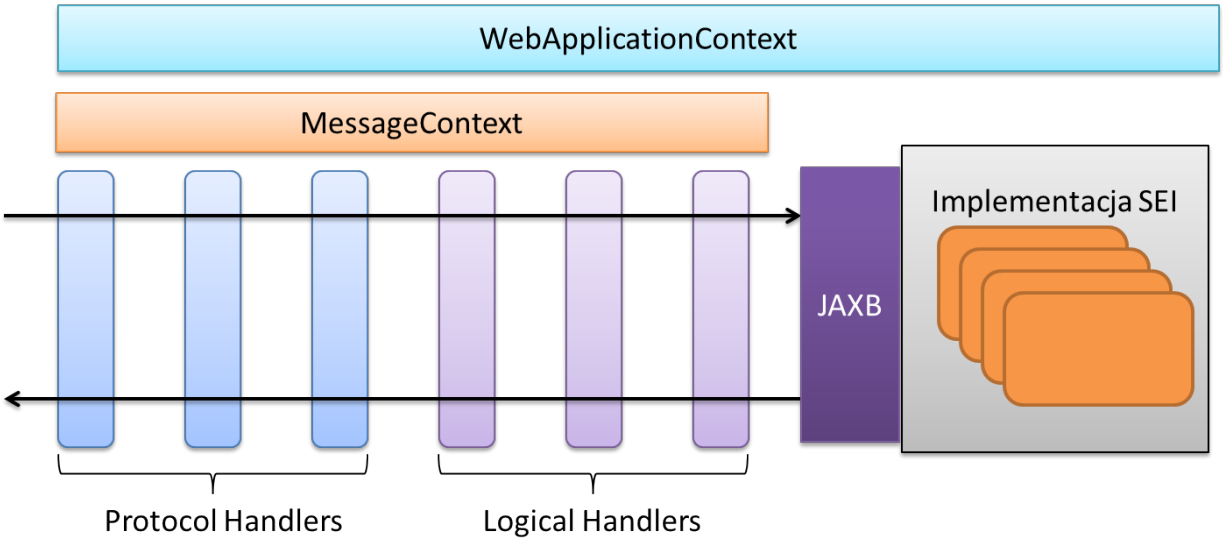
```
1 @Addressing(enabled = true, required = true)
2 @MTOM(enabled = true, threshold = 1024)
3 @WebService(...)
4 public class MyService { ... }
```

12.7. Handler chain

JAX-WS wspiera łańcuch handlerów po stronie klienta i serwera — uruchamianych przed i po każdym wywołaniu metody serwisu. Handlery dzielimy na:

Logiczne
: Dostęp tylko do ciała wiadomości (implementują `LogicalHandler`).

Protokołu
: Dostęp do całej wiadomości (implementują `SOAPHandler`).



Rysunek 27. Handler chain

Dla wiadomości **wychodzącej**: najpierw handlery logiczne, potem protokołu. Dla **przychodzącej**: odwrotnie.

Interfejs `Handler`:

```
1 public interface Handler<C extends MessageContext> {
2     public boolean handleMessage(C context);
3     public boolean handleFault(C context);
4     public void close(MessageContext context);
5 }
```

Przykładowy `SOAPHandler`:

```
1 public class LoggerHandler implements SOAPHandler<SOAPMessageContext> {
2     @Override
3     public boolean handleMessage(SOAPMessageContext ctx) {
4         try { ctx.getMessage().writeTo(System.out); }
5         catch (SOAPException | IOException e) { ... }
6         return true;
7     }
8     @Override public Set<QName> getHeaders() { return null; }
9     @Override public void close(MessageContext ctx) { }
10    @Override public boolean handleFault(SOAPMessageContext ctx) { return true; }
11 }
```

12.7.1. Rejestracja handlerów (klient)

Przez obiekt serwisu:

```
1 service.setHandlerResolver(portInfo -> {
2     List<Handler> chain = new ArrayList<>();
3     chain.add(new LoggingSOAPHandler());
```

```
4     return chain;
5 });
```

Przez konfigurację wiązania proxy:

```
1 BindingProvider provider = (BindingProvider) port;
2 Binding binding = provider.getBinding();
3 List<Handler> chain = binding.getHandlerChain();
4 chain.add(new LoggingSOAPHandler());
5 binding.setHandlerChain(chain);
```

Deklaratywnie przez adnotację `@HandlerChain`:

```
1 @WebService
2 @HandlerChain(file = "StockQuote_handler.xml")
3 public interface StockQuotePortType { }
```

```
1 <handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
2     <handler-chain>
3         <handler>
4             <handler-name>LoggingSOAPHandler</handler-name>
5             <handler-class>com.example.LoggingSOAPHandler</handler-class>
6         </handler>
7     </handler-chain>
8 </handler-chains>
```

Atrybut `file` w `@HandlerChain` jest ścieżką **względną do klasy**, na której adnotacja jest umieszczona — plik XML szukany jest w tym samym pakiecie (katalogu classpath). Ścieżka `"StockQuote_handler.xml"` oznacza, że plik musi znajdować się obok skompilowanego `.class` w katalogu klas (np. `WEB-INF/classes/com/example/StockQuote_handler.xml` gdy SEI jest w pakiecie `com.example`). Niepoprawna ścieżka objawia się wyjątkiem `WebServiceException` przy starcie kontenera bez jasnej informacji o brakującym pliku.

Po stronie serwera dozwolona jest tylko forma deklaratywna (`@HandlerChain` nad SEI/SIB lub w deskryptorze).

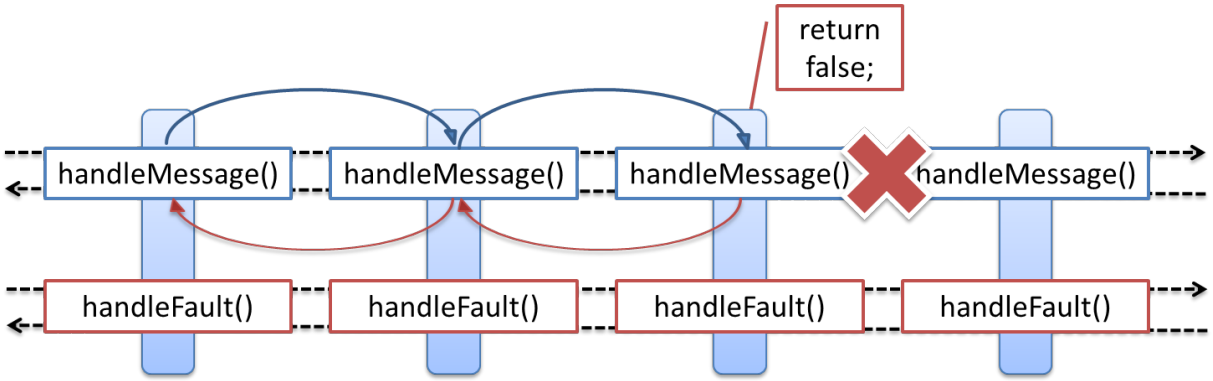
12.7.2. Tryby pracy łańcucha

Normalny

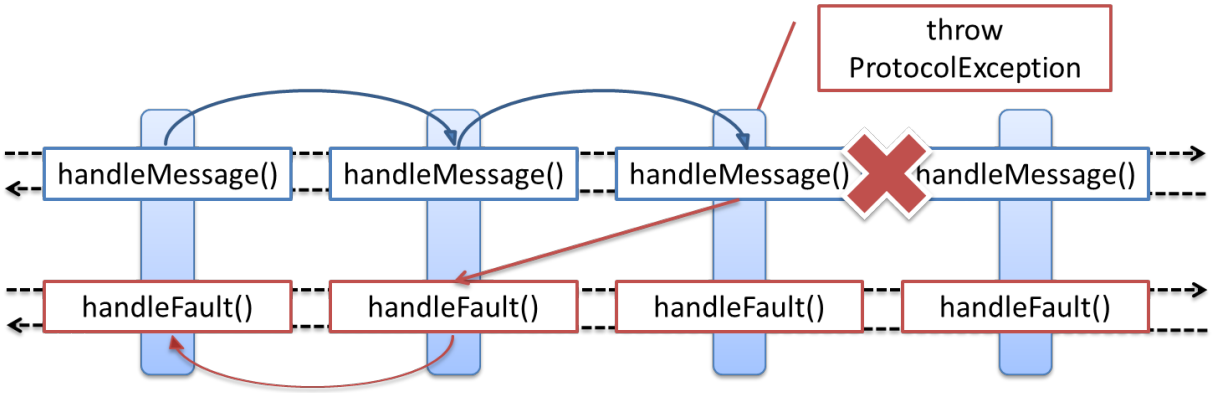
Wywoływane `handleMessage`. Zwrócenie `false` odwraca kierunek przetwarzania.

Obsługi błędu

Wywoływane `handleFault` — uruchamiany po wyrzuceniu `ProtocolException`.



Rysunek 28. Powrót false z handlera



Rysunek 29. Wyjątek ProtocolException w handlerze

Handlery współdzielą stan przez `MessageContext` (rozszerza `Map<String,Object>`). Zakres własności: `MessageContext.Scope.APPLICATION` lub `MessageContext.Scope.HANDLER`.

12.7.2.1. Predefiniowane własności MessageContext

Klucz (stała `MessageContext.*`)	Opis
`MESSAGE_OUTBOUND_PROPERTY` (`javax.xml.ws.handler.message.outbound`)	`Boolean` — `true` dla wiadomości wychodzących
`INBOUND_MESSAGE_ATTACHMENTS` (`javax.xml.ws.binding.attachments.inbound`)	`Map<String,DataHandler>` — załączniki wiadomości przychodzącej
`OUTBOUND_MESSAGE_ATTACHMENTS` (`javax.xml.ws.binding.attachments.outbound`)	`Map<String,DataHandler>` — załączniki wiadomości wychodzącej
`WSDL_DESCRIPTION` (`javax.xml.ws.wsdl.description`)	Lokalizacja pliku WSDL (`java.net.URI`)
`WSDL_SERVICE` (`javax.xml.ws.wsdl.service`)	Nazwa serwisu WSDL (`javax.xml.namespace.QName`)

Klucz (stała `MessageContext.*`)	Opis
`WSDL_PORT` (`javax.xml.ws.wsdl.port`)	Nazwa portu WSDL (`QName`)
`WSDL_OPERATION` (`javax.xml.ws.wsdl.operation`)	Nazwa operacji WSDL (`QName`)
`HTTP_RESPONSE_CODE` (`javax.xml.ws.http.response.code`)	Kod odpowiedzi HTTP (`Integer`)
`HTTP_RESPONSE_HEADERS` (`javax.xml.ws.http.response.headers`)	Nagłówki odpowiedzi HTTP (`Map<String,List<String>>`)
`HTTP_REQUEST_METHOD` (`javax.xml.ws.http.request.method`)	Metoda żądania HTTP (`String`)
`HTTP_REQUEST_HEADERS` (`javax.xml.ws.http.request.headers`)	Nagłówki żądania HTTP (`Map<String,List<String>>`)
`QUERY_STRING` (`javax.xml.ws.http.request.querystring`)	Query string z żądania HTTP (`String`)
`SERVLET_CONTEXT` (`javax.xml.ws.servlet.context`)	`javax.servlet.ServletContext`
`SERVLET_REQUEST` (`javax.xml.ws.servlet.request`)	`javax.servlet.http.HttpServletRequest`
`SERVLET_RESPONSE` (`javax.xml.ws.servlet.response`)	`javax.servlet.http.HttpServletResponse`

12.8. Komunikacja asynchroniczna

Metody asynchroniczne dostępne są z SEI w dwóch odmianach (postfix `Async`):

Odpytywania (polling):

```
1 Response<GetQuoteResponse> resp = port.getQuoteAsync("MHP");
2 while (!resp.isDone()) { /* inne operacje */ }
3 GetQuoteResponse output = resp.get();
```

Metoda zwrotna (callback):

```
1 port.getQuoteAsync("MHP", response -> {
2     try {
```

```
        GetQuoteResponse output = response.get();
        System.out.println(output.getReturn());
    } catch (ExecutionException | InterruptedException e) {
        e.printStackTrace();
    }
});
```

Generowanie metod asynchronicznych wymaga konfiguracji narzędzia `wsimport`:

```
<bindings xmlns="http://java.sun.com/xml/ns/jaxws">
    <enableAsyncMapping>true</enableAsyncMapping>
</bindings>
```

```
wsimport -b <plik-konfiguracyjny> -s src -keep stockquote.wsdl
```

12.9. Obsługa załączników

JAX-WS zapewnia pełne wsparcie dla:

- **MTOM** — całkowicie zautomatyzowany, wymaga tylko włączenia wsparcia,
- **WS-I Attachments Profile** — przez adnotację `@javax.xml.bind.annotation.XmlAttachmentRef` (tylko na polach/parametrach typu `DataHandler`).

```
@XmlRootElement
class Foo {
    @XmlAttachmentRef
    @XmlAttribute
    DataHandler data;

    @XmlAttachmentRef
    @XmlElement
    DataHandler body;
}
```

W SEI:

```
@WebMethod(operationName = "addPerson")
public int add(@WebParam(name = "name") String name,
               @WebParam(name = "surname") int surname,
               @XmlAttachmentRef DataHandler photo);
```

12.10. Tworzenie klienta w sposób dynamiczny (Dispatch)

Gdy kontrakt nie jest znany w czasie kompilacji, można użyć *Dynamic Invocation Interface* (DII). `Dispatch<T>` pozwala wysyłać wiadomości SOAP bez wygenerowanego SEI

— programista sam buduje XML żądania i przetwarza XML odpowiedzi.

Interfejs `Dispatch<T>`:

```
1 public interface Dispatch<T> extends BindingProvider {
2     public T invoke(T msg);
3     public Response<T> invokeAsync(T msg);
4     public Future<?> invokeAsync(T msg, AsyncHandler<T> handler);
5     public void invokeOneWay(T msg);
6 }
```

Obsługiwane typy: `Source`, `SOAPMessage`, `DataSource`, JAXB.

12.10.1. Tryb PAYLOAD vs MESSAGE

`Service.Mode` określa co dokładnie jest treścią parametru `T`:

Tryb	Zawartość T	Kiedy stosować
`Service.Mode.PAYLOAD` (domyślny)	Tylko zawartość elementu `<Body>` (bez koperty i nagłówka)	Prosty dostęp do ciała — nagłówki zarządza JAX-WS
`Service.Mode.MESSAGE`	Kompletna koperta SOAP (Envelope + Header + Body)	Gdy trzeba odczytywać lub ustawiać nagłówki SOAP (np. WS-Security, WS-Addressing) ręcznie

Przykład — `Dispatch<Source>` w trybie PAYLOAD

```
 1 String endpointUrl = "http://example.com/stockquote";
 2 QName serviceName = new QName("http://example.com/stockquote.wsdl", "StockQuoteService");
 3 QName portName    = new QName("http://example.com/stockquote.wsdl", "StockQuotePort");
 4
 5 Service service = Service.create(serviceName);
 6 service.addPort(portName, SOAPBinding.SOAP11HTTP_BINDING, endpointUrl);
 7
 8 Dispatch<Source> dispatch = service.createDispatch(
 9         portName, Source.class, Service.Mode.PAYLOAD);
10
11 // Ciało żądania — tylko zawartość Body, bez koperty
12 String payloadXml =
13     "<GetLastTradePrice xmlns='http://example.com/stockquote.xsd'>" +
14     "  <tickerSymbol>USDPLN</tickerSymbol>" +
15     "</GetLastTradePrice>";
16
17 Source request = new StreamSource(new StringReader(payloadXml));
18 Source response = dispatch.invoke(request);
19
20 // Przetwarzanie odpowiedzi przez TrAX/DOM
21 TransformerFactory tf = TransformerFactory.newInstance();
22 Transformer t = tf.newTransformer();
23 StringWriter sw = new StringWriter();
24 t.transform(response, new StreamResult(sw));
25 System.out.println(sw.toString());
```

Przykład — `Dispatch<SOAPMessage>` w trybie MESSAGE

```
Dispatch<SOAPMessage> dispatch = service.createDispatch(
        portName, SOAPMessage.class, Service.Mode.MESSAGE);

// Budowanie pełnej koperty SOAP przez SAAJ
MessageFactory mf = MessageFactory.newInstance();
SOAPMessage soapRequest = mf.createMessage();
SOAPBody body = soapRequest.getSOAPBody();
QName bodyName = new QName("http://example.com/stockquote.xsd", "GetLastTradePrice", "m");
SOAPBodyElement bodyElement = body.addBodyElement(bodyName);
bodyElement.addChildElement("tickerSymbol").addTextNode("USDPLN");

// Dodanie nagłówka (np. korelacja, security token)
SOAPHeader header = soapRequest.getSOAPHeader();
Name hName = soapRequest.getSOAPPart().getEnvelope()
                        .createName("TransactionId", "tx", "http://example.com/tx");
SOAPHeaderElement txHeader = header.addHeaderElement(hName);
txHeader.addTextNode("TX-12345");

SOAPMessage soapResponse = dispatch.invoke(soapRequest);
SOAPBody responseBody = soapResponse.getSOAPBody();
System.out.println(responseBody.getTextContent());
```

Tryb `MESSAGE` z typem `SOAPMessage` daje pełną kontrolę nad kopertą SOAP — identyczna możliwość co `SOAPHandler`, lecz po stronie klienta bez pośrednictwa łańcucha handlerów.

12.11. Niskopoziomowe tworzenie serwisu (Provider)

Gdy potrzebujemy przetwarzać wiadomości w sposób ogólny (np. pośrednik, XSLT), używamy interfejsu `javax.xml.ws.Provider<T>`:

```
@ServiceMode(value = Service.Mode.PAYLOAD)
@WebServiceProvider(portName = "StockQuotePort",
                    serviceName = "StockQuoteService",
                    wsdlLocation = "StockQuoteService.wsdl")
public class StockQuoteProviderImpl implements Provider<Source> {
    public Source invoke(Source source) {
        // przetwarzanie żądania i generowanie odpowiedzi
    }
}
```

`@WebServiceProvider` ma parametry: `portName`, `serviceName`, `targetNamespace`, `wsdlLocation`. Wadą jest brak możliwości automatycznego generowania kontraktu.

`@ServiceMode` przyjmuje wartości `Service.Mode.MESSAGE` lub `Service.Mode.PAYLOAD` (domyślny).

12.12. Obsługa wyjątków

JAX-WS definiuje hierarchię:

- `WebServiceException` (bazowy)
  - `ProtocolException`
    - `SOAPFaultException`
    - `HTTPException`

Własne wyjątki mapujące się na SOAP Fault muszą:

1. być zadeklarowane w SEI,
2. mieć adnotację `@WebFault`,
3. posiadać metodę `FaultBean getFaultInfo()`,
4. mieć konstruktory `(String, FaultBean)` i `(String, FaultBean, Throwable)`.

```
@WebFault(name = "faultDetail", targetNamespace = "...")
class OperationException extends Exception {
    OperationException(String message, FaultDetail faultInfo) { ... }
    OperationException(String message, FaultDetail faultInfo, Throwable cause) { ... }
    FaultDetail getFaultInfo() { ... }
}
```

Atrybuty `@WebFault`: `name`, `targetNamespace`, `faultBean`, `messageName`.

12.13. Wiązania wspierane przez JAX-WS

Adnotacja `@javax.xml.ws.BindingType` nad SEI lub `Provider` definiuje rodzaj wiązania:

Wiązanie	Stała
SOAP 1.1/HTTP	`SOAPBinding.SOAP11HTTP_BINDING`
SOAP 1.2/HTTP	`SOAPBinding.SOAP12HTTP_BINDING`
SOAP 1.1/HTTP + MTOM	`SOAPBinding.SOAP11HTTP_MTOM_BINDING`
SOAP 1.2/HTTP + MTOM	`SOAPBinding.SOAP12HTTP_MTOM_BINDING`
XML/HTTP	`HTTPBinding.HTTP_BINDING`

Interfejs `SOAPBinding` pozwala na definiowanie ról, ustawianie łańcucha handlerów i włączanie MTOM:

```
public interface SOAPBinding extends Binding {
    public static final String SOAP11HTTP_BINDING =
            "http://schemas.xmlsoap.org/wsdl/soap/http";
    public static final String SOAP12HTTP_BINDING =
            "http://www.w3.org/2003/05/soap/bindings/HTTP/";
    public static final String SOAP11HTTP_MTOM_BINDING =
            "http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true";
    public static final String SOAP12HTTP_MTOM_BINDING =
            "http://www.w3.org/2003/05/soap/bindings/HTTP/?mtom=true";

    public Set<String> getRoles();
```

```
    public void setRoles(Set<String> roles);
    public boolean isMTOMEnabled();
    public void setMTOMEnabled(boolean flag);
}
```

Interfejs `HTTPBinding`:

```
public interface HTTPBinding extends Binding {
    public static final String HTTP_BINDING =
            "http://www.w3.org/2004/08/wsdl/http";
}
```

Wiązanie XML/HTTP realizowane jest tylko przez obiekty `Provider` i obsługuje wyłącznie `LogicalHandler`. Skojarzony wyjątek `HTTPException` pozwala na określenie HTTP status code odpowiedzi.

12.14. JAX-WS — podsumowanie

JAX-WS integruje następujące technologie w jednym modelu programowania:

Komponent	Rola w JAX-WS
JAXB	Wiązanie typów XMLSchema z klasami Javy — marshal/unmarshal przy każdym wywołaniu
SAAJ	Dostęp do wiadomości SOAP na poziomie protokołu przez `SOAPHandler`
WS-Addressing	Asynchroniczne wywołania i routing wiadomości (`@Addressing`, `AddressingFeature`)
MTOM/XOP	Wydajne przesyłanie danych binarnych (`@MTOM`, `MTOMFeature`, `DataHandler`)
WSDL 1.1	Kontrakt serwisu — importowany przez `wsimport` lub generowany przez `wsgen`

Programista wybiera między:

- `wsimport` (*contract-first*) — pełna kontrola kontraktu; preferowane przy integracji z zewnętrznymi systemami,
- `wsgen` (*code-first*) — szybkie prototypowanie; kształt WSDL zależy od implementacji JAX-WS.

Kolejny rozdział opisuje WSEE (JSR-109) — specyfikację wdrażania serwisów JAX-WS w kontenerach Java EE: zasady pakowania archiwów WAR i EJB-JAR, konfigurację deskryptorów `webservices.xml` i wstrzykiwanie referencji przez `@WebServiceRef`.

Część IV: Wdrożenie i transport

13. Web Services for Java EE — WSEE

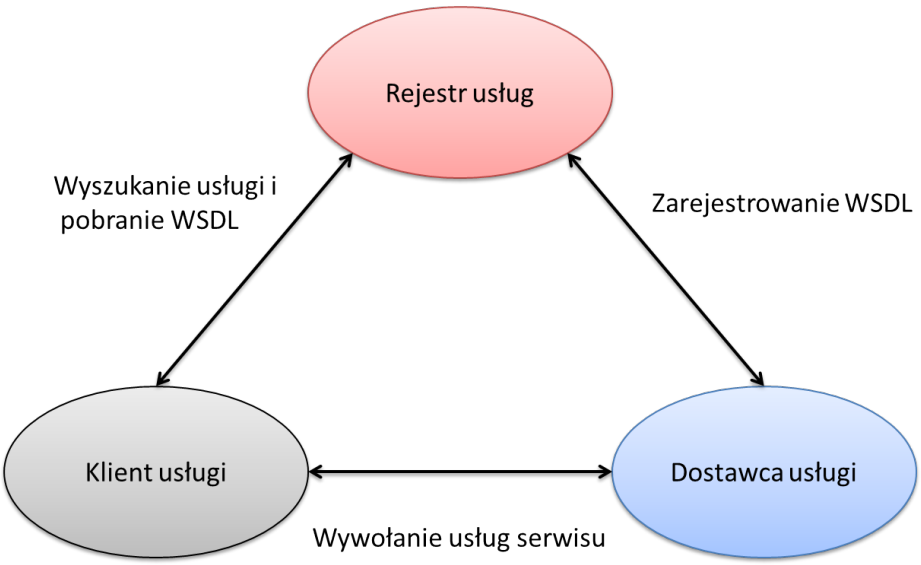
WSEE (JSR-109) definiuje zasady pakowania, wdrażania i konfigurowania serwisów JAX-WS w kontenerach Java EE — jej przestrzeganie gwarantuje przenośność między serwerami aplikacji zgodnymi z Java EE 6 (GlassFish, JBoss AS 7, WebLogic, WebSphere). Specyfikacja opisuje jak wdrożyć serwis oparty na adnotacjach JAX-WS w kontenerze EJB lub Web, jak spakować archiwum WAR lub EJB-JAR i jak skonfigurować deskryptory wdrożenia.

Architektura określa:

- tworzenie i uruchamianie klientów usług w kontenerach Java EE (EJB, Web, Application Client),
- pakowanie i umieszczanie usług na serwerach Java EE (jako komponentów EJB i Web),
- mapowanie funkcjonalności Java EE na specyfikacje usług,
- wsparcie mapowania wywołań SOAP na metody komponentów,
- wymóg generowania i udostępniania dokumentu WSDL,
- deklaratywny (deskryptory) sposób konfiguracji usług,
- wsparcie dla SOAP 1.1/1.2, SwA, WSDL 1.1, UDDI 1.0 przez HTTP(S).

13.1. Ogólna architektura usługi sieciowej

Poza usługą i jej klientem istnieje rejestr usług, w którym klient może wyszukiwać usługi i pobierać opis ich funkcjonalności w postaci dokumentów WSDL.



Rysunek 30. Architektura WSEE

13.2. Metody implementacji usługi

- jako *Stateless Session Bean* lub *Singleton Session Bean* (od EJB 3.1) w kontenerze EJB,
- jako zwykła klasa Java (POJO) z adnotacjami JAX-WS.

13.3. Model programowania usług sieciowych

13.3.1. Port component

Podstawowym zbiorem elementów usługi z punktu widzenia kontenera jest *port component*:

***Service Endpoint Interface* (SEI)**
: Interfejs określający metody implementowane przez daną usługę. Może być podany jawnie lub generowany z SIB.

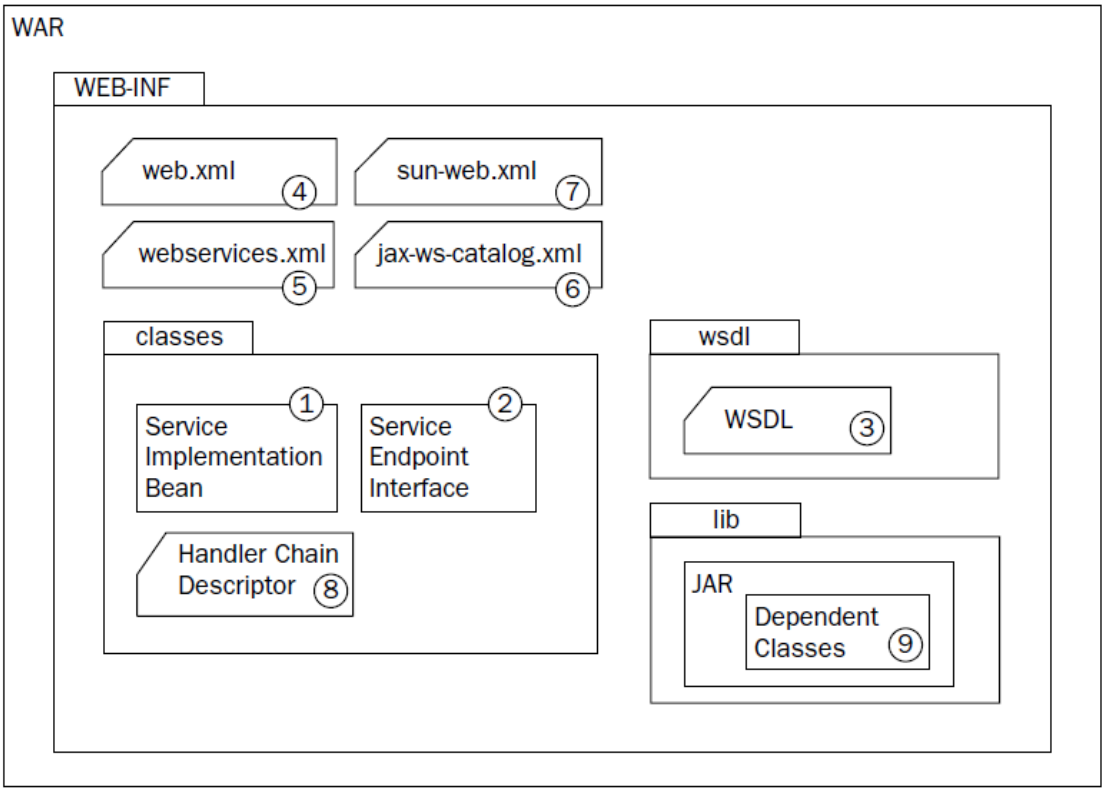
***Service Implementation Bean* (SIB)**
: Klasa Javy implementująca metody biznesowe usługi (SEI).

Security Role Reference
: Logiczne role zapisane w deskryptorze — powiązane z fizycznymi rolami serwera.

13.3.2. Pakowanie

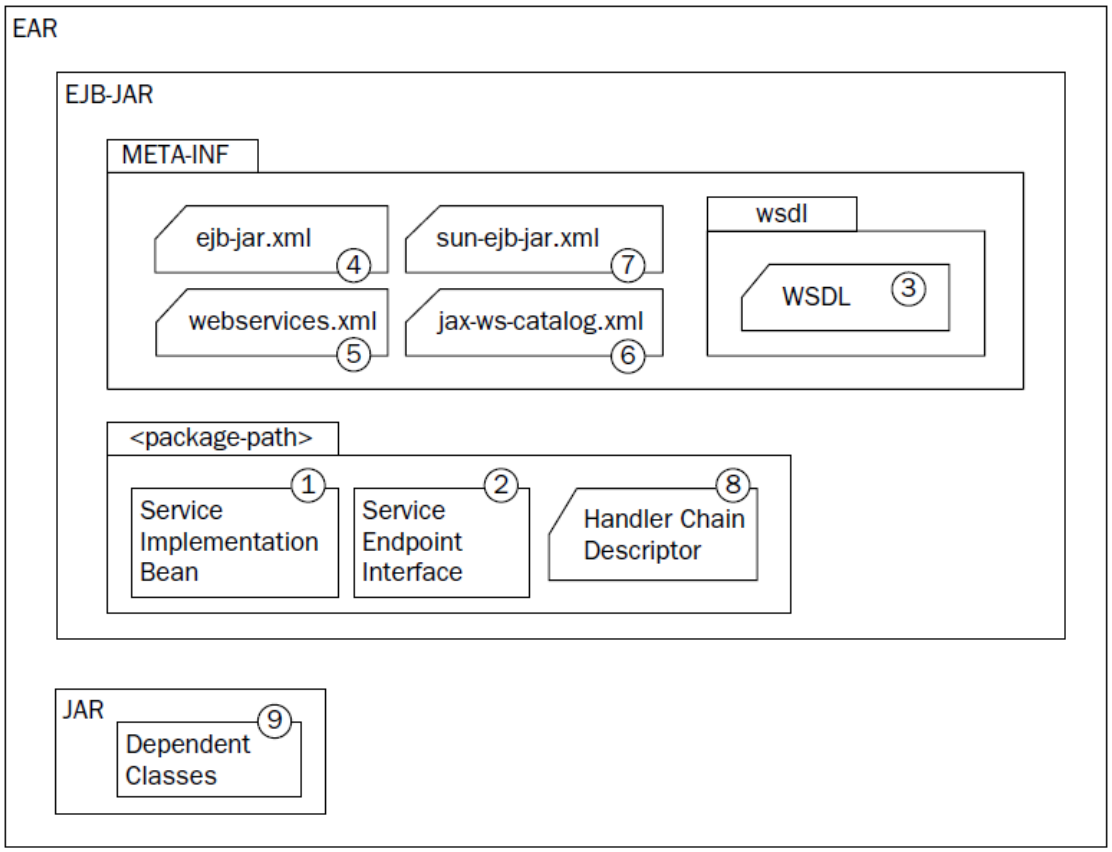
Komponenty mogą być przenoszone w archiwach:

- **WAR** — katalog `WEB-INF/` zawiera: `web.xml`, `webservices.xml`, `jax-ws-catalog.xml`, katalog `wsdl/` z dokumentem WSDL, katalog `classes/` z klasami, `lib/` z JAR-ami.



Rysunek 31. Struktura WAR

- **EJB-JAR** (wewnątrz EAR) — katalog `META-INF/` zawiera: `ejb-jar.xml`, `webservices.xml`, `jax-ws-catalog.xml`, katalog `wsdl/`.



Rysunek 32. Struktura EJB-JAR

13.4. Deskryptory

Deskryptor `webservices.xml` przechowuje konfigurację usług sieciowych.

Opis elementów `port-component`:

Element	Opis
`port-component-name`	Nazwa portu
`service-endpoint-interface`	Wskazuje SEI
`service-impl-bean`	Nazwa komponentu wejściowego (serwlet lub EJB)
`wsdl-file`	Względna ścieżka do WSDL
`wsdl-service`	Pełna nazwa (z przestrzenią nazw) dla web serwisu
`wsdl-port`	Pełna nazwa portu
`enable-mtom`	Konfiguracja MTOM
`addressing`	Konfiguracja WS-Addressing
`respect-binding`	Pedantyczne sprawdzanie sekcji binding WSDL
`protocol-binding`	Wiązanie: `SOAP11_HTTP`, `SOAP12_HTTP`, `##SOAP11_HTTP_MTOM`, ...

Element	Opis
`handler-chains`	Lista łańcuchów handlerów

Przykład — usługa jako *stateless session bean*:

`ejb-jar.xml`

```
<ejb-jar>
    <enterprise-beans>
        <session>
            <display-name>HelloServiceEJB</display-name>
            <ejb-name>HelloServiceEJB</ejb-name>
            <service-endpoint>HelloServiceSEI</service-endpoint>
            <ejb-class>HelloServiceBean</ejb-class>
            <session-type>Stateless</session-type>
        </session>
    </enterprise-beans>
</ejb-jar>
```

`webservices.xml`

```
<webservices>
    <webservice-description>
        <webservice-description-name>HelloServiceEJB</webservice-description-name>
        <wsdl-file>META-INF/wsdl/HelloServiceEJB.wsdl</wsdl-file>
        <port-component>
            <port-component-name>HelloServiceInfPort</port-component-name>
            <wsdl-port xmlns:my="http://hello">my:HelloServiceInfPort</wsdl-port>
            <service-endpoint-interface>HelloServiceSEI</service-endpoint-interface>
            <service-impl-bean>
                <ejb-link>HelloServiceEJB</ejb-link>
            </service-impl-bean>
        </port-component>
    </webservice-description>
</webservices>
```

Konfiguracja z klasami `WSServletContextListener` i `WSServlet` jest specyficzna dla implementacji Metro (JAX-WS RI). Na serwerach Java EE 6 zgodnych ze specyfikacją (GlassFish, JBoss AS 7+) serwis JAX-WS wykrywany jest automatycznie na podstawie adnotacji `@WebService` — deskryptor `web.xml` nie jest wymagany.

`web.xml` — *przykład z Metro (JAX-WS RI)*

```
<web-app>
    <listener>
        <listener-class>
            com.sun.xml.ws.transport.http.servlet.WSServletContextListener
        </listener-class>
    </listener>
    <servlet>
        <servlet-name>HelloService</servlet-name>
        <servlet-class>
```

```
            com.sun.xml.ws.transport.http.servlet.WSServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>HelloService</servlet-name>
        <url-pattern>/sayhello</url-pattern>
    </servlet-mapping>
</web-app>
```

13.5. Klient serwisu sieciowego w Java EE

W kontenerze Java EE referencja do serwisu wstrzykiwana jest adnotacją `@WebServiceRef`:

```
@Stateless
public class OrderEJB {

    @WebServiceRef(wsdlLocation = "META-INF/wsdl/StockQuoteService.wsdl")
    private StockQuoteService stockQuoteService;

    public float getPrice(String ticker) {
        StockQuotePortType port = stockQuoteService.getStockQuotePort();
        return port.getLastTradePrice(ticker);
    }
}
```

Kontener zarządza cyklem życia obiektu serwisu i wstrzykuje go przed pierwszym użyciem. WSDL wskazany w `wsdlLocation` pobierany jest z archiwum (ścieżka względna do `META-INF/wsdl/`) lub spod podanego adresu URL (ścieżka absolutna).

W środowisku Java SE (lub bez wstrzykiwania) port tworzony jest ręcznie:

```
URL wsdlUrl = new URL("http://example.com/stockquote?wsdl");
QName serviceName = new QName("http://example.com/stockquote.wsdl",
                              "StockQuoteService");
Service service = Service.create(wsdlUrl, serviceName);
StockQuotePortType port = service.getPort(StockQuotePortType.class);
float price = port.getLastTradePrice("USDPLN");
```

13.6. WSEE — podsumowanie

WSEE (JSR-109) definiuje zasady przenośności serwisów JAX-WS w kontenerach Java EE:

- serwis jako *Stateless Session Bean* lub *Singleton* dziedziczy transakcyjność, bezpieczeństwo i wstrzykiwanie zasobów kontenera EJB,
- serwis jako POJO (`@WebService`) wdrażany jest w kontenerze Web (WAR),
- deskryptor `webservices.xml` pozwala nadpisać ustawienia adnotacji bez zmiany kodu (MTOM, WS-Addressing, handler chains, protocol-binding),

- kontener generuje i udostępnia WSDL pod adresem `http://host/kontekst/serwis?wsdl`,
- klient Java EE uzyskuje referencję przez `@WebServiceRef`; klient Java SE — przez `Service.create()`.

Kolejne rozdziały opisują JMS (*Java Message Service*) — asynchroniczne przesyłanie wiadomości przez kolejki i tematy, oraz mechanizmy zabezpieczenia serwisów sieciowych (WS-Security).

14. Messaging Oriented Middleware — JMS

Wszystkie omówione wcześniej technologie komunikacji — HTTP, SOAP, JAX-WS — zakładają model synchroniczny: nadawca czeka na odpowiedź odbiorcy. Gdy przetwarzanie trwa długo, gdy system docelowy bywa chwilowo niedostępny lub gdy jedna wiadomość powinna trafić do wielu odbiorców jednocześnie, lepszym rozwiązaniem jest *messaging oriented middleware* (MOM).

Protokół HTTP może nie być najlepszym rozwiązaniem, gdy:

- przesyłane komunikaty nie wymagają odpowiedzi,
- potrzeba dużo czasu na przetworzenie żądania,
- nadawca nie jest adresatem ewentualnej odpowiedzi,
- jest wielu adresatów komunikatu,
- wymagane jest wsparcie dla rozproszonych transakcji,
- wymagany jest bardziej stabilny model komunikacji.

W takim przypadku zamiast HTTP można użyć kolejek wiadomości. W Java EE standard ten definiuje specyfikacja *Java Message Service* (JMS 1.1, JSR-914).

Działanie MOM można porównać do usług pocztowych — nadawca wysyła list, a poczta dba o dostarczenie. MOM oddziela klientów od siebie, uwierzytelnia nadawców i odbiorców, nadzoruje ruch i gwarantuje dostarczenie wiadomości nawet gdy odbiorca jest chwilowo nieaktywny.

Systemy MOM oferują dwa modele:

- **point-to-point** (*queue*) — wiadomość od nadawcy do jednego odbiorcy,
- **publish/subscriber** (*topic*) — jeden komunikat do wszystkich zainteresowanych odbiorców.

14.1. JMS API

API JMS zdefiniowane jest w pakiecie `javax.jms`.

Obiekt	Opis
`ConnectionFactory`	Punkt wejściowy — zawiera informacje do nawiązania połączenia z brokerem; thread-safe
`Connection`	Aktywne połączenie z systemem dostawcy
`Session`	Kontekst wymiany wiadomości; tworzy Producer, Consumer i wiadomości
`MessageProducer`	Wysyła wiadomości do kolejki/tematu

Obiekt	Opis
`MessageConsumer`	Pobiera wiadomości z kolejki/tematu
`Destination`	Identyfikuje kolejkę lub temat (obiekt administrowalny)

Wszystkie podstawowe interfejsy posiadają rozszerzenia dla kolejek (prefix `Queue`) i tematów (prefix `Topic`).

14.1.1. Typy wiadomości

JMS definiuje pięć typów wiadomości dziedziczących po interfejsie `javax.jms.Message`:

Typ	Opis
`TextMessage`	Ciało tekstowe (`String`) — najczęściej używany; naturalny dla XML i SOAP
`BytesMessage`	Surowe bajty (`byte[]`) — dla danych binarnych
`MapMessage`	Pary klucz-wartość (`String` → typy proste)
`ObjectMessage`	Serializowany obiekt Java (musi implementować `Serializable`)
`StreamMessage`	Strumień wartości prostych — odczyt w kolejności zapisu

Standardowe pobranie fabryki z JNDI:

```
1 Context ctx = new InitialContext();
2 QueueConnectionFactory queueCF =
3     (QueueConnectionFactory) ctx.lookup("QueueConnectionFactory");
```

Lub bezpośrednio (np. Apache ActiveMQ):

```
1 ConnectionFactory connectionFactory =
2     new org.apache.activemq.ActiveMQConnectionFactory(url_to_broker);
```

Wysyłanie wiadomości:

```
 1 Connection connection = connectionFactory.createConnection();
 2 connection.start();
 3 Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
 4 Topic topic = session.createTopic("myTopic");
 5 MessageProducer producer = session.createProducer(topic);
 6
 7 TextMessage message = session.createTextMessage();
 8 message.setText("HELLO JMS WORLD");
 9 producer.send(message);
10
```

```
11 connection.close();
```

Asynchroniczny odbiorca:

```
1 MessageConsumer consumer = session.createConsumer(topic);
2 consumer.setMessageListener(new MessageListener() {
3     public void onMessage(Message message) {
4         try {
5             if (message instanceof TextMessage) {
6                 System.out.println(((TextMessage) message).getText());
7             }
8         } catch (JMSException e) { e.printStackTrace(); }
9     }
10 });
```

14.1.2. Parametry sesji i tryby potwierdzeń

Metoda `createSession(transacted, acknowledgeMode)` przyjmuje dwa parametry:

`transacted = true`

Sesja transakcyjna — wiadomości zatwierdzane przez `session.commit()` lub wycofywane przez `session.rollback()`. Parametr `acknowledgeMode` jest wówczas ignorowany.

`Session.AUTO_ACKNOWLEDGE`

Broker uznaje wiadomość za dostarczoną automatycznie po powrocie z `receive()` lub `onMessage()`.

`Session.CLIENT_ACKNOWLEDGE`

Konsument jawnie wywołuje `message.acknowledge()` po pomyślnym przetworzeniu. Potwierdzenie obejmuje wszystkie dotychczas odebrane wiadomości w sesji.

`Session.DUPS_OK_ACKNOWLEDGE`

Leniwe potwierdzanie — dopuszcza duplikaty przy awarii; zmniejsza narzut sieciowy.

14.1.3. Synchroniczny odbiór

```
1 MessageConsumer consumer = session.createConsumer(queue);
2 Message msg = consumer.receive();         // blokuje w nieskończoność
3 Message msg = consumer.receive(5000);     // czeka maks. 5 000 ms
4 Message msg = consumer.receiveNoWait();   // zwraca null gdy brak wiadomości
```

14.1.4. Selektory wiadomości

Selektor ogranicza odbiór do wiadomości spełniających warunek SQL-92 — ewaluowany

po stronie brokera; konsument nie pobiera odfiltrowanych wiadomości:

```
1 MessageConsumer consumer = session.createConsumer(queue,
2     "priority > 3 AND region = 'EU'");
```

Selektor może operować na nagłówkach JMS i właściwościach wiadomości (`setStringProperty`, `setIntProperty`) — nie może odwoływać się do ciała wiadomości.

14.1.5. Trwałe subskrypcje (Topic)

W modelu pub/sub wiadomości opublikowane, gdy konsument jest offline, są domyślnie tracone. *Durable subscriber* przechowuje wiadomości do momentu ich odebrania:

```
1 connection.setClientID("clientApp");
2 TopicSubscriber subscriber =
3     session.createDurableSubscriber(topic, "orderUpdates");
```

Subskrypcja identyfikowana jest przez `clientID` połączenia i nazwę subskrypcji. Usunięcie trwałej subskrypcji: `session.unsubscribe("orderUpdates")`.

14.1.6. Nagłówki wiadomości JMS

Nagłówek	Opis
`JMSDestination`	Informacja gdzie wiadomość została wysłana
`JMSDeliveryMode`	`NON_PERSISTENT` lub `PERSISTENT` (wytrzyma awarię systemu kolejkowego)
`JMSMessageID`	Identyfikator wiadomości (używany w `JMSCorrelationID` odpowiedzi)
`JMSTimestamp`	Czas dostarczenia do kolejki
`JMSReplyTo`	Kolejka na którą ma zostać odesłana odpowiedź (ustawiana przez klienta)
`JMSCorrelationID`	Identyfikator wiadomości, na którą ta jest odpowiedzią
`JMSType`	Identyfikator typu wiadomości

Poza nagłówkami, wiadomości mogą mieć właściwości dowolnych typów (`boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, `String`). Na ich podstawie buduje się *selektory* — odfiltrowywanie wiadomości bez pobierania.

14.2. Message Driven Bean (MDB)

W kontenerze Java EE zalecanym sposobem konsumowania wiadomości JMS jest

Message Driven Bean (MDB). MDB jest bezstanowym komponentem EJB wywoływanym automatycznie przez kontener gdy wiadomość dotrze do kolejki lub tematu — kontener zarządza połączeniem, sesją i konsumentem.

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType",
                              propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "destination",
                              propertyValue = "jms/OrderQueue"),
    @ActivationConfigProperty(propertyName = "acknowledgeMode",
                              propertyValue = "Auto-acknowledge")
})
public class OrderProcessorMDB implements MessageListener {

    @EJB
    private OrderService orderService;

    @Override
    public void onMessage(Message message) {
        try {
            if (message instanceof TextMessage) {
                orderService.process(((TextMessage) message).getText());
            }
        } catch (JMSException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Właściwości MDB konfigurowane przez `@ActivationConfigProperty`:

Właściwość	Opis
`destinationType`	`javax.jms.Queue` lub `javax.jms.Topic`
`destination`	Nazwa JNDI kolejki lub tematu
`acknowledgeMode`	`Auto-acknowledge` lub `Dups-ok-acknowledge`
`subscriptionDurability`	`Durable` (tylko Topic) — trwała subskrypcja
`messageSelector`	Selektor SQL-92 filtrujący wiadomości po stronie brokera

Zalety MDB w porównaniu z ręcznym `MessageConsumer`:

- kontener zarządza cyklem życia — brak kodu zarządzającego `Connection` i `Session`,
- uczestniczy w transakcjach zarządzanych przez kontener (CMT — domyślnie `Required`),
- obsługuje wstrzykiwanie zasobów (`@EJB`, `@PersistenceContext`, `@Resource`),
- bezpieczny wątkowo — kontener może uruchamiać wiele instancji równolegle.

14.3. SOAP over JMS

Specyfikacja *SOAP over Java Message Service 1.0* (W3C) standaryzuje przesyłanie wiadomości SOAP przez systemy JMS dla SOAP 1.1 i 1.2.

Ciałem wiadomości JMS (`BytesMessage` lub `TextMessage`) jest koperta SOAP. Wspierane wzorce: `request/response` i `one-way`.

Wymagane właściwości wiadomości:

Właściwość	Opis
`SOAPJMS_requestURI`	URI JMS bez parametrów zapytania
`SOAPJMS_bindingVersion`	Wersja specyfikacji SOAP/JMS (aktualnie 1.0)
`SOAPJMS_soapAction`	Odpowiednik nagłówka `SOAPAction` z HTTP
`SOAPJMS_isFault`	`true` jeśli ciało zawiera SOAPFault
`SOAPJMS_targetService`	Serwis, do którego kierowane jest żądanie
`SOAPJMS_contentType`	Typ wiadomości, np. `text/xml`

Wiązanie WSDL dla SOAP/JMS — transport `http://www.w3.org/2010/soapjms/`, adres jako JMS URI:

```
<binding name="StockQuoteSoapJMSBinding" type="tns:StockQuotePortType">
    <soap11:binding style="document"
                    transport="http://www.w3.org/2010/soapjms/"/>
</binding>

<service name="StockQuoteService">
    <port name="StockQuotePort_jms"
          binding="tns:StockQuoteSoapJMSBinding">
        <soap11:address location="jms:jndi:myQueue?
            targetService=stockquote&amp;priority=8&amp;
            replyToName=interested"/>
    </port>
</service>
```

14.4. JMS w architekturze serwisów sieciowych

JMS rozszerza architekturę serwisów o komunikację asynchroniczną i odporną na awarie:

Element	Rola
`Queue` (point-to-point)	Jeden nadawca, jeden odbiorca; gwarancja dostarczenia; wiadomość czeka aż do odebrania

Element	Rola
`Topic` (pub/sub)	Jeden nadawca, wielu odbiorców; trwałość opcjonalna (*durable subscriber*)
MDB	Konsumowanie wiadomości po stronie EJB — transakcje CMT, wstrzykiwanie zasobów
SOAP over JMS	Zamiana transportu HTTP na JMS przy zachowaniu kontraktu WSDL

Specyfikacja SOAP over JMS pozwala serwisowi JAX-WS działać przez JMS jako transport — wystarczy zmienić adres w WSDL i wiązanie; logika biznesowa pozostaje bez zmian. Wzorzec ten jest przydatny, gdy wymagana jest niezawodna asynchroniczna integracja między systemami.

Kolejne rozdziały opisują WS-Security i mechanizmy zabezpieczeń Java EE stosowane do ochrony serwisów sieciowych.

Część V: Bezpieczeństwo i tematy zaawansowane

15. Bezpieczeństwo usług sieciowych

Bezpieczeństwo jest nieodłącznym elementem każdego serwisu sieciowego dostępnego przez Internet. Podczas projektowania bezpiecznej usługi należy uwzględnić następujące aspekty:

Poufność (*confidentiality*)
: Implementacja polityki prywatności. Zapewnienie, że wrażliwe dane nie mogą zostać skradzione lub upublicznione.

Spójność (*integrity*)
: Zapewnienie, że przesyłane i przechowywane dane nie zostały zmienione w sposób przypadkowy lub przez osoby trzecie.

Odpowiedzialność i niezaprzeczalność (*accountability, nonrepudiation*)
: Zapewnienie, że partnerzy są odpowiedzialni za swoje akcje. Zalicza się tu również zdolność do wykrywania ataku i przeciwdziałania mu.

Dostępność (*availability*)
: Zapewnienie ciągłego dostępu uwierzytelnionym użytkownikom i ochrona przed awariami lub atakami.

Skala prowadzonych działań i tworzonych zabezpieczeń powinna być adekwatna do ryzyka związanego z działaniem aplikacji.

Podczas budowania wielowarstwowych systemów kwestie bezpieczeństwa implementowane są za pomocą technologii: uwierzytelniania (*authentication*), autoryzacji (*authorization*) i szyfrowania (*cryptography*).

15.1. Wybór mechanizmu bezpieczeństwa — przegląd

Przed przejściem do szczegółów poszczególnych technologii warto wybrać właściwy poziom i mechanizm ochrony na podstawie wymagań scenariusza. Poniższa tabela stanowi przewodnik — szczegółowe omówienie każdego z mechanizmów znajduje się w dalszych sekcjach rozdziału.

Mechanizm	Poziom	Złożoność konfiguracji	Kiedy stosować
HTTPS/TLS	Transport	Niska	Szyfrowanie kanału dla wszystkich serwisów — punkt wyjścia

Mechanizm	Poziom	Złożoność konfiguracji	Kiedy stosować
HTTP Basic	Transport	Niska	Uwierzytelnienie użytkownikiem/hasłem; **wyłącznie** przez HTTPS (hasło w base64 = jawny tekst)
HTTP Digest	Transport	Niska	Jak Basic, lecz hasło nie jest przesyłane wprost; nadal wymaga HTTPS
WS-Security + `UsernameToken`	Wiadomość	Średnia	Gdy wiadomość przechodzi przez pośredników SOAP; token chroniony niezależnie od transportu
WS-Security + X.509	Wiadomość	Wysoka	Podpis cyfrowy i szyfrowanie end-to-end; wymaga infrastruktury PKI
SAML	Federacja / Wiadomość	Wysoka	Single Sign-On między różnymi domenami (SP + IdP); delegacja tożsamości

Kluczowa różnica — bezpieczeństwo transportu vs wiadomości

Bezpieczeństwo transportowe (HTTPS/TLS) chroni kanał komunikacji punkt-punkt. Jeśli wiadomość przechodzi przez pośrednika SOAP (*intermediary*), na węźle pośrednim jest odszyfrowana i ponownie zaszyfrowana — pośrednik ma wgląd w jej treść.

Bezpieczeństwo wiadomości (WS-Security) chroni samą wiadomość — podpisane i zaszyfrowane dane są bezpieczne niezależnie od liczby węzłów pośrednich i niezależnie od protokołu transportowego (HTTP, JMS, SMTP).

W systemach, gdzie wiadomości nie przechodzą przez pośredników SOAP, zazwyczaj wystarczy HTTPS. W systemach z pośrednikami lub wymagających długoterminowego przechowywania podpisanych dokumentów stosuje się WS-Security.

15.2. Zabezpieczenia na poziomie protokołu transportowego

15.2.1. Uwierzytelnianie poprzez HTTP (RFC 2617)

RFC 2617 określa następujące metody przesyłania informacji uwierzytelniających:

- *Basic Access Authentication*
- *Digest Access Authentication*

Gdy użytkownik próbuje uzyskać dostęp do zabezpieczonych treści, serwer odpowiada `HTTP 401 Unauthorized` z nagłówkiem `WWW-Authenticate`.

Basic — hasło + nazwa użytkownika w base64:

```
1 HTTP/1.1 401 Unauthorized
2 WWW-Authenticate: Basic realm="realms"
```

```
1 GET /dir/index.html HTTP/1.1
2 Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

Digest — obliczana suma MD5 (hasło nigdy nie jest przesyłane wprost):

```
1 HTTP/1.1 401 Unauthorized
2 WWW-Authenticate: Digest realm="realms",
3    qop="auth,auth-int",
4    nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
5    opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

```
1 GET /dir/index.html HTTP/1.1
2 Authorization: Digest username="Mufasa",
3    realm="realms",
4    nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
5    uri="/dir/index.html",
6    qop=auth, nc=00000001, cnonce="0a4f113b",
7    response="6629fae49393a05397450978507c4ef1",
8    opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

Digest zabezpiecza przed atakami *replay* i *chosen-plaintext*.

15.2.2. Szyfrowanie SSL/TLS

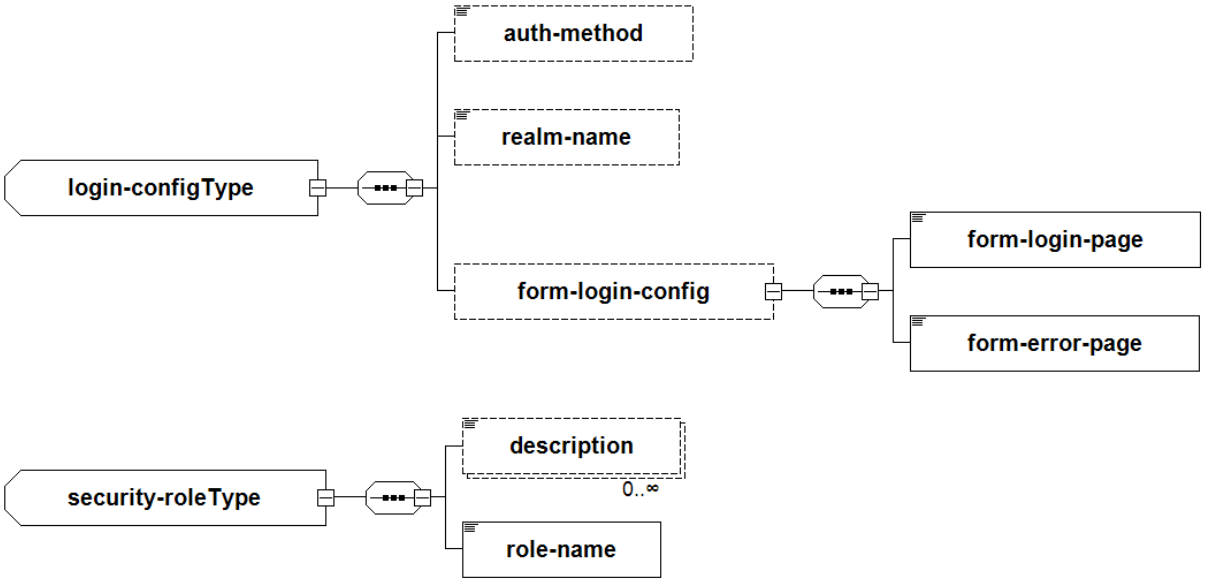
Transport Layer Security (TLS, rozwinięcie SSL) jest powszechnie używanym protokołem do zabezpieczania transmisji danych. Może być używany z HTTP, Telnet, SMTP, FTP. Oparty o algorytmy symetryczne i asymetryczne; certyfikaty umożliwiają uwierzytelnienie serwera (a opcjonalnie klienta).

15.2.3. Uwierzytelnianie i szyfrowanie w Java EE

Konfiguracja w `web.xml` za pomocą elementów: `security-constraint`, `login-config`, `security-role`.

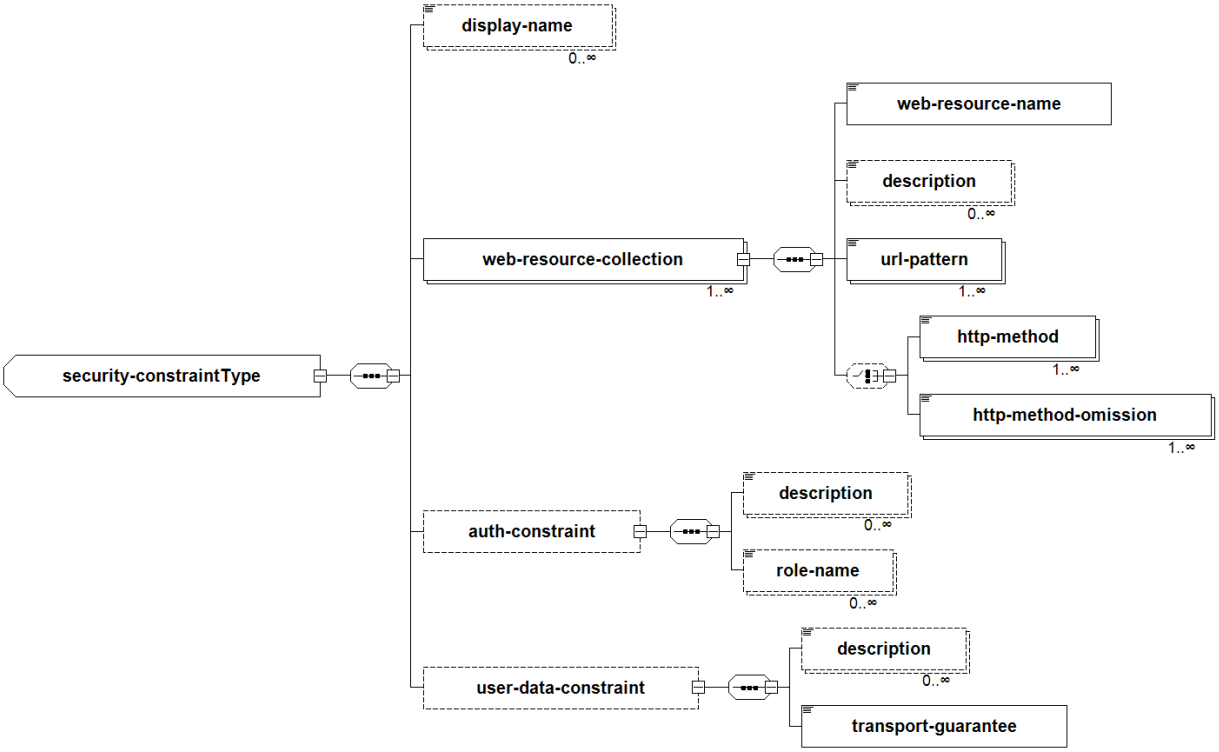
```
<web-app>
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Success</web-resource-name>
            <url-pattern>/welcome.jsp</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>webuser</role-name>
        </auth-constraint>
    </security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>default</realm-name>
    </login-config>
    <security-role>
        <role-name>webuser</role-name>
    </security-role>
</web-app>
```

Wartości `auth-method`: `NONE`, `BASIC`, `DIGEST`, `FORM`, `CLIENT-CERT`.

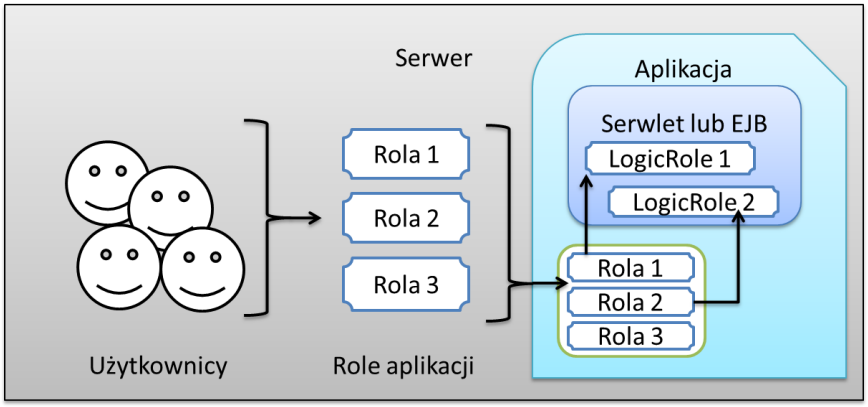


Rysunek 33. Typy login-config

Element `user-data-constraint/transport-guarantee` włącza wymóg szyfrowania: `NONE` (domyślne), `INTEGRAL`, `CONFIDENTIAL` (dwie ostatnie wymuszają TLS).



Rysunek 34. Security constraints



Rysunek 35. Mapowanie ról logicznych

Mapowanie ról logicznych na role serwera:

```
<servlet>
    <servlet-name>ServletName</servlet-name>
    <security-role-ref>
        <role-name>LogicRole</role-name>
        <role-link>Role</role-link>
    </security-role-ref>
</servlet>
```

Odpowiednik w `ejb-jar.xml` dla EJB. Specyfikacja EJB pozwala na deklaratywne kontrolowanie dostępu do pojedynczych metod:

```
@Singleton
@RolesAllowed("Users")
public class Calculator {
    @RolesAllowed("Administrator")
    public void setNewRate(int rate) { ... }

    public int getRate() { ... }
}
```

Adnotacje: `@RolesAllowed`, `@PermitAll`, `@DenyAll`.

15.3. Bezpieczeństwo na poziomie wiadomości

15.3.1. Metody kryptograficzne

15.3.1.1. Rodzaje kluczy kryptograficznych

Klucze symetryczne — do szyfrowania i odszyfrowania używa się jednego klucza. Klucz musi być znany tylko komunikującym się stronom. Przykłady: AES, Triple DES, RC4. Szyfrowanie jest szybkie, ale dystrybucja klucza trudna.

Klucze asymetryczne — dwa różne klucze; wiadomości zakodowane jednym mogą być odszyfrowane tylko drugim. Klucz publiczny służy do szyfrowania, prywatny do odszyfrowania. Przykład w JCA: RSA. Znacznie wolniejsze od symetrycznych.

15.3.1.2. Jednokierunkowe funkcje skrótu

Generują stałej długości skrót z dowolnego wejścia. Słabe (niezalecane): MD5, SHA-1. Zalecane: SHA-2, SHA-3.

15.3.2. XML Security (W3C)

XML-Security jest zbiorem rekomendacji opisujących zastosowanie podpisów cyfrowych i szyfrowania w SOAP:

- *XML Encryption Syntax and Processing* (XMLEnc)
- *XML Signature Syntax and Processing* (XML-DSig)
- *XML Key Management Specification* (XKMS)

Przykładowe implementacje: Apache Santuario, XWS-Security.

15.3.2.1. XML Encryption (XMLEnc)

Szyfrowanie dowolnych danych XML — zaszyfrowany element zastępowany jest `EncryptedData`. Przestrzeń nazw: `http://www.w3.org/2001/04/xmlenc#`

```
<EncryptedData Id? Type? MimeType? Encoding?>
    <EncryptionMethod/>?
    <ds:KeyInfo>
        <EncryptedKey>? | <ds:KeyName>? | ...
    </ds:KeyInfo>?
    <CipherData>
        <CipherValue>?  <!-- lub -->
        <CipherReference URI?>?
    </CipherData>
</EncryptedData>
```

Typ `#Element` — zaszyfrowany jest cały element; typ `#Content` — tylko zawartość.

Szyfrowanie elementu i zawartości

Dokument źródłowy:

```
<PaymentInfo xmlns='http://example.com/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
    <Number>4019 2445 0277 5567</Number>
    <Issuer>Example Bank</Issuer>
    <Expiration>04/14</Expiration>
  </CreditCard>
</PaymentInfo>
```

Szyfrowanie całego elementu `CreditCard` (typ `#Element`):

```
<PaymentInfo xmlns='http://example.com/paymentv2'>
  <Name>John Smith</Name>
  <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
                 xmlns='http://www.w3.org/2001/04/xmlenc#'>
    <CipherData>
      <CipherValue>A23B45C56</CipherValue>
    </CipherData>
  </EncryptedData>
</PaymentInfo>
```

Szyfrowanie zawartości elementu `CreditCard` (typ `#Content`):

```
<PaymentInfo xmlns='http://example.com/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
    <EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
        Type='http://www.w3.org/2001/04/xmlenc#Content'>
      <CipherData>
        <CipherValue>A23B45C56</CipherValue>
      </CipherData>
    </EncryptedData>
  </CreditCard>
</PaymentInfo>
```

Szyfrowanie danych innych niż XML

Gdy atrybut `Type` jest nieobecny lub ma inną wartość niż `#Element`/`#Content`, dane traktowane są jako niebędące XML i przekazywane aplikacji z `MimeType` i `Encoding`. Zaszyfrowane dane wskazuje się przez `CipherReference`:

```
<EncryptedData MimeType="image/png"
    Encoding="http://www.w3.org/2000/09/xmldsig#base64">
  <EncryptionMethod
      Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
  <CipherData>
    <CipherReference URI="cid:bar"/>
  </CipherData>
</EncryptedData>
```

EncryptedKey — zaszyfrowany klucz w KeyInfo

Klucz symetryczny szyfrujący dane jest sam szyfrowany kluczem publicznym odbiorcy i umieszczany w `EncryptedKey` wewnątrz `KeyInfo`:

```
<EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
    Type="http://www.w3.org/2001/04/xmlenc#Element">
  <EncryptionMethod
    Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
  <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <EncryptedKey>
      <EncryptionMethod
        Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
      <ds:KeyInfo>
        <ds:X509Data>
          <ds:X509SubjectName>CN=Michal,O=JavaTech,ST=WA,C=PL</ds:X509SubjectName>
        </ds:X509Data>
      </ds:KeyInfo>
      <CipherData>
        <CipherValue>...</CipherValue>
      </CipherData>
    </EncryptedKey>
  </ds:KeyInfo>
  <CipherData>
    <CipherValue>A23B45C56</CipherValue>
  </CipherData>
</EncryptedData>
```

Wiele zaszyfrowanych elementów — ReferenceList

Gdy wiele elementów jest szyfrowanych tym samym kluczem, stosuje się `ReferenceList` z elementami `DataReference`:

```
<PaymentInfo xmlns="http://example.com/paymentv2">
  <Name>
    <EncryptedData Id="nameInfo" xmlns="http://www.w3.org/2001/04/xmlenc#"
        Type="http://www.w3.org/2001/04/xmlenc#Content">
```

```
      <CipherData><CipherValue>BDERDF2</CipherValue></CipherData>
    </EncryptedData>
  </Name>
  <CreditCard Limit="5,000" Currency="USD">
    <EncryptedData Id="cardInfo" xmlns="http://www.w3.org/2001/04/xmlenc#"
        Type="http://www.w3.org/2001/04/xmlenc#Content">
      <CipherData><CipherValue>A23B45C56</CipherValue></CipherData>
    </EncryptedData>
  </CreditCard>
  <EncryptedKey>
    <EncryptionMethod Algorithm="..."/>
    <CipherData><CipherValue>...</CipherValue></CipherData>
    <ReferenceList>
      <DataReference URI="#nameInfo"/>
      <DataReference URI="#cardInfo"/>
    </ReferenceList>
  </EncryptedKey>
</PaymentInfo>
```

Odwrotna metoda — referencja z elementu zaszyfrowanego do klucza przez `RetrievalMethod` i `CarriedKeyName`:

```
<CreditCard Limit="5,000" Currency="USD">
  <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
      Type="http://www.w3.org/2001/04/xmlenc#Content">
    <ds:KeyInfo xmlns:ds='http://www.w3.org/2000/09/xmldsig#'>
      <ds:RetrievalMethod URI='#EK'
          Type="http://www.w3.org/2001/04/xmlenc#EncryptedKey"/>
      <ds:KeyName>MySuperKey</ds:KeyName>
    </ds:KeyInfo>
    <CipherData><CipherValue>A23B45C56</CipherValue></CipherData>
  </EncryptedData>
</CreditCard>
<EncryptedKey Id="EK">
  <EncryptionMethod Algorithm="..."/>
  <CipherData><CipherValue>...</CipherValue></CipherData>
  <CarriedKeyName>MySuperKey</CarriedKeyName>
</EncryptedKey>
```

Jednoczesne użycie podpisu i szyfrowania

Gdy podpis obejmuje mieszane referencje (część zaszyfrowana, część nie), stosuje się transformację *Decryption Transform for XML Signature* (`http://www.w3.org/2002/07/decrypt#XML`). Nakazuje ona odszyfrowanie elementów przed weryfikacją podpisu; element `Except` wyklucza wskazane bloki (te podpisane już w stanie zaszyfrowanym):

```
<PaymentInfo xmlns="http://example.com/paymentv2"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
    xmlns:dcrpt="http://www.w3.org/2002/07/decrypt#"
    xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
  <Name>
    <xenc:EncryptedData Id="nameInfo"
        Type="http://www.w3.org/2001/04/xmlenc#Content">
      <xenc:CipherData><xenc:CipherValue>BDERDF2</xenc:CipherValue></xenc:CipherData>
    </xenc:EncryptedData>
```

```
  </Name>
  <xenc:CreditCard Limit="5,000" Currency="USD">
    <xenc:EncryptedData Id="cardInfo"
        Type="http://www.w3.org/2001/04/xmlenc#Content">
      <xenc:CipherData><xenc:CipherValue>A23B45C56</xenc:CipherValue></xenc:CipherData>
    </xenc:EncryptedData>
  </xenc:CreditCard>
  <ds:Signature>
    ...
    <ds:Reference URI="">
      <ds:Transforms>
        <ds:Transform
            Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
        <ds:Transform
            Algorithm="http://www.w3.org/2002/07/decrypt#XML">
          <dcrpt:Except URI="#cardInfo"/>
        </ds:Transform>
      </ds:Transforms>
      ...
    </ds:Reference>
    ...
  </ds:Signature>
</PaymentInfo>
```

15.3.2.2. XML Signature (XML-DSig)

Określa reguły podpisywania i weryfikacji fragmentów XML. Przestrzeń nazw: `http://www.w3.org/2000/09/xmldsig#`

```
<Signature Id?>
    <SignedInfo>
        <CanonicalizationMethod/>
        <SignatureMethod/>
        <Reference URI?>+
            <Transforms>?
            <DigestMethod/>
            <DigestValue/>
        </Reference>
    </SignedInfo>
    <SignatureValue/>
    <KeyInfo>?
</Signature>
```

Proces tworzenia podpisu:

1. Zastosowanie transformacji do podpisywanych elementów,
2. obliczenie skrótu po transformacjach i umieszczenie w `DigestValue`,
3. umieszczenie elementów `Reference` w `SignedInfo`,
4. kanonizacja `SignedInfo` i obliczenie jego skrótu,
5. zaszyfrowanie skrótu kluczem prywatnym → `SignatureValue` (base64).

Tworzenie referencji i DigestValue

```
<Reference URI="http://www.example.com/podanie.xml">
  <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
  <DigestValue>60NvZvtdTB+7UnlLp/H24p7h4bs=</DigestValue>
</Reference>
<Reference URI="http://www.example.com/logo.gif">
  <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
  <DigestValue>N/4EN0I1q8BxKAM6GRrCaiClFEU=</DigestValue>
</Reference>
```

Kompletny element Signature z CanonicalizationMethod

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    <CanonicalizationMethod
      Algorithm="http://www.w3.org/2006/12/xml-c14n11"/>
    <SignatureMethod
      Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
    <Reference URI="http://www.example.com/podanie.xml">
      ...
    </Reference>
    <Reference URI="http://www.example.com/logo.gif">
      ...
    </Reference>
  </SignedInfo>
  <SignatureValue>
    hTHQJyd3C6ww/OJz07PqBdznSUOsCpF69w2tln/PFLdx/EP4/VKX
  </SignatureValue>
</Signature>
```

Canonical XML sprowadza dokument XML do postaci normalnej (atrybuty alfabetycznie, jawne przestrzenie nazw, jednolite cudzysłowy), by weryfikacja podpisu była możliwa mimo nieistotnych różnic binarnych. URI algorytmu: `http://www.w3.org/2006/12/xml-c14n11`.

Transformacje w Reference

Element `Transforms` wewnątrz `Reference` określa operacje stosowane przed obliczeniem skrótu. Ostatnią transformacją powinna być normalizacja:

```
<Reference URI="http://www.example.com/podanie.xml">
  <Transforms>
    <Transform Algorithm="http://www.w3.org/2006/12/xml-c14n11"/>
  </Transforms>
  <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
  <DigestValue>60NvZvtdTB+7UnlLp/H24p7h4bs=</DigestValue>
</Reference>
```

Inne zdefiniowane transformacje:

- *Base-64* — koduje dane wejściowe za pomocą base64,
- *XPath Filtering* — wybiera znaczące elementy wyrażeniem XPath,

- *Enveloped Signature Transform* — usuwa element `Signature` z podpisywanego elementu (użycie: `http://www.w3.org/2000/09/xmldsig#enveloped-signature`),
- *XSLT Transform* — stosuje transformację XSLT.

Element KeyInfo

Element `KeyInfo` dostarcza odbiorcy informacji o kluczu potrzebnym do weryfikacji podpisu:

```
<Signature>
  <SignedInfo>...</SignedInfo>
  <SignatureValue>...</SignatureValue>
  <KeyInfo>
    <KeyValue>
      <RSAKeyValue>
        <Modulus>uCiukpgOaOmrq1fPUTH3CAXxuFm....</Modulus>
        <Exponent>AQBB</Exponent>
      </RSAKeyValue>
    </KeyValue>
    <X509Data>
      <X509SubjectName>CN=Michal,O=JavaTech,ST=WA,C=PL</X509SubjectName>
      <X509IssuerSerial>
        <X509IssuerName>CN=Test CA,O=Trust Inc,ST=WA,C=PL</X509IssuerName>
        <X509SerialNumber>167355</X509SerialNumber>
      </X509IssuerSerial>
      <X509Certificate>MIICeDCC3+iDAN....</X509Certificate>
    </X509Data>
  </KeyInfo>
</Signature>
```

Zamiast dołączać klucz bezpośrednio, można podać referencję zewnętrzną:

```
<KeyInfo>
  <RetrievalMethod URI="http://example.com/example-certificate.crt"/>
</KeyInfo>
```

Położenie podpisu

Wyróżniamy trzy typy (odnoszą się do typów referencji, nie samego podpisu):

- *enveloped* — podpis jest potomkiem podpisywanego elementu,
- *enveloping* — podpis otacza podpisywany element (umieszczony w `Object`),
- *detached* — podpis oddzielony od podpisywanych danych.

Przykład podpisu typu *enveloping*:

```
<Signature>
  <SignedInfo>
    <Reference URI="#MyOrder">
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
```

```
      <DigestValue>...</DigestValue>
    </Reference>
  </SignedInfo>
  <Object Id="MyOrder">
    <Order>
      <LineItem>Web Service Java 6</LineItem>
      <LineItem>Very Funny Book</LineItem>
    </Order>
  </Object>
</Signature>
```

Ogólny schemat podpisu XML-DSig

```
<Signature ID?>              // główny element
  <SignedInfo>               // metadane i lista podpisanych elementów
    <CanonicalizationMethod/>// algorytm kanonizacji użyty przy obliczaniu podpisu
    <SignatureMethod/>       // algorytm haszowania i szyfrowania
    (<Reference URI?>        // wskazuje konkretny podpisany element
      (<Transforms>)?        // opcjonalne transformacje przed obliczeniem skrótu
      <DigestMethod>         // algorytm haszowania
      <DigestValue>          // wartość skrótu zakodowana w base64
    </Reference>)+
  </SignedInfo>
  <SignatureValue>           // wartość podpisu zaszyfrowana kluczem prywatnym
  (<KeyInfo>)?               // klucz do weryfikacji lub wskazanie skąd go pobrać
  (<Object ID?>)*            // element pozwalający na rozszerzenia (enveloping)
</Signature>
```

15.3.2.3. XML Key Management Specification (XKMS)

Definiuje dystrybucję i rejestrowanie kluczy publicznych:

- **X-KISS** — protokół do odnajdywania i weryfikowania kluczy z `KeyInfo`,
- **X-KRSS** — protokół do zarządzania i rejestrowania kluczy.

15.3.3. OASIS Web Services Security (WSS)

WSS jest zestawem zasad opisujących sposoby implementacji bezpieczeństwa w SOAP:

- *SOAP Message Security* (WS-Security)
- *UsernameToken Profile*
- *X.509 Certificate Token Profile*
- *SAML Token Profile*
- *Kerberos Token Profile*
- *SOAP Messages with Attachments (SwA) Profile*

15.3.3.1. SOAP Message Security (WS-Security)

Większość informacji WSS umieszczona jest w nagłówku `Security`:

```
<S11:Envelope xmlns:S11="..."
    xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/...">
    <S11:Header>
        <wsse:Security>
            <wsse:UsernameToken>
                <wsse:Username>Zoe</wsse:Username>
            </wsse:UsernameToken>
        </wsse:Security>
    </S11:Header>
    ...
</S11:Envelope>
```

15.3.3.2. UsernameToken Profile

Definiuje element `UsernameToken` do przesyłania danych logowania:

```
<wsse:Security>
    <wsse:UsernameToken>
        <wsse:Username>NNK</wsse:Username>
        <wsse:Password Type="...#PasswordDigest">
            weYI3nXd8LjMNVksCKFV8t3rgHh3Rw==
        </wsse:Password>
        <wsse:Nonce>WScqanjCEAC4mQoBE07sAQ==</wsse:Nonce>
        <wsu:Created>2003-07-16T01:24:32Z</wsu:Created>
    </wsse:UsernameToken>
</wsse:Security>
```

Do generowania kluczy szyfrujących używany jest mechanizm iterowanego SHA1:

- K_1 = SHA1(password + Salt)
- K_2 = SHA1(K_1)
- ...
- K_N = SHA1(K_{N-1})

15.3.3.3. X.509 Certificate Token Profile

Specyfikacja umożliwia używanie certyfikatów X.509 do szyfrowania, podpisywania i uwierzytelniania. Element `BinarySecurityToken` transportuje certyfikat; atrybut `ValueType` wskazuje na X.509v3:

```
<wsse:BinarySecurityToken wsu:Id="binarytoken"
  ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3"
  EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-
1.0#Base64Binary">
  MIIEZzCCA9CgAwIBAgIQEmtJZc0...
</wsse:BinarySecurityToken>
```

Referencja do tokenu przez `SecurityTokenReference` (np. w `KeyInfo` podpisu):

```
<ds:KeyInfo>
  <wsse:SecurityTokenReference>
    <wsse:Reference URI="#binarytoken"/>
  </wsse:SecurityTokenReference>
</ds:KeyInfo>
```

Certyfikat nie musi być dołączany do wiadomości — referencja może bazować na danych certyfikatu (`X509IssuerSerial`):

```
<ds:KeyInfo>
  <wsse:SecurityTokenReference>
    <ds:X509Data>
      <ds:X509IssuerSerial>
        <ds:X509IssuerName>O=Javatech</ds:X509IssuerName>
        <ds:X509SerialNumber>12345678</ds:X509SerialNumber>
      </ds:X509IssuerSerial>
    </ds:X509Data>
  </wsse:SecurityTokenReference>
</ds:KeyInfo>
```

Identyfikacja przez skrót certyfikatu (*thumbprint*) za pomocą `KeyIdentifier`:

```
<wsse:SecurityTokenReference>
  <wsse:KeyIdentifier
    ValueType="http://docs.oasis-open.org/wss/oasis-wss-soap-message-security-1.1#ThumbprintSHA1">
    LKiQ/CmFrJDJqCLFcjlhIsmZ/+0=
  </wsse:KeyIdentifier>
</wsse:SecurityTokenReference>
```

15.3.3.4. SOAP Messages with Attachments (SwA) Profile

Rekomendacja WSS określa metody zabezpieczeń wiadomości z załącznikami (SwA). Definiuje transformacje kanoniczne dla części MIME:

- tylko treść (`Attachment-Content-Signature-Transform`),
- treść + wybrane nagłówki (`Attachment-Complete-Signature-Transform`),
- zaszyfrowana treść (`Attachment-Ciphertext-Transform`) — głównie do podpisywania.

Przykład — podpisany załącznik

```
Content-Type: multipart/related; boundary="BoundaryStr" type="text/xml"

--BoundaryStr
Content-Type: text/xml
<S11:Envelope xmlns:S11="http://schemas.xmlsoap.org/soap/envelope/"
              xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
              xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
              xmlns:ds="http://www.w3.org/2000/09/xmldsig1#">
```

```
  <S11:Header>
    <wsse:Security>
      <wsse:BinarySecurityToken
        wsu:Id="CertAssociatedWithSigningKey"
        EncodingType="...#Base64Binary"
        ValueType="...#x509v3">
        ...
      </wsse:BinarySecurityToken>
      <ds:Signature>
        <ds:SignedInfo>
          <ds:CanonicalizationMethod
           Algorithm='http://www.w3.org/2001/10/xml-exc-c14n#'/>
          <ds:SignatureMethod
           Algorithm='http://www.w3.org/2000/09/xmldsig#rsa-sha1'/>
          <ds:Reference URI="cid:bar">
            <ds:Transforms>
              <ds:Transform
                Algorithm="...#Attachment-Content-Signature-Transform"/>
            </ds:Transforms>
            <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
            <ds:DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>6lwx3r0vKtMup4Nbe</ds:SignatureValue>
        <ds:KeyInfo>
          <wsse:SecurityTokenReference>
            <wsse:Reference URI="#CertAssociatedWithSigningKey"/>
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
      </ds:Signature>
    </wsse:Security>
  </S11:Header>
  <S11:Body>some items</S11:Body>
</S11:Envelope>
--BoundaryStr
Content-Type: image/png
Content-ID: <bar>
Content-Transfer-Encoding: base64
the image
```

Przykład — zaszyfrowane ciało i załącznik

`EncryptedKey` z `ReferenceList` wskazuje na zaszyfrowany załącznik (`#EA`, przez `CipherReference`) i zaszyfrowane ciało (`#ED`):

```
Content-Type: multipart/related; boundary="BoundaryStr" type="text/xml"
--BoundaryStr
Content-Type: text/xml
<S11:Envelope xmlns:S11="http://schemas.xmlsoap.org/soap/envelope/"
              xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-
1.0.xsd"
              xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
              xmlns:ds="http://www.w3.org/2000/09/xmlsig1#">
  <S11:Header>
    <wsse:Security>
      <xenc:EncryptedKey Id='EK'>
        <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
        <ds:KeyInfo Id="keyinfo">
          <wsse:SecurityTokenReference>...</wsse:SecurityTokenReference>
```

```
14         </ds:KeyInfo>
15         <CipherData><CipherValue>xyzabc</CipherValue></CipherData>
16         <ReferenceList>
17           <DataReference URI='#EA'/>
18           <DataReference URI='#ED'/>
19         </ReferenceList>
20       </xenc:EncryptedKey>
21       <xenc:EncryptedData Id='EA'
22         Type="http://docs.oasis-open.org/wss/oasis-wss-SwAProfile-1.1#Attachment-Content-Only"
23         MimeType="image/png">
24         <xenc:EncryptionMethod Algorithm='http://www.w3.org/2001/04/xmlenc#aes128-cbc'/>
25         <xenc:CipherData>
26           <xenc:CipherReference URI="cid:bar">
27             <xenc:Transforms>
28               <ds:Transform Algorithm="...#Attachment-Ciphertext-Transform"/>
29             </xenc:Transforms>
30           </xenc:CipherReference>
31         </xenc:CipherData>
32       </xenc:EncryptedData>
33     </wsse:Security>
34   </S11:Header>
35   <S11:Body>
36     <xenc:EncryptedData Id='ED'>
37       <xenc:EncryptionMethod Algorithm='http://www.w3.org/2001/04/xmlenc#aes128-cbc'/>
38       <xenc:CipherData>
39         <xenc:CipherValue>6lwx3rvEPO0vKtMup4Nbe</xenc:CipherValue>
40       </xenc:CipherData>
41     </xenc:EncryptedData>
42   </S11:Body>
43 </S11:Envelope>
44 --BoundaryStr
45 Content-Type: application/octet-stream
46 Content-ID: <bar>
47 Content-Transfer-Encoding: binary
48     BinaryCipherData
```

15.3.4. Security Assertion Markup Language (SAML)

Standard przeznaczony do wymiany informacji uwierzytelniających i autoryzacyjnych. U jego podstaw leży możliwość realizacji dwóch scenariuszy:

***Single Sign-On* (SSO)**

Pozwala na scentralizowanie serwisów uwierzytelniających. Dzięki SSO dostawcy usług nie muszą tworzyć i przechowywać baz danych z danymi uwierzytelniającymi, natomiast użytkownicy nie muszą pamiętać wielu loginów i haseł. Użytkownik loguje się raz do serwisu uwierzytelniającego — dostawcy tożsamości (IdP) — a następnie, chcąc skorzystać z dowolnej usługi, przesyła asercję wydaną przez IdP, na podstawie której serwis identyfikuje użytkownika i tworzy dla niego sesję.

Identity Federation

Oznacza porozumienie pomiędzy dostawcami usług odnośnie tożsamości użytkownika. Dzięki temu mogą oni komunikować się między sobą i wymieniać informacje na temat użytkownika.

15.3.4.1. Części składowe specyfikacji

Assertions

Opisują twierdzenia na temat tożsamości użytkownika, jego właściwości i uprawnień. Asercje mogą służyć uwierzytelnianiu, przekazywaniu właściwości (np. stanowisko, status VIP) oraz autoryzacji (np. czy użytkownik ma prawo do danego zasobu).

Protocols

Łączą zestawy asercji w użyteczne jednostki realizujące konkretne operacje: żądanie uwierzytelnienia z odpowiednią asercją, pojedyncze wylogowanie (*Single Logout*) łączące żądanie wylogowania z serwisu i propagację tego żądania do innych uczestników.

Bindings

Określają sposób transportowania wiadomości SAML między uczestnikami. Przykłady: *HTTP Redirect* (najpopularniejszy), *SAML SOAP*, *HTTP POST*.

Profiles

Łączą asercje, protokoły i wiązania w gotowe rozwiązania, np. *Web Browser SSO Profile*, *Single Logout Profile*.

Metadata* i *Authentication Context

Opisują konfigurację uczestników i kontekst uwierzytelnienia (metoda, siła zabezpieczeń).

Przestrzenie nazw SAML 2.0: `urn:oasis:names:tc:SAML:2.0:assertion` oraz `urn:oasis:names:tc:SAML:2.0:protocol`.

15.3.4.2. Przykład asercji uwierzytelniającej

```
<saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
                Version="2.0"
                IssueInstant="2014-01-31T12:00:00Z">
  <saml:Issuer Format="urn:oasis:names:SAML:2.0:nameid-format:entity">
    http://idp.example.com
  </saml:Issuer>
  <saml:Subject>
    <saml:NameID
      Format="urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress">
      j.doe@example.com
    </saml:NameID>
  </saml:Subject>
  <saml:Conditions
    NotBefore="2014-01-31T12:00:00Z"
    NotOnOrAfter="2014-01-31T12:10:00Z">
    <saml:AudienceRestriction>
      <saml:Audience>https://sp.example.com/SAML2</saml:Audience>
    </saml:AudienceRestriction>
  </saml:Conditions>
  <saml:AuthnStatement
```

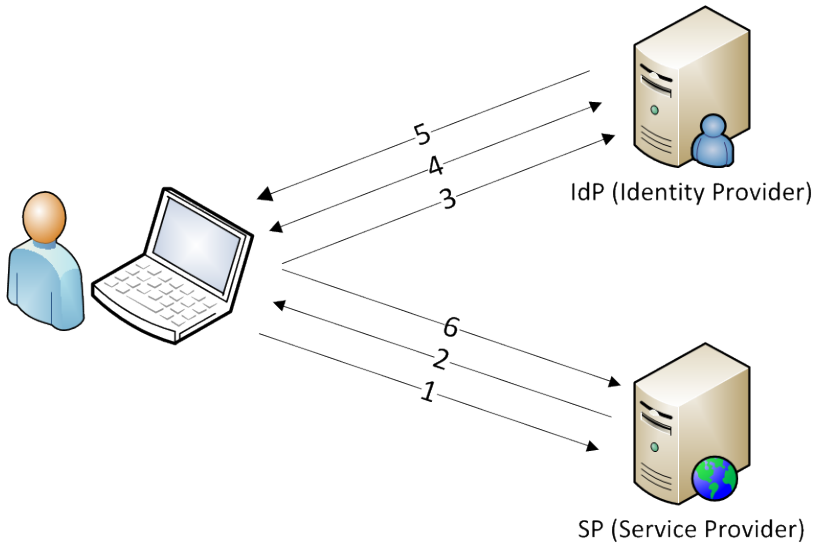
```
    AuthnInstant="2014-01-31T12:00:00Z" SessionIndex="67775277772">
    <saml:AuthnContext>
      <saml:AuthnContextClassRef>
        urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport
      </saml:AuthnContextClassRef>
    </saml:AuthnContext>
  </saml:AuthnStatement>
</saml:Assertion>
```

Asercja zawiera identyfikację wystawcy (`Issuer` — `http://idp.example.com`), który twierdzi, że użytkownik identyfikowany adresem email `j.doe@example.com` został uwierzytelniony za pomocą hasła przesłanego po SSL. Asercja jest ważna 10 minut (`Conditions`) i może być użyta wyłącznie przez `https://sp.example.com/SAML2`.

Asercja ta mogła zostać wydana w odpowiedzi na następujące żądanie:

```
<samlp:AuthnRequest
  xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
  xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
  ID="identifier_1"
  Version="2.0"
  IssueInstant="2014-01-31T11:55:00Z"
  AssertionConsumerServiceIndex="1">
  <saml:Issuer>https://sp.example.com/SAML2</saml:Issuer>
  <samlp:NameIDPolicy
    AllowCreate="true"
    Format="urn:oasis:names:tc:SAML:2.0:nameid-format:transient"/>
</samlp:AuthnRequest>
```

15.3.4.3. Web Browser SSO Profile



Rysunek 36. SAML Web Browser SSO

Przebieg Web Browser SSO:

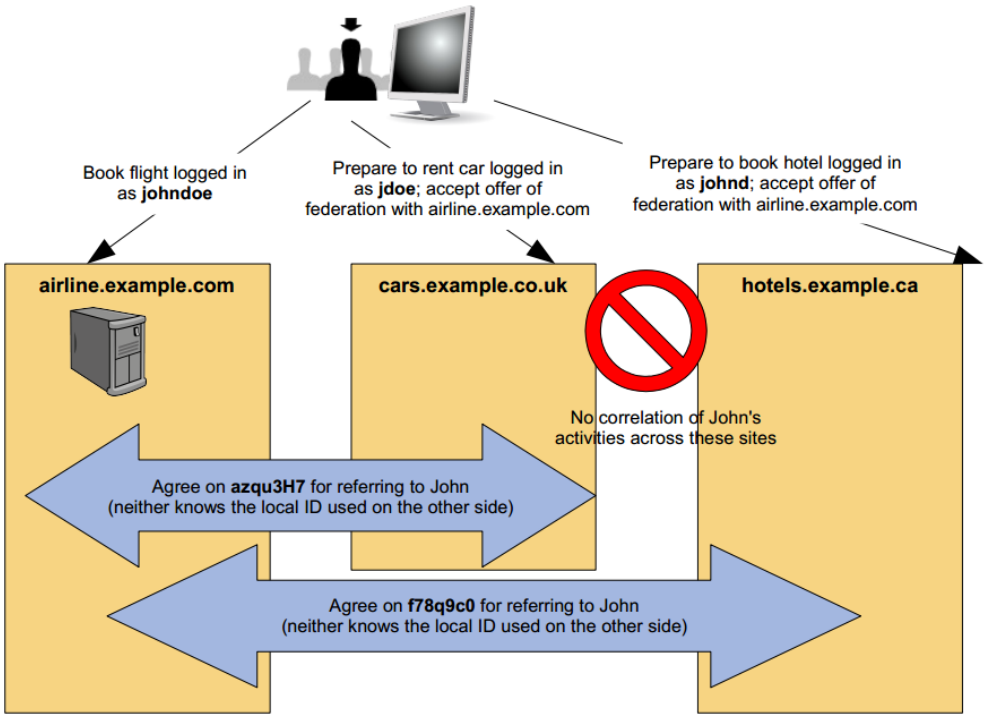
1. Użytkownik wykonuje żądanie do dostawcy usługi (SP),

2. SP odmawia dostępu i żąda uwierzytelnienia u zarządcy tożsamości (IdP),
3. użytkownik wysyła żądanie do IdP,
4. klient loguje się na serwerze IdP,
5. IdP odsyła podpisaną odpowiedź uwierzytelniającą,
6. klient przekazuje ją do SP.

Zarówno żądanie uwierzytelnienia jak i odpowiedź od IdP są podpisane cyfrowo — klient nie może ich podrobić.

15.3.4.4. Identity Federation Profile

W federacji każdy serwis może pełnić rolę dostawcy tożsamości. Przykładowy scenariusz federacji między trzema serwisami:



Rysunek 37. Federacja tożsamości SAML

1. Użytkownik loguje się do serwisu `airline.example.com` jako `johndoe`.
2. Korzystając z SSO, używając `airline.example.com` jako dostawcy tożsamości, loguje się w serwisie `cars.example.co.uk` jako `jdoe`.
3. Serwis `cars.example.co.uk` łączy identyfikator przesłany przez `airline.example.com` z nowo utworzonym kontem `jdoe`.
4. Od tej chwili oba serwisy mogą wymieniać informacje o aktywności użytkownika.
5. W ten sam sposób użytkownik może założyć konto w serwisie `hotels.example.ca`, jednak z innym identyfikatorem pseudonimowym.
6. Tylko serwis `airline.example.com` wie, że użytkownik identyfikowany przez `azqu3H7` jest tożsamy z `f78q9c0` — pseudonimy w poszczególnych serwisach są od siebie niezależne.

15.4. WS-SecurityPolicy

Mnogość rodzajów zabezpieczeń możliwych do zastosowania w komunikacji SOAP skutkuje koniecznością posiadania mechanizmu opisu wymagań. Taki opis dostarcza specyfikacja *WS-SecurityPolicy* (OASIS). Pozwala ona określić każdy aspekt użytych zabezpieczeń: wymagania dotyczące protokołu transportowego, użyte algorytmy, wymagane elementy koperty SOAP.

Specyfikacja wyróżnia trzy podstawowe grupy asercji:

- *protection assertions* — wskazują, które części wiadomości są chronione i jak (szyfrowanie i podpisywanie),
- *token assertions* — wymuszają użycie konkretnych tokenów nagłówka `Security`,
- *security binding assertions* — opisują właściwości komunikacji, podzielone ze względu na metodę zapewnienia poufności.

15.4.1. Protection Assertions

Ten typ asercji służy do wskazania, które elementy koperty mają być szyfrowane lub podpisane. Specyfikacja zezwala na dołączanie ich tylko do elementów WSDL: `wsdl:binding/wsdl:operation/wsdl:input`, `⋯/output`, `⋯/fault`.

15.4.1.1. Dołączanie asercji — PolicyReference

```
<!-- plik dostępny pod adresem http://www.example.com/policies.xml -->
<wsp:Policy wsu:Id="BodySignature"
            xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
            xmlns:wsp="http://www.w3.org/ns/ws-policy"
            xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SignedParts>
        <sp:Body/>
      </sp:SignedParts>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Łączenie z konkretną wiadomością za pomocą `PolicyReference`:

```
<operation name="OperationName"
    xmlns:wsp="http://www.w3.org/ns/ws-policy">
  <operation soapAction="..."/>
  <input>
    <body use="literal"/>
    <wsp:PolicyReference
        URI="http://www.example.com/policies.xml#BodySignature"
        required="true"/>
  </input>
```

```
10   <output>
11     <body use="literal"/>
12     <wsp:PolicyReference
13         URI="http://www.example.com/policies.xml#BodySignature"
14         required="true"/>
15   </output>
16 </operation>
```

Atrybut `required` (z przestrzeni WSDL) oznacza, że element musi być rozumiany i wspierany przez klienta — nie że musi być przez niego używany.

15.4.1.2. Integrity Assertions

```
1 <sp:SignedParts>
2   <sp:Body/>
3   <sp:Header Name="xs:NCName"? Namespace="xs:anyURI"/>*
4 </sp:SignedParts>
```

lub (XPath do konkretnych elementów):

```
1 <sp:SignedElements XPathVersion="xs:anyURI"?>
2   <sp:XPath>xs:string</sp:XPath>+
3 </sp:SignedElements>
```

Pusty `SignedParts` oznacza podpisanie wszystkich nagłówków kierowanych do końcowego odbiorcy i ciała wiadomości.

15.4.1.3. Confidentiality Assertions

```
1 <sp:EncryptedParts>
2   <sp:Body/>
3   <sp:Header Name="xs:NCName"? Namespace="xs:anyURI"/>*
4 </sp:EncryptedParts>
```

lub:

```
1 <sp:EncryptedElements XPathVersion="xs:anyURI"?>
2   <sp:XPath>xs:string</sp:XPath>+
3 </sp:EncryptedElements>
```

Pusty `EncryptedParts` oznacza szyfrowanie tylko ciała wiadomości.

15.4.1.4. Required Elements Assertion

Wskazuje elementy, które muszą się pojawić w wiadomości:

```
<sp:RequiredElements XPathVersion="xs:anyURI"?>
  <sp:XPath>xs:string</sp:XPath>+
</sp:RequiredElements>
```

15.4.2. Token Assertions

Tokeny są nośnikami informacji odnośnie zabezpieczeń. Każda asercja tokenu może mieć atrybut `IncludeToken` (URI z wartością):

- `…/IncludeToken/Never` — token nie może być przesyłany w wiadomości,
- `…/IncludeToken/Once` — token dołączany tylko raz; później dopuszczalne zewnętrzne referencje,
- `…/IncludeToken/AlwaysToRecipient` — token zawsze dołączany przez inicjatora (klienta),
- `…/IncludeToken/AlwaysToInitiator` — token zawsze dołączany przez serwer,
- `…/IncludeToken/Always` — domyślne; token dołączany do każdej wiadomości.

15.4.2.1. UsernameToken

```
<sp:UsernameToken sp:IncludeToken="xs:anyURI"? xmlns:sp="...">
  <wsp:Policy xmlns:wsp="...">
    ( <sp:NoPassword/> | <sp:HashPassword/> )?
    ( <sp:WssUsernameToken10/> | <sp:WssUsernameToken11/> )?
  </wsp:Policy>
</sp:UsernameToken>
```

15.4.2.2. X509Token

```
<sp:X509Token sp:IncludeToken="xs:anyURI"? xmlns:sp="...">
  <wsp:Policy xmlns:wsp="...">
    <sp:RequireKeyIdentifierReference/> ?
    <sp:RequireIssuerSerialReference/> ?
    <sp:RequireEmbeddedTokenReference/> ?
    <sp:RequireThumbprintReference/> ?
    ( <sp:WssX509V3Token10/> | <sp:WssX509Pkcs7Token10/> |
      <sp:WssX509V1Token11/> | <sp:WssX509V3Token11/> | ... )?
  </wsp:Policy>
</sp:X509Token>
```

Tokeny opakowuje się elementami:

- `SupportingTokens` — tokeny umieszczane w nagłówku `Security` bez dodatkowych zabezpieczeń,
- `SignedSupportingTokens` — tokeny automatycznie włączane do podpisu.

15.4.3. Security Binding Assertions

- `TransportBinding` — zabezpieczenia dostarczane przez protokół transportowy,
- `SymmetricBinding` — klucze symetryczne do szyfrowania (wymiana kluczy przez SSL lub klucz asymetryczny),
- `AsymmetricBinding` — obie strony posiadają certyfikaty; klucz asymetryczny do podpisu i szyfrowania.

15.4.3.1. TransportBinding

```
<wsp:Policy xmlns:wsp="http://www.w3.org/ns/ws-policy"
            xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
  <sp:TransportBinding>
    <wsp:Policy>
      <sp:TransportToken>
        <wsp:Policy>
          <sp:HttpsToken>
            <wsp:Policy>
              <sp:HttpBasicAuthentication/>
            </wsp:Policy>
          </sp:HttpsToken>
        </wsp:Policy>
      </sp:TransportToken>
      <sp:AlgorithmSuite>
        <wsp:Policy>
          <sp:Basic256/>
        </wsp:Policy>
      </sp:AlgorithmSuite>
    </wsp:Policy>
  </sp:TransportBinding>
</wsp:Policy>
```

`HttpsToken` włącza HTTPS; `AlgorithmSuite` `Basic256` — AES do szyfrowania symetrycznego, RSA do asymetrycznego.

15.4.3.2. SymmetricBinding — przykład

Wiązanie symetryczne z certyfikatem X.509 i tokenem `Username` w roli dodatkowego podpisu:

```
<wsp:Policy wsu:Id="WSS11UsernameWithCertificates_policy"
            xmlns:wsp="http://www.w3.org/ns/ws-policy"
            xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SymmetricBinding>
        <wsp:Policy>
          <sp:ProtectionToken>
            <wsp:Policy>
              <sp:X509Token
                sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/Never">
                <wsp:Policy>
```

```
                <sp:RequireThumbprintReference/>
                <sp:WssX509V3Token11/>
              </wsp:Policy>
            </sp:X509Token>
          </wsp:Policy>
        </sp:ProtectionToken>
        <sp:AlgorithmSuite>
          <wsp:Policy><sp:Basic256/></wsp:Policy>
        </sp:AlgorithmSuite>
        <sp:Layout>
          <wsp:Policy><sp:Strict/></wsp:Policy>
        </sp:Layout>
        <sp:IncludeTimestamp/>
        <sp:OnlySignEntireHeadersAndBody/>
      </wsp:Policy>
    </sp:SymmetricBinding>
    <sp:SignedSupportingTokens>
      <wsp:Policy>
        <sp:UsernameToken sp:IncludeToken="AlwaysToRecipient">
          <wsp:Policy><sp:WssUsernameToken11/></wsp:Policy>
        </sp:UsernameToken>
      </wsp:Policy>
    </sp:SignedSupportingTokens>
  </wsp:All>
 </wsp:ExactlyOne>
</wsp:Policy>

<wsp:Policy wsu:Id="UsernameForCertificates_message_policy">
 <wsp:ExactlyOne>
  <wsp:All>
   <sp:SignedParts><sp:Body/></sp:SignedParts>
   <sp:EncryptedParts><sp:Body/></sp:EncryptedParts>
  </wsp:All>
 </wsp:ExactlyOne>
</wsp:Policy>
```

Przykładowa wiadomość zgodna z tą polityką (certyfikat rozpoznawany przez `ThumbprintSHA1`, `UsernameToken` włączony w podpis, ciało zaszyfrowane):

```
<soap:Envelope xmlns:soap="..." xmlns:ds="..." xmlns:xenc="..."
               xmlns:wsse="..." xmlns:wsu="...">
 <soap:Header>
  <wsse:Security soap:mustUnderstand="1">
   <xenc:ReferenceList>
    <xenc:DataReference URI="#encBody"/>
   </xenc:ReferenceList>
   <wsu:Timestamp wsu:Id="T0">
    <wsu:Created>2014-01-13T08:42:00Z</wsu:Created>
   </wsu:Timestamp>
   <wsse:UsernameToken wsu:Id="usernameToken">
    <wsse:Username>Michal</wsse:Username>
    <wsse:Password Type="wsse:PasswordText">pass</wsse:Password>
   </wsse:UsernameToken>
   <ds:Signature>
    <ds:SignedInfo>
     <ds:Reference URI="#T0">...</ds:Reference>
     <ds:Reference URI="#usernameToken">...</ds:Reference>
     <ds:Reference URI="#body">...</ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>HFLP...</ds:SignatureValue>
```

```
      <ds:KeyInfo>
        <wsse:SecurityTokenReference>
          <wsse:KeyIdentifier
            ValueType="...#ThumbPrintSHA1">
            LKiQ/CmFrJDJqCLFcjlhIsmZ/+0=
          </wsse:KeyIdentifier>
        </wsse:SecurityTokenReference>
      </ds:KeyInfo>
    </ds:Signature>
  </wsse:Security>
</soap:Header>
<soap:Body wsu:Id="body">
  <xenc:EncryptedData wsu:Id="encBody">
    <ds:KeyInfo>
      <wsse:KeyIdentifier ValueType="...#ThumbPrintSHA1">
        LKiQ/CmFrJDJqCLFcjlhIsmZ/+0=
      </wsse:KeyIdentifier>
    </ds:KeyInfo>
    <xenc:CipherData>
      <xenc:CipherValue>...</xenc:CipherValue>
    </xenc:CipherData>
  </xenc:EncryptedData>
</soap:Body>
</soap:Envelope>
```

15.4.3.3. AsymmetricBinding — przykład

Obie strony posiadają certyfikaty (`InitiatorToken` zawsze dołączany, `RecipientToken` nigdy — rozpoznawany z kontekstu):

```
<wsp:Policy wsu:Id="with_cert_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:AsymmetricBinding>
        <wsp:Policy>
          <sp:InitiatorToken>
            <wsp:Policy>
              <sp:X509Token sp:IncludeToken=".../IncludeToken/AlwaysToRecpt">
                <wsp:Policy><sp:WssX509V3Token10/></wsp:Policy>
              </sp:X509Token>
            </wsp:Policy>
          </sp:InitiatorToken>
          <sp:RecipientToken>
            <wsp:Policy>
              <sp:X509Token sp:IncludeToken=".../IncludeToken/Never">
                <wsp:Policy><sp:WssX509V3Token10/></wsp:Policy>
              </sp:X509Token>
            </wsp:Policy>
          </sp:RecipientToken>
          <sp:AlgorithmSuite>
            <wsp:Policy><sp:Basic256/></wsp:Policy>
          </sp:AlgorithmSuite>
          <sp:Layout>
            <wsp:Policy><sp:Strict/></wsp:Policy>
          </sp:Layout>
          <sp:IncludeTimestamp/>
          <sp:OnlySignEntireHeadersAndBody/>
        </wsp:Policy>
      </sp:AsymmetricBinding>
```

```
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>

<wsp:Policy wsu:Id="Certificates_message_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SignedParts><sp:Body/></sp:SignedParts>
      <sp:EncryptedParts><sp:Body/></sp:EncryptedParts>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Przykładowa wiadomość (ciało podpisane kluczem inicjatora i zaszyfrowane kluczem odbiorcy):

```
<soap:Envelope xmlns:soap="..." xmlns:xenc="..." xmlns:ds="...">
  <soap:Header>
    <wsse:Security soap:mustUnderstand="1" xmlns:wsse="..." xmlns:wsu="...">
      <xenc:ReferenceList>
        <xenc:DataReference URI="#encBody"/>
      </xenc:ReferenceList>
      <wsu:Timestamp wsu:Id="T0">
        <wsu:Created>2014-01-13T08:42:00Z</wsu:Created>
      </wsu:Timestamp>
      <wsse:BinarySecurityToken wsu:Id="binaryToken"
          ValueType="...#X509v3"
          EncodingType="...#Base64Binary">
        MIIEZzCCA9CgAwIBAgIQEmtJZc0...
      </wsse:BinarySecurityToken>
      <ds:Signature>
        <ds:SignedInfo>
          <ds:Reference URI="#T0">...</ds:Reference>
          <ds:Reference URI="#body">...</ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>HFLP...</ds:SignatureValue>
        <ds:KeyInfo>
          <wsse:SecurityTokenReference>
            <wsse:Reference URI="#binaryToken"/>
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
      </ds:Signature>
    </wsse:Security>
  </soap:Header>
  <soap:Body wsu:Id="body">
    <xenc:EncryptedData wsu:Id="encBody">
      <ds:KeyInfo>
        <wsse:KeyIdentifier EncodingType="...#Base64Binary"
          ValueType="...#X509SubjectKeyIdentifier">
          MIGfMa0GCSq...
        </wsse:KeyIdentifier>
      </ds:KeyInfo>
      <xenc:CipherData>
        <xenc:CipherValue>...</xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </soap:Body>
</soap:Envelope>
```

15.5. Wybór mechanizmu bezpieczeństwa

Mechanizm	Poziom	Kiedy stosować
HTTPS/TLS	Transport	Większość serwisów — szyfrowanie kanału; najprostsze w konfiguracji
HTTP Basic/Digest	Transport	Uwierzytelnienie klienta; **wyłącznie przez HTTPS**
WS-Security + UsernameToken	Wiadomość	Gdy wiadomość przechodzi przez pośredników SOAP
WS-Security + X.509	Wiadomość	Podpis cyfrowy i szyfrowanie end-to-end; wymaga PKI
SAML	Federacja	Single Sign-On między różnymi domenami (SP + IdP)

Bezpieczeństwo na poziomie transportu (HTTPS) chroni kanał komunikacji, ale nie zabezpiecza wiadomości gdy przechodzi przez pośredników SOAP (*intermediaries*). Bezpieczeństwo na poziomie wiadomości (WS-Security) jest niezależne od transportu — podpisane i zaszyfrowane dane są bezpieczne niezależnie od liczby węzłów pośrednich.

Praktyczna konfiguracja WS-Security dla serwisów JAX-WS realizowana jest przez WSIT (Metro) — opisane w kolejnym rozdziale.

16. WSIT — Web Services Interoperability Technology

Specyfikacje WS-Security i pokrewne opisują *co* powinno być zapewnione. WSIT (Metro) dostarcza konkretną implementację tych standardów dla platformy Java/GlassFish. WSIT jest szczególnie istotny dla interoperacyjności z platformą .NET: zarówno Java EE (przez Metro), jak i .NET (przez WCF) implementują te same standardy WS-*, co umożliwia budowanie systemów heterogenicznych bez niestandardowych rozszerzeń.

WSIT jest projektem open-source zapoczątkowanym przez *Sun Microsystems* pod nazwą *Tango*. Jest częścią projektu *Glassfish Metro*. Podstawowym celem projektu było stworzenie implementacji najczęściej wykorzystywanych rekomendacji WS-*, tak aby zapewnić pełną interoperacyjność z usługami napisanymi w platformie .NET (Windows Communication Foundation — WCF). Projekt bazuje na XWSS (*XML and Web Services Security*).

W ramach WSIT zaimplementowane zostały następujące specyfikacje:

WS-Trust

Protokół wymiany i walidacji tokenów bezpieczeństwa pomiędzy usługą a wystawcą tokenów (STS — *Security Token Service*). Stanowi podstawę federacji tożsamości między różnymi domenami zaufania.

WS-SecureConversation

Rozszerzenie WS-Trust definiujące mechanizm ustanowienia *Security Context Token* (SCT) — tokenu sesji, dzięki któremu kolejne wiadomości w ramach jednej sesji mogą używać lżejszego szyfrowania symetrycznego zamiast każdorazowej wymiany certyfikatów.

WS-SecurityPolicy

Język polityk (oparty na WS-Policy) pozwalający na deklaratywny opis wymagań bezpieczeństwa serwisu: jakie tokeny są wymagane, które elementy wiadomości muszą być podpisane lub zaszyfrowane. Polityki umieszczane są w dokumencie WSDL i są odczytywane automatycznie przez klienta.

WS-ReliableMessaging

Protokół gwarantujący dostarczenie wiadomości pomimo awarii sieci lub węzłów pośrednich. Zapewnia dostarczenie co najmniej raz (*at-least-once*), co najwyżej raz (*at-most-once*) lub dokładnie raz (*exactly-once*) oraz zachowanie kolejności wiadomości.

WS-AtomicTransactions/Coordination

Specyfikacje umożliwiające koordynację rozproszonych transakcji atomowych (odpowiednik 2PC) pomiędzy usługami działającymi na różnych platformach.

WS-MetadataExchange

Protokół pobierania metadanych serwisu (WSDL, schematy XML, polityki WS-Policy) za pośrednictwem samego protokołu SOAP — bez konieczności odrębnego dostępu HTTP do WSDL.

SOAP over TCP

Optymalizacja transportu SOAP przy użyciu połączeń TCP zamiast HTTP — zmniejsza narzut protokołu w komunikacji wewnątrzsieciowej między zaufanymi węzłami.

16.1. Konfiguracja WSIT w praktyce

WSIT integruje się z JAX-WS na trzy sposoby: przez adnotacje na klasie serwisu, przez polityki WS-Policy osadzone w dokumencie WSDL lub przez plik konfiguracyjny `wsit-server.xml`.

16.1.1. WS-ReliableMessaging — włączenie przez WS-Policy w WSDL

Najczęściej stosowaną metodą konfiguracji jest osadzenie polityki WS-Policy bezpośrednio w dokumencie WSDL. Klient (Metro lub WCF) odczytuje politykę i automatycznie konfiguruje swój stos komunikacyjny.

Przykład polityki WS-ReliableMessaging (`exactly-once`) w WSDL:

```
<definitions ...
    xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata"
    xmlns:wsp="http://www.w3.org/ns/ws-policy"
    xmlns:wsrmp="http://docs.oasis-open.org/ws-rx/wsrmp/200702">

  <!-- Polityka niezawodnego przesyłania wiadomości -->
  <wsp:Policy wsu:Id="ReliableMessagingPolicy"
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
    <wsp:ExactlyOne>
      <wsp:All>
        <wsam:Addressing>          <!-- WS-Addressing jest wymagane przez WS-RM -->
          <wsp:Policy/>
        </wsam:Addressing>
        <wsrmp:RMAssertion>        <!-- główna asercja niezawodnego dostarczenia -->
          <wsp:Policy>
            <wsrmp:DeliveryAssurance>
              <wsp:Policy>
                <wsrmp:ExactlyOnce/>  <!-- możliwe też: AtLeastOnce, AtMostOnce -->
              </wsp:Policy>
            </wsrmp:DeliveryAssurance>
          </wsp:Policy>
        </wsrmp:RMAssertion>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>

  <!-- Dołączenie polityki do wiązania -->
  <binding name="OrderServiceBinding" type="tns:OrderService">
    <soap:binding style="document"
```

```
                transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsp:PolicyReference URI="#ReliableMessagingPolicy"/>
    <operation name="placeOrder">
      <soap:operation soapAction="urn:placeOrder"/>
      <input>  <soap:body use="literal"/> </input>
      <output> <soap:body use="literal"/> </output>
    </operation>
  </binding>
</definitions>
```

16.1.2. WS-SecureConversation — polityka sesji bezpieczeństwa

WS-SecureConversation skraca koszt uwierzytelnienia przez wynegocjowanie jednorazowego tokenu sesji (SCT — *Security Context Token*) zamiast dołączania certyfikatu do każdej wiadomości:

```
<wsp:Policy wsu:Id="SecureConversationPolicy"
    xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
    xmlns:sc="http://docs.oasis-open.org/ws-sx/ws-secureconversation/200512">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SymmetricBinding>
        <wsp:Policy>
          <sp:ProtectionToken>
            <wsp:Policy>
              <sp:SecureConversationToken
                  sp:IncludeToken=".../IncludeToken/AlwaysToRecipient">
                <wsp:Policy>
                  <sp:RequireDerivedKeys/>
                  <sp:BootstrapPolicy>  <!-- polityka pierwszego uzgodnienia -->
                    <wsp:Policy>
                      <sp:TransportBinding>
                        <wsp:Policy>
                          <sp:TransportToken>
                            <wsp:Policy><sp:HttpsToken/></wsp:Policy>
                          </sp:TransportToken>
                        </wsp:Policy>
                      </sp:TransportBinding>
                    </wsp:Policy>
                  </sp:BootstrapPolicy>
                </wsp:Policy>
              </sp:SecureConversationToken>
            </wsp:Policy>
          </sp:ProtectionToken>
          <sp:AlgorithmSuite>
            <wsp:Policy><sp:Basic256/></wsp:Policy>
          </sp:AlgorithmSuite>
          <sp:IncludeTimestamp/>
        </wsp:Policy>
      </sp:SymmetricBinding>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Po uzgodnieniu SCT kolejne wiadomości w sesji są podpisywane i szyfrowane szybkim kluczem symetrycznym zamiast certyfikatem RSA — redukcja czasu odpowiedzi serwisu

rzędu 3–10× przy długich sesjach.

16.1.3. Adnotacja `@BindingType` i gotowe konfiguracje Metro

Dla najprostszych scenariuszy Metro dostarcza gotowe stałe `SOAPBinding`:

```
// SOAP 1.2 + MTOM
@WebService
@BindingType(SOAPBinding.SOAP12HTTP_MTOM_BINDING)
public class DocumentService { ... }

// SOAP 1.1 + WS-Addressing (wymagany)
@WebService
@Addressing(enabled = true, required = true)
public class OrderService { ... }
```

W produkcyjnych scenariuszach (WS-Security, WS-RM, WS-SecureConversation) zaleca się konfigurację przez polityki WS-Policy w WSDL — jest ona czytelna dla wszystkich klientów zgodnych z WS-* i nie wiąże serwisu z konkretną implementacją Metro.

16.2. WSIT w projekcie JAX-WS

WSIT integruje się z JAX-WS przez adnotacje na klasie serwisu i polityki WS-SecurityPolicy osadzone w dokumencie WSDL — klient czytając WSDL konfiguruje się automatycznie.

Standard	Zastosowanie w praktyce
WS-SecurityPolicy	Deklaratywny opis wymagań bezpieczeństwa w WSDL
WS-Trust / STS	Federacja tożsamości — wydawanie i walidacja tokenów (SAML, X.509, Username)
WS-SecureConversation	Jeden token sesji zamiast wymiany certyfikatu przy każdej wiadomości
WS-ReliableMessaging	Gwarantowane dostarczenie przez sieci zawodne (*at-least-once*, *exactly-once*)
WS-AtomicTransactions	Rozproszone transakcje 2PC między serwisami na różnych platformach
WS-MetadataExchange	Automatyczne pobieranie WSDL i polityk przez klienta przez SOAP

Kolejny rozdział opisuje ebXML — standard OASIS do elektronicznego handlu B2B, który uzupełnia web serwisy o modelowanie procesów biznesowych i rejestry partnerów.

17. Electronic Business XML — ebXML

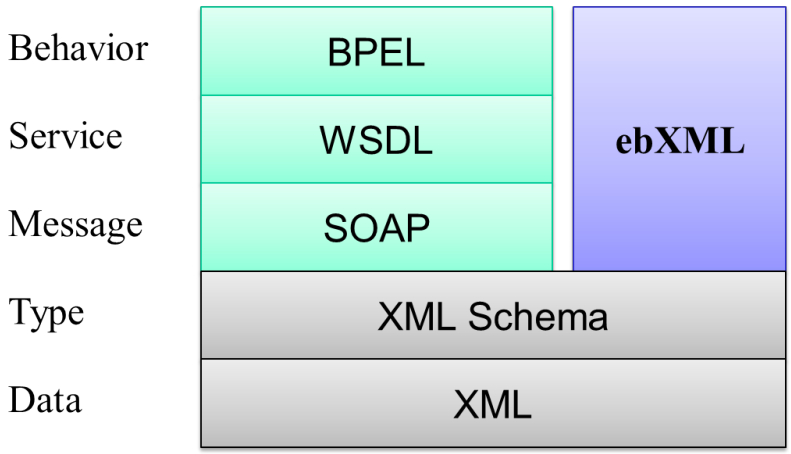
Gospodarka światowa opiera się w głównej mierze na handlu, a w nim podstawą jest komunikacja. W świecie *B2B* wymagane jest dokładne sprawdzenie partnera, szybkie i obiektywne porównanie ofert i nakreślenie szczegółowego porozumienia. Celem jest zminimalizowanie udziału człowieka w tym procesie.

ebXML jest standardem OASIS i UN/CEFACT, który określa dokumenty i sposoby prowadzenia handlu w Internecie pomiędzy przedsiębiorstwami.

Do prowadzenia handlu B2B w sieci potrzeba:

- medium łączącego systemy partnerów,
- języka wymiany danych zrozumiałego dla wszystkich,
- powszechnie dostępnego rejestru potencjalnych partnerów,
- mechanizmu do prostego i trafnego wyszukiwania podmiotów w rejestrach,
- uznanych przez wszystkich modeli procesów biznesowych prowadzących do zawierania transakcji,
- odpowiedniego poziomu bezpieczeństwa.

Choć ebXML wykorzystuje XML jako fundament, jest technologią uzupełniającą względem web serwisów.



Rysunek 38. WS vs ebXML

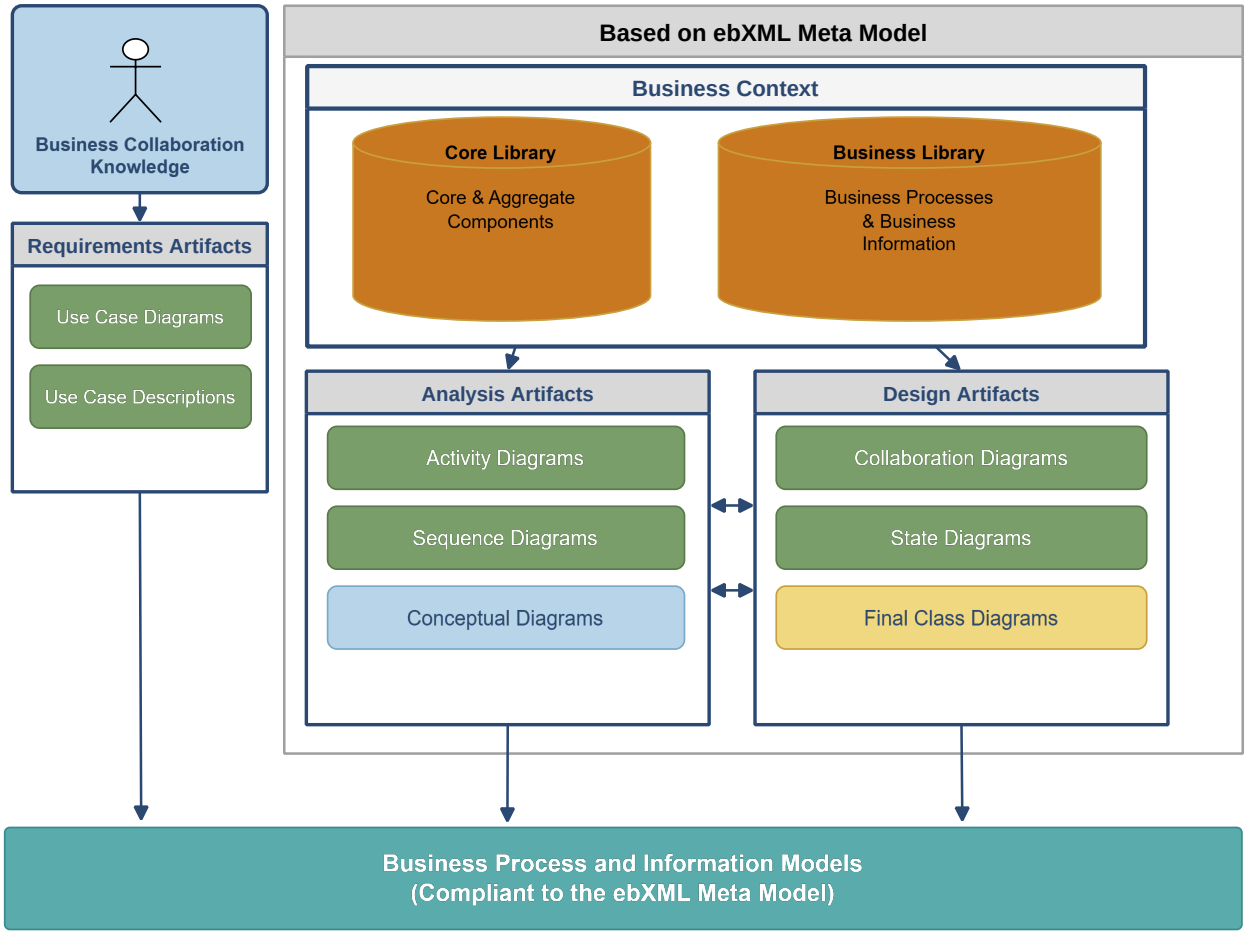
Standard ebXML szczegółowo definiuje:

- pojęcia używane w handlu B2B,
- struktury wymienianych dokumentów,
- konkretne procesy biznesowe,
- mechanizmy i języki wymiany danych,
- rejestry,
- zagadnienia bezpieczeństwa.

Standard zbudowany jest w sposób modularny i luźny — firmy mogą implementować tylko wybrane fragmenty.

17.1. Proces biznesowy (Business Processes)

Standard *ebXML* definiuje zestaw elementów zawartych w *Core Library*, z których buduje się konkretne wiadomości i zestawia w proces biznesowy (BP). Proces jest dokładnym opisem przebiegu komunikacji, wysyłanych wiadomości i możliwych odpowiedzi, rodzaju partnerów, zawiera definicję początku i końca procesu, sukcesu i porażki.



Rysunek 39. Tworzenie procesu biznesowego ebXML

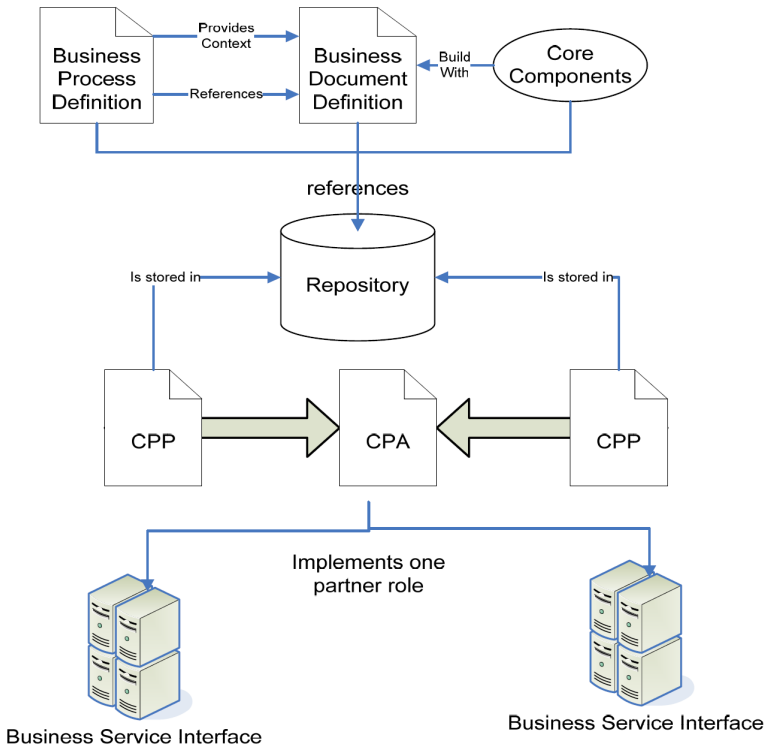
17.2. Collaboration Protocol Profile (CPP) i Agreement (CPA)

Proces biznesowy opisywany jest w dokumencie XML umieszczonym w *Collaboration Protocol Profile* (CPP), który opisuje:

- szczegółowe informacje (nazwę, dane kontaktowe, branżę),
- zdolności techniczne partnera,
- wymagania techniczne stawiane podmiotom współpracującym,
- wspierane procesy biznesowe,

- opis wspieranych zabezpieczeń i używanych certyfikatów.

CPP umieszcza się w repozytoriach ebXML, w których mogą być wyszukiwane przez zainteresowane strony. Jeśli uczestnik odnajdzie partnera, dokonuje wzajemnej analizy CPP w celu określenia wspólnie wspieranych technologii i procesów. Wynikiem jest dokument *Collaboration Protocol Agreement* (CPA), który określa szczegóły współpracy.



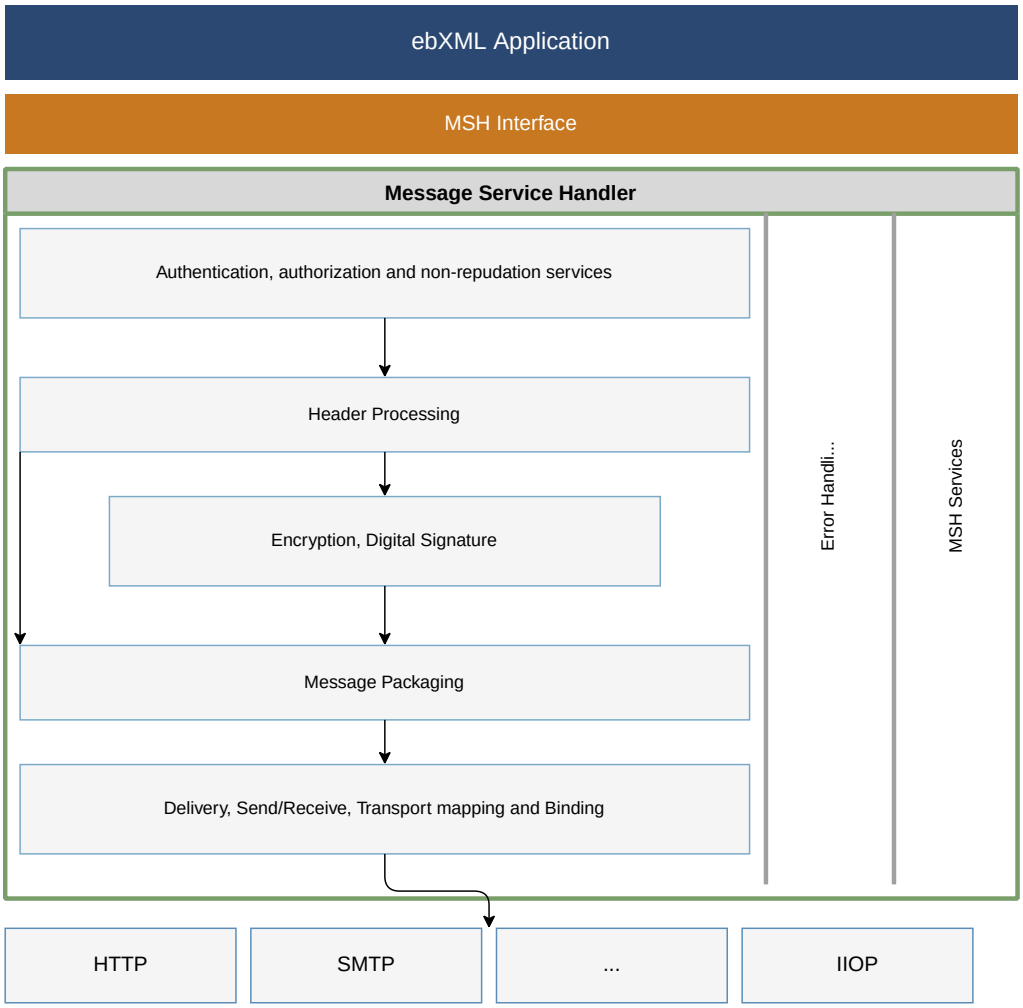
Rysunek 40. CPP/CPA

17.3. Messaging Service

Specyfikacja *Messaging Service* (OASIS) definiuje:

- sposób pakowania wiadomości ebXML (w koperty SOAP) i wysyłania ich standardowymi protokołami,
- bezpieczną i niezawodną metodę wymiany wiadomości,
- metody wewnętrznego rutingu.

Właściwości te implementuje komponent *Message Service Handler* (MSH).



Rysunek 41. Message Service Handler

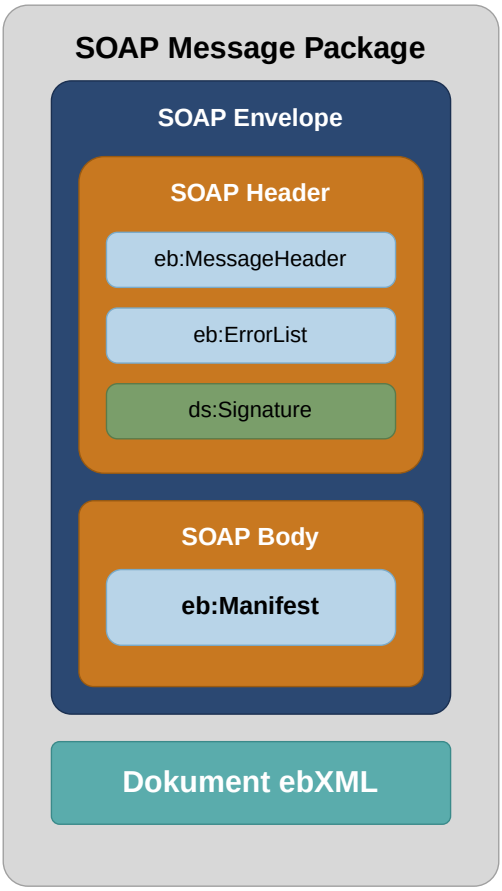
Specyfikacja określa następujące rozszerzenia do protokołu SOAP:

`MessageHeader`
: Obowiązkowy element nagłówkowy koperty SOAP. Wskazuje adresata i nadawcę, identyfikator CPA i inne informacje kontekstowe.

`ErrorList`
: Opcjonalny nagłówek w odpowiedzi — zawiera błędy i ostrzeżenia dotyczące poprzedniej wiadomości.

`Manifest`
: Właściwy dokument ebXML zaleca się umieszczać jako oddzielny załącznik (SwA), a referencję odnotowywać w `Manifest` w ciele koperty.

Podpis realizowany jest za pomocą XML-DSig.



Rysunek 42. Koperta SOAP ebXML

17.3.1. Elementy MessageHeader

Element	Opis
`From` / `To`	Nadawca i odbiorca wraz z rolami zgodnymi ze specyfikacją procesu
`CPAId`	Identyfikator CPA (lub zastępczy zestaw parametrów)
`ConversationId`	Unikalny identyfikator zestawu wiadomości składających się na dany proces
`Service`	Nazwa i typ serwisu
`Action`	Akcja zlecana w ramach serwisu
`MessageData`	Metadane: identyfikator, czas, `RefToMessageId`, `TimeToLive`
`DuplicateElimination`	Sygnalizuje, że duplikaty powinny być ignorowane

```
<eb:MessageHeader eb:id="..." eb:version="2.0" SOAP:mustUnderstand="1">
    <eb:From>
        <eb:PartyId>uri:example.com</eb:PartyId>
        <eb:Role>http://example.com/roles/Buyer</eb:Role>
    </eb:From>
    <eb:To>
```

```
        <eb:PartyId eb:type="someType">QRS543</eb:PartyId>
        <eb:Role>http://example.com/roles/Seller</eb:Role>
    </eb:To>
    <eb:CPAId>http://www.oasis-open.org/cpa/123456</eb:CPAId>
    <eb:ConversationId>987654321</eb:ConversationId>
    <eb:Service eb:type="myservicetypes">QuoteToCollect</eb:Service>
    <eb:Action>NewPurchaseOrder</eb:Action>
    <eb:MessageData>
        <eb:MessageId>UUID-2</eb:MessageId>
        <eb:Timestamp>2000-07-25T12:19:05</eb:Timestamp>
        <eb:RefToMessageId>UUID-1</eb:RefToMessageId>
        <eb:TimeToLive>123462555</eb:TimeToLive>
    </eb:MessageData>
    <eb:DuplicateElimination/>
</eb:MessageHeader>
```

17.3.2. ErrorList

```
<eb:ErrorList eb:id="3490sdo" eb:highestSeverity="error"
              eb:version="2.0" SOAP:mustUnderstand="1">
    <eb:Error eb:errorCode="SecurityFailure"
              eb:severity="Error"
              eb:location="ds:Signature">
        <eb:Description xml:lang="en-US">
            Validation of signature failed
        </eb:Description>
    </eb:Error>
</eb:ErrorList>
```

17.3.3. Manifest

Składa się z elementów `Reference` wskazujących na dokumenty ebXML (załączniki lub URL).

```
<eb:Manifest eb:id="Manifest" eb:version="2.0">
    <eb:Reference eb:id="pay01"
                  xlink:href="cid:payload-1"
                  xlink:role="http://regrep.org/gci/purchaseOrder">
        <eb:Schema
            eb:location="http://regrep.org/gci/purchaseOrder/po.xsd"
            eb:version="2.0"/>
        <eb:Description xml:lang="en-US">
            Purchase Order for 100,000 widgets
        </eb:Description>
    </eb:Reference>
</eb:Manifest>
```

17.4. ebXML — podsumowanie

ebXML definiuje kompletny stos do elektronicznego handlu B2B:

Komponent	Rola
CPP	Profil możliwości technicznych i procesów biznesowych jednego uczestnika
CPA	Umowa techniczna pomiędzy dwoma uczestnikami (wynik porównania CPP)
MSH (*Message Service Handler*)	Implementacja warstwy transportowej i bezpieczeństwa
Messaging Service	Pakowanie w SOAP, niezawodne dostarczenie, wewnętrzny routing
Rejestr ebXML	Repozytorium CPP wyszukiwanych przez potencjalnych partnerów handlowych

Na egzaminie 1Z0-897 wymagana jest znajomość architektury ebXML na poziomie ogólnym — umiejętność rozróżnienia CPP od CPA, roli MSH i sposobu osadzania dokumentów ebXML w kopercie SOAP (jako załączniki SwA z referencją w `Manifest`). Szczegółowa implementacja ebXML wykracza poza zakres egzaminu.

Ostatni rozdział podsumowuje zakres i strukturę egzaminu OCEJWSD (1Z0-897).

Załącznik A: Egzamin OCEJWSD

Certyfikat *Oracle Certified Expert, Java EE 6 Web Services Developer* (OCEJWSD) potwierdza szeroką wiedzę z zakresu usług sieciowych na platformie Java EE 6. Certyfikowany programista potrafi projektować, implementować i zabezpieczać serwisy SOAP i REST z użyciem JAX-WS, JAX-RS, JAXB, SAAJ oraz mechanizmów bezpieczeństwa Java EE.

Dane egzaminu:

- *Java Platform, Enterprise Edition 6 Web Services Developer Certified Expert Exam*
- oznaczenie: **1Z0-897**
- 68 pytań, z czego ocenianych jest 60
- próg zaliczenia: **64%** prawidłowych odpowiedzi
- czas trwania: **90 minut**
- wymagany uprzednio zdany egzamin OCJP (Oracle Certified Professional, Java SE — dawniej SCJP); certyfikat OCEJWSD nie zostanie wydany bez spełnienia tego warunku

A.1. Zakres egzaminu

Zagadnienie	Pytań
Apply best practices to design and implement web services	9
Configure Message Level security for a SOAP web service	7
Configure, Secure, and Deploy JavaEE Web Services	8
Create a RESTful web service	5
Create a RESTful web service implemented by an EJB component	3
Create a SOAP based web service implemented by an EJB component	4
Create a SOAP web service	4
Create a web service client for a web service	4
Create low-level SOAP web services	4
Use MTOM and MIME in a SOAP web service	7
Use WS-Addressing with a SOAP web service	4

Nazwy kategorii podane są w brzmieniu oryginalnym (angielskim) zgodnie z oficjalną dokumentacją Oracle. Spośród 68 pytań ocenianych jest 60 — 8 pytań jest nie ocenianych i służy kalibracji przyszłych egzaminów.

A.2. Wskazówki do egzaminu

A.2.1. Obszary o największej liczbie pytań

Pięć najważniejszych kategorii (razem 36 z 60 ocenianych pytań):

1. **Apply best practices** (9 pyt.) — projektowanie kontraktu, WSDL-first vs code-first, document/literal wrapped jako domyślny styl JAX-WS, dobór REST vs SOAP dla danego przypadku, projektowanie URI RESTful.
2. **Configure, Secure, and Deploy** (8 pyt.) — WSEE (JSR-109), `webservices.xml`, `@WebServiceRef`, bezpieczeństwo deklaratywne (`web.xml`, `@RolesAllowed`), typy wdrożeń (WAR vs EJB-JAR).
3. **Message Level Security** (7 pyt.) — WS-Security, `UsernameToken`, X.509, SAML; kluczowa różnica: bezpieczeństwo transportowe (TLS) vs bezpieczeństwo wiadomości (WS-Security).
4. **MTOM i MIME** (7 pyt.) — `@MTOM`, `MTOMFeature`, `@BindingType(SOAP11HTTP_MTOM_BINDING)`, `DataHandler`, `@XmlMimeType`, parametr `threshold`.
5. **WS-Addressing** (4 pyt.) — `@Addressing`, `AddressingFeature(enabled, required)`, `W3CEndpointReference`, nagłówki `To`/`From`/`ReplyTo`/`MessageID`.

A.2.2. Kluczowe adnotacje i klasy

Adnotacja / Klasa	Technologia i cel
`@WebService`, `@WebMethod`, `@WebParam`, `@WebResult`	JAX-WS (JSR-181) — definiowanie kontraktu
`@SOAPBinding` (`style`, `use`, `parameterStyle`)	JAX-WS — styl wiadomości SOAP
`@RequestWrapper`, `@ResponseWrapper`	JAX-WS — kontrola elementów opakowujących WRAPPED
`@HandlerChain` + `SOAPHandler` / `LogicalHandler`	JAX-WS — przechwytywanie wiadomości
`@Addressing`, `@MTOM`, `@BindingType`	JAX-WS — rozszerzenia jakości usług
`@WebFault`	JAX-WS — mapowanie wyjątku Java na SOAP Fault
`@WebServiceRef`	WSEE — wstrzykiwanie referencji do serwisu w kontenerze Java EE
`@XmlRootElement`, `@XmlType`, `@XmlElement`, `@XmlAttribute`	JAXB — mapowanie klasy na XML
`@XmlJavaTypeAdapter`	JAXB — niestandardowa konwersja typów (`Date`, `Map`)

Adnotacja / Klasa	Technologia i cel
`@Path`, `@GET` / `@POST` / `@PUT` / `@DELETE`	JAX-RS — definicja zasobu REST
`@Produces`, `@Consumes`, `@PathParam`, `@QueryParam`	JAX-RS — negocjacja treści i parametry
`@MessageDriven`, `@ActivationConfigProperty`	JMS/MDB — bean konsumujący kolejkę JMS
`Dispatch<T>` / `Provider<T>`	JAX-WS — dynamiczny klient / niskopoziomowy serwis

A.2.3. Typowe pułapki egzaminacyjne

- `@SOAPBinding` — **domyślny styl** to `DOCUMENT` / `LITERAL` / `WRAPPED` — nie RPC.
- `@Oneway` — metoda musi zwracać `void` i nie deklarować sprawdzanych wyjątków.
- `@WebFault` — wyjątek musi mieć metodę `getFaultInfo()` i odpowiednie konstruktory.
- **Handler chain** — dla wiadomości wychodzącej: najpierw `LogicalHandler`, potem `SOAPHandler`; dla przychodzącej: odwrotnie.
- `SESSION.AUTO_ACKNOWLEDGE` w JMS — potwierdzenie następuje po powrocie z `receive()` lub `onMessage()`, nie w momencie wywołania.
- **MTOM threshold** — dane poniżej progu pozostają base64 wewnątrz XML (nie są optymalizowane).

A.2.4. Co NIE jest zakresie egzaminu

- **WSDL 2.0** — egzamin obejmuje wyłącznie WSDL 1.1,
- **JAX-RS 2.0** (Client API: `ClientBuilder`, `WebTarget`) — egzamin oparty na JAX-RS 1.1 (Java EE 6),
- **JMS 2.0** (`JMSContext`, uproszczone API) — egzamin dotyczy JMS 1.1,
- szczegółowa implementacja ebXML i WSIT/Metro,
- Jakarta EE i pakiety `jakarta.` (egzamin używa `javax.`).

Jeśli znasz nowsze wersje platformy (Java EE 7/8 lub Jakarta EE), zwróć szczególną uwagę na funkcjonalności, które **nie istniały** w Java EE 6: `ClientBuilder` (JAX-RS 2.0), `JMSContext` (JMS 2.0), `@Suspended`/`AsyncResponse` (JAX-RS 2.0 async). Pytanie o JAX-RS Client API na tym egzaminie odnosi się do **ręcznego** tworzenia żądań HTTP, nie do `ClientBuilder`.