




# Web services (continued)

Patryk Czarnik

XML and Applications 2013/2014  
Lecture (week) 7 - 18.11.2013

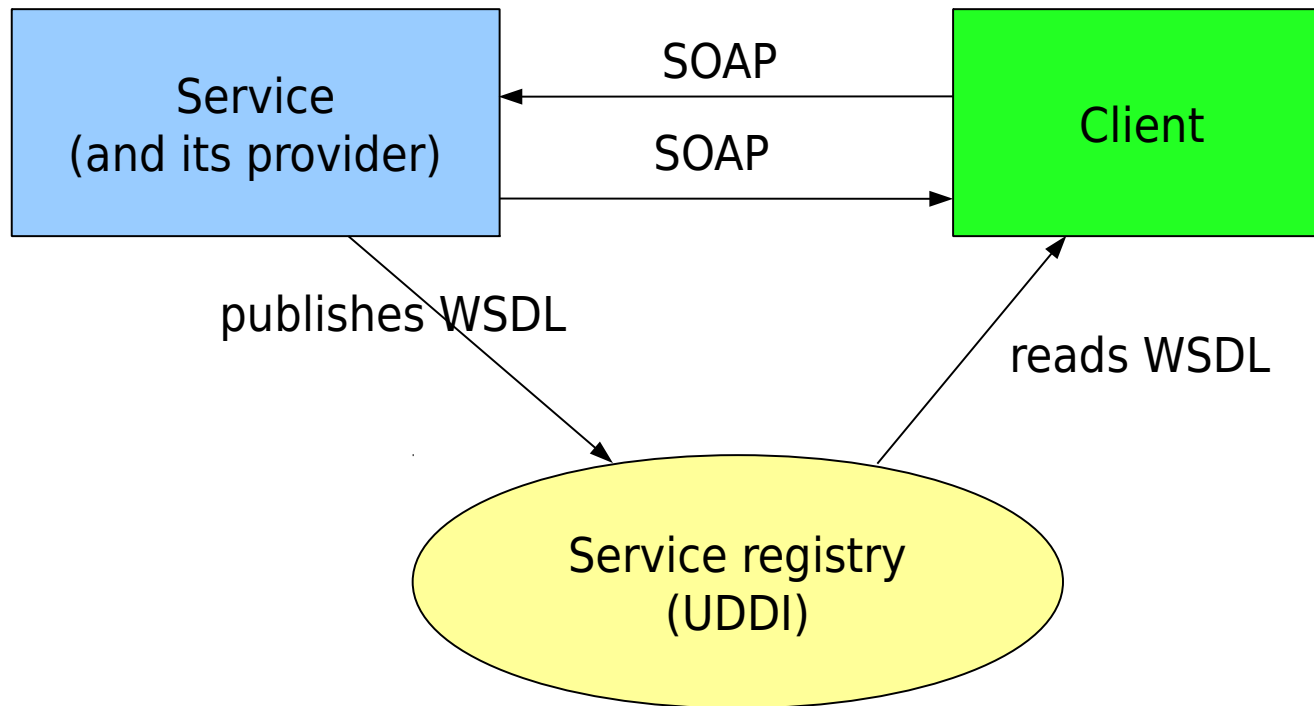
# Evolution of internet applications

- human  human
  - email
  - WWW sites written manually
- application  human
  - web applications (e.g. an internet shop)
- application  application
  - low-level technologies and ad-hoc solutions
  - “web services”

# Web Services

- Idea: a website for programs (instead of people)
- General definition
  - communication based on high-level protocols
  - structural messages
  - services described
  - searching services
- Concrete definition: “Classical” Web-Services
  - HTTP or other protocols
  - SOAP
  - WSDL
  - UDDI
  - Web Services Interoperability

# Classical vision of web services operation



In fact, most of deployed solutions don't use the UDDI layer

# Basic standards – recall

- SOAP – communication protocol
  - mainly definition of message (envelope) format
  - XML message with optional binary attachments
  - headers (optional XML elements) and body content (one XML element according to WS-I BP)
- WSDL – service description (interface)
  - XML element or type definitions written with XML Schema
  - **port type** – set of operations with their input and output messages
  - **binding** – how requests and responses will be sent through the net (in our case: how they are represented as SOAP messages)
  - service **instance** (set of **ports**) – address where the service is available

# Service registration and discovery

- Idea
  - service provider registers service
  - user searches for service and finds it in registry
- Universal Description Discovery and Integration (UDDI)
  - available as service (SOAP)
  - business category-based directory (“yellow pages”)
  - searching basing on service name, description (“white pages”)
  - registration and updates for service providers

## UDDI – issues

- Main issue – who can register?
  - anybody – chaos and low reliability
  - accepted partners – an institution responsible for access policy needed, no such (widely accepted) institution exists
- Reality
  - UDDI rarely used
  - if ever – for “local” SOA-based solutions (intranets)

# Service Oriented Architecture

- Old-school approach for software building (when we have some logic already developed and we want to use it again):
  - link and compile static components – code, libraries, etc.
- SOA approach:
  - use working services to obtain existing logic
  - (to make it possible) build your pieces of software as services
  - Result: services built basing on other services
- Main differences (advantages?)
  - we don't have to include a component to use it
  - we avoid not only code duplication, but also a duplication of working logic

**SOA is trendy**  
But use it reasonably, please!



# Web services in Java

Basically – web services and web service clients can be built from scratch in any technology

- but it would be the same mistake as reading XML documents char by char.
- Low-level technologies:
  - HTTP servlets and HTTP clients supported by XML processing APIs (DOM, SAX, StAX, JAXB, Transformers, ...)
  - SOAP with Attachments API for Java (**SAAJ**)
    - extension of DOM directly supporting SOAP
- High level approach (with low level hooks available):
  - Java API for XML Web Services (**JAX-WS**)

# Web Services in Java

- WS support (XML APIs, SAAJ, JAX-WS) present in Java SE
  - JAX-WS and some of XML APIs since version 6.0
- Client side:
  - Possible to develop and run WS client in Java SE without any additional libraries!
- Server side:
  - Developing and compiling WS server (without any vendor-specific extensions) available in Java SE
  - Running a service requires an application server and a WS implementation
    - “Big” app servers (Glassfish, JBoss, WebSphere...) have preinstalled WS implementations
    - Lightweight servers (Tomcat or even Jetty) can be used by applications equipped with appropriate libraries and configuration

# SAAJ

- Package `javax.xml.soap`
- Main class – `SOAPMessage`
- Tree-like representation of SOAP messages
  - extension of DOM
  - easy access to existing and building fresh SOAP messages
  - support for HTTP headers, binary attachments, ...
- Easy sending of requests from client side
  - see example `SAAJ_Weather`
- Possible implementation of server side as a servlet
  - see example `SAAJ_Server`

# JAX-WS – introduction

- Annotation-driven
- Uses JAXB to translate Java objects to/from XML
- Central point: Service Endpoint Interface (SEI)
  - Java interface representing a WS port type
    - `kalkulator.Kalkulator` and `pakiet.Service` in our examples
- Translation between web services world (WSDL) and Java
  - **top-down**: from WSDL generate Java
    - server side – service interface and implementation skeleton
    - client side – proxy class enabling easy remote invocations
    - both sides – auxiliary classes, usually JAXB counterparts of XML elements appearing in messages
  - **bottom-up**: from Java code generate WSDL  
(and treat the Java code as a WS implementation)
    - usually done automatically during application deployment

# Advantages and risks of using JAX-WS

- High level view on web service
  - details of communication and SOAP/XML not (necessarily) visible to a programmer
  - proxy object on client side enables to transparently invoke methods on server-side just like on local objects
- Automatic generation/interpretation of WSDL
  - conformance to WSDL controlled by system
- Bottom-up scenario – easy introduction of WS interface to already existing systems
  - or for programmers not familiar with WSDL/XML details
- Risk of
  - accidental service interface (WSDL)  
(automatically generated, not elaborated enough)
  - inefficiency

# J-WS – main elements

- Class level annotations:
  - `@WebService`, `@SOAPBinding`
- Method-level annotations:
  - `@WebMethod`, `@OneWay`, `@SOAPBinding`, `@RequestWrapper`, `@ResponseWrapper`
- Parameter-level annotations:
  - `@WebParam`
  - `@WebResult` (syntactically a method annotation, applies to what the method returns)
- Support for specific technologies
  - `@MTOM` – automatically created binary attachments
  - `@Addressing` – adds WS-Addressing headers

## J-WS – low level hooks

- Providers – low level server side
  - Useful when
    - high efficiency required (e.g. streaming processing)
    - XML technology used in implementation
- Dispatch – low level client side
- One way methods
- Asynchronous client calls
- Handlers and handler chains
  - additional processing of messages between client and server logic
  - one place to perform common logic: logging, authentication, session binding

# JAX-WS examples

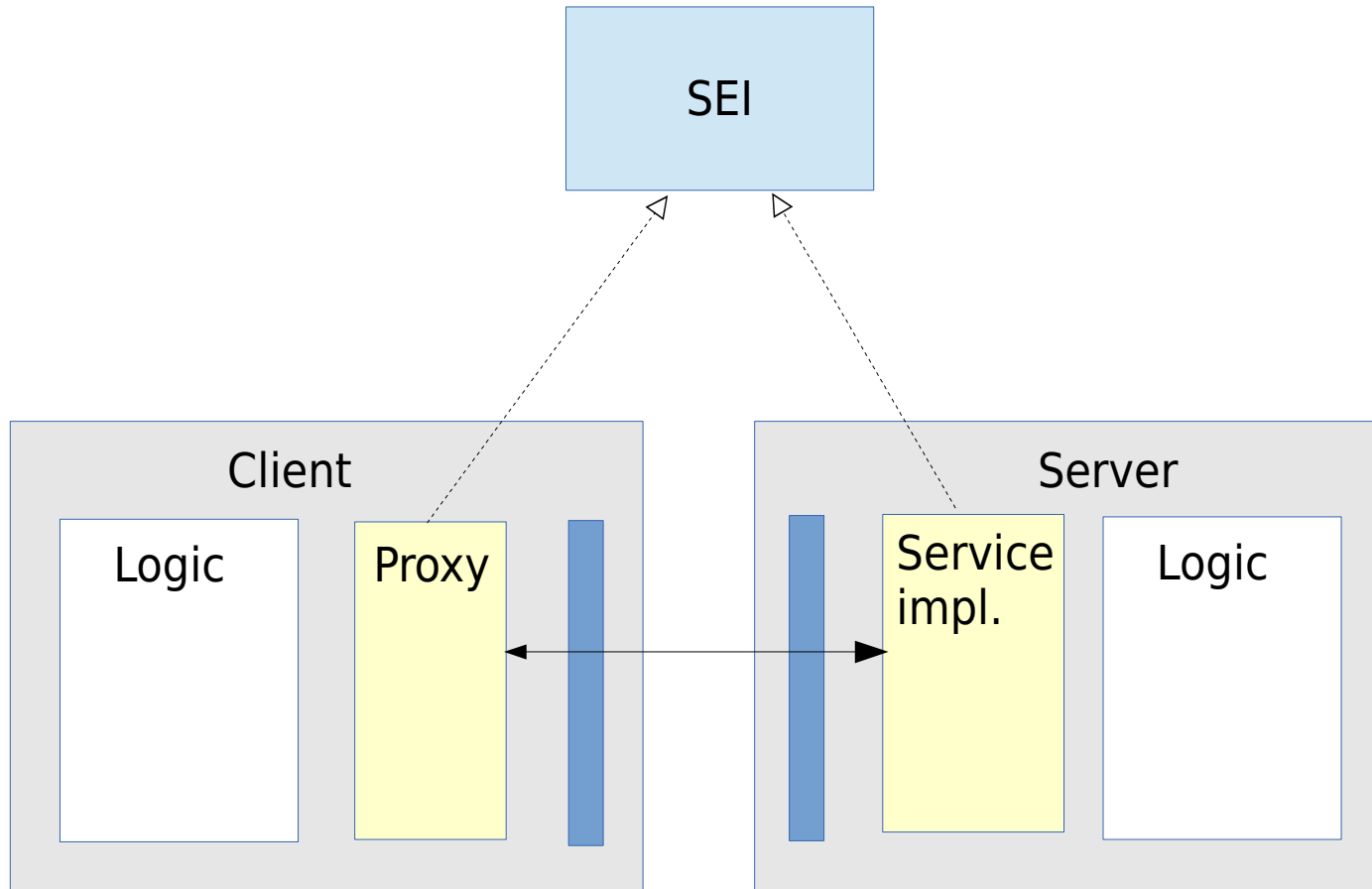
Details to note:

- top-down (**Kalkulator**):
  - (different) form of WSDL in RPC and Document styles
  - 3 ways WSDL can be translated to Java (and SOAP) (RPC, document-wrapped, document-bare)
  - **@WebService** annotation in implementation class
- bottom-up (**Hello**)
  - how annotations affect SOAP messages (and WSDL)
  - how Java objects are represented in SOAP messages (JAXB)
- high level proxy clients (**JAXWS\_Weather**)



# JAX-WS architecture

When both sides written in Java...



High level Java clients available  
also for non-Java servers!

# WSDL and SOAP interaction

- Basically – specified through binding element in WSDL
  - not so simple, because of many possibilities
- **RPC** style
  - SOAP XML structure derived basing on operation name and message parts
- **Document** style
  - theoretically designed to allow sending arbitrary XML documents
  - in practice also used for RPC realisation, but the author of WSDL has to define an appropriate document structure
    - (some tools may be helpful, e.g. bottom-up service generation in Java JAX-WS)
- Message use: **literal** or **encoded**.
  - We should use literal in modern applications.

# Web Services advantages and problems

- Advantages:
  - Standardised, platform-independent technology
  - Interoperability
  - Existing tools and libraries
- Main drawbacks:
  - Inefficiency
    - size of messages → transfer, memory usage
    - data representation translated many times on the road from client to server (and vice versa) → processor usage / time
  - Complex standards, especially when using something more than raw WSDL+SOAP

# Are Web Services good or bad?

It depends on the actual case, of course.

- Web Service recommended when
  - Many partners or public service (standardisation)
  - Heterogeneous architecture
  - Text and structural data already present in problem domain
  - Interoperability and flexibility more important than efficiency
- Web Service?... not necessarily
  - Internal, homogeneous solution.
  - Binary and flat data
  - Efficiency more important than interoperability and flexibility

# REST – motivation

- Complexity and inefficiency of SOAP-based services led designers/researchers to propose other solutions
  - service-oriented
  - but simpler (and less general) than classical WS
- The most popular alternative these days:  
Representational State Transfer (**REST**)
  - Idea by Roy Fielding (2002)
  - Very popular solution for integration of JavaScript clients (AJAX) with servers
  - In Java (EE) available through JAX-RS interface

# REST – basic ideas

- Service = set of resources
  - resource identified by its URL
  - best practices: URLs unique, resources organised in collections

`http://rest.example.org/service/orders/302312`

- Resources
  - are representable (e.g. as XML, other formats available)
  - can be transferred through the net
- HTTP – protocol for remote access to the resources
  - HTTP methods (GET, PUT, etc) used directly

# HTTP methods (in REST, but not only)

- GET – read the resource
  - no side effects
- PUT – write the resource
  - request body contains new contents
  - for writing new and overriding existing resources
- DELETE – deletes the resource
- POST – “get this data and something with it”
  - conceptually incompatible with REST ideas
  - used in practice to call remote logic more complex than reading or writing a resource
- OPTIONS, HEAD – no special meaning in REST
  - well, getting last modification time makes sense in REST...

# JAX-RS – REST in Java

- Java API for RESTful Services (JAX-RS)
- Annotation driven API
- Support for different ways of passing arguments
- Content-type negotiation
  - the same resource may be available in different formats
- Easy to write HTTP servers
  - REST-specific logic has to be written manually