



INFS803

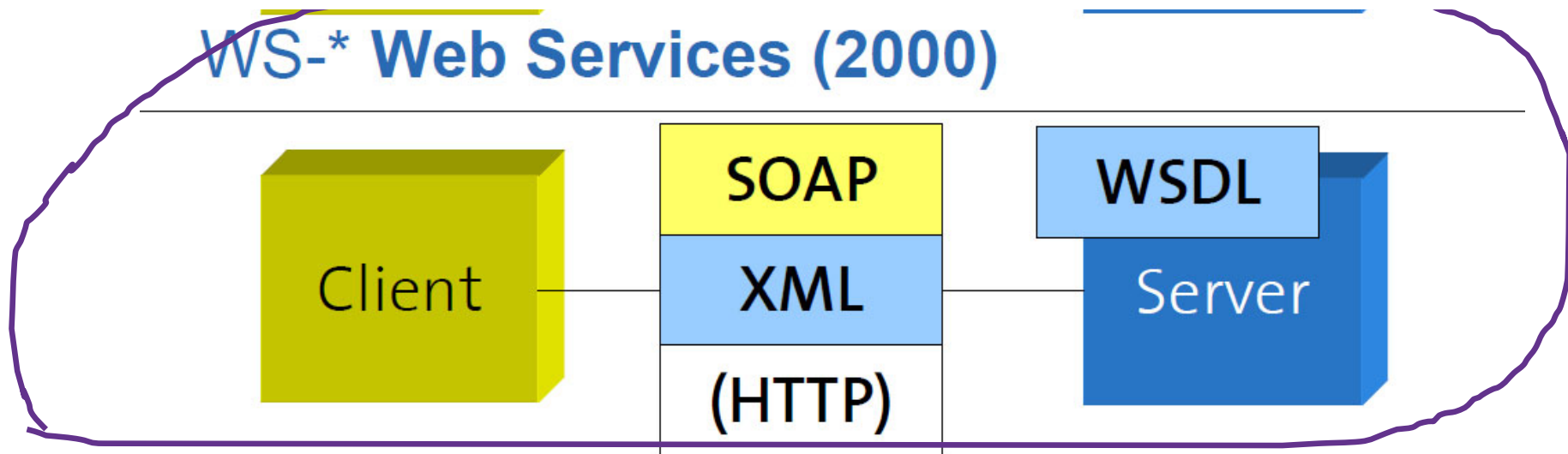
Cloud Computing

**Web service protocols and developing
RESTful services**

1. Web service architecture: traditional and RESTful
2. Data description language: XML and JSON
3. SOAP
4. WSDL
5. JSON Schema
6. Developing Restful Web APIs

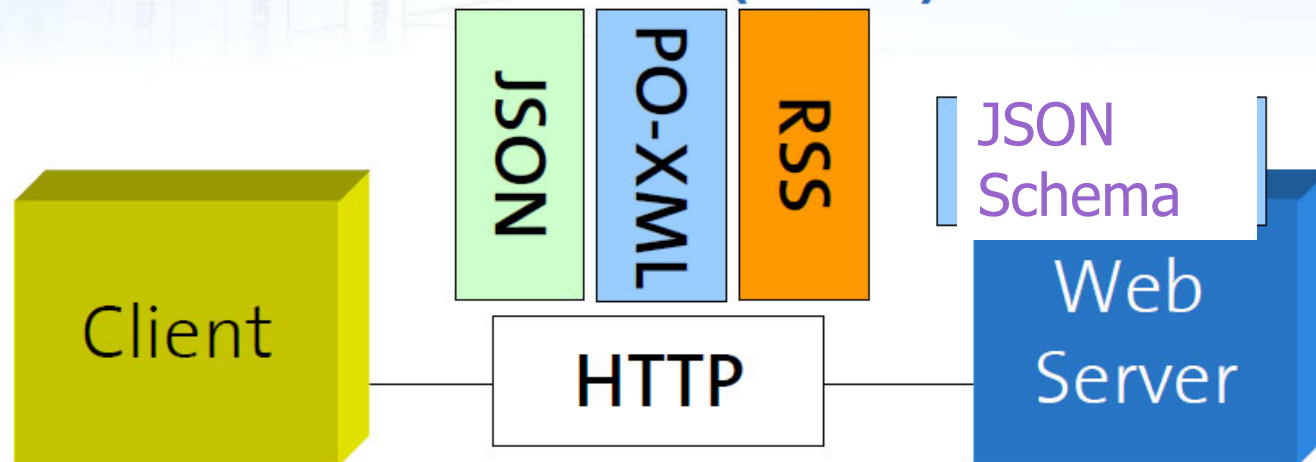
Traditional Web service invocation

WS-* Web Services (2000)

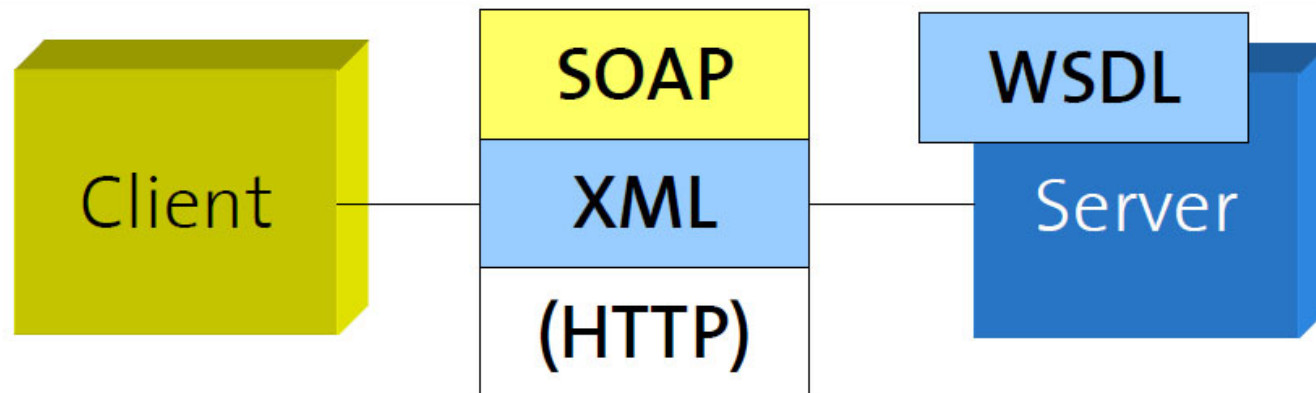


Restful Web Services/APIs

RESTful Web Services (2007)



WS-* Web Services (2000)



- XML basics
 - Elements and syntax
 - From XML to JSON
 - DTD (Document Type Definition)

- Markup language for structured information
- A markup language is a set of symbols that can be placed in the text of a document to demarcate (create boundaries) and label the parts of that document
- XML is a meta-markup language because it lets you create your own markup language
- XML allows author to *customise* own elements, i.e., their own tags
- The elements of the author's own choice are used to **structure data** and **provide it meaning**

Why use XML?

- Simplicity
- Extensibility
- Interoperability
- Openness
- Platform-independent, industry accepted standard
- Integration of all traditional databases
- One data, different views
- Internationalization (Unicode)
- Extensive software tools available
- Industry-specific schemas

- HTML has fixed tag semantics (defined by browser behaviour) and pre-defined tags
 - <H1>, <P>, <BODY>, <TD> ...
 - W3C keeps extending this tag set and semantics (ie, what happens when displayed in a browser)
- XML specifies neither a tag set nor semantics
 - It is a meta-language for describing (markup and other) languages
 - You can define your own tags, and the structure of these tags
 - Semantics are provided by applications that process XML documents or by style-sheets

XML vs. HTML : Simple Example

```
<TABLE>
  <TR>
    <TD>Thomas</TD><TD>Atkins</TD>
  </TR>
  <TR>
    <TD>age:</TD><TD>30</TD>
  </TR>
</TABLE>
```

HTML, using
pre-defined
tags which
have pre-
defined
meanings wrt
presentation
in a browser

```
<Person>
  <Name>
    <First>Thomas</First>
    <Last>Atkins</Last>
  </Name>
  <Age>30</Age>
</Person>
```

XML, with
user-defined
tags which
are chosen to
convey intent
of their
content

Example: classes.xml

```
<?xml version="1.0"?>
<classes>
  <class>
    <classID>CS115</classID>
    <department>ComputerScience</department><credits req="yes">3</credits>
    <instructor>Adams</instructor><title>Programming Concepts</title>
  </class>
  <class>
    <classID semester="fall">CS205</classID>
    <department>ComputerScience</department><credits req="yes">3</credits>
    <instructor>Dykes</instructor><title>JavaScript</title>
  </class>
  <class>
    <classID semester="fall">CS255</classID>
    <department>ComputerScience</department><credits req="no">3</credits>
    <instructor>Brunner</instructor><title>Java</title>
  </class>
</classes>
```

XML Elements and Syntax

- ❑ Elements compose a document, e.g., in a Unit Outline

`<credit_points>12.5</credit_points>`

credit_points is a type different from *ref* containing the name of a reference

- ❑ Example for a Unit Outline:

```
<section>
  <sectionname>References</sectionname>
  <ref>Castro E., XML for the World Wide Web, Peachpit Press, 2000</ref>
  <ref>Deitel, H., et al., XML: How to Program, Prentice Hall, 2001</ref>
  <ref>Holzner, S., Inside XML, New Riders Publishing, 2001</ref>
</section>
```

XML Elements and Syntax

- ❑ The example shows the following:
 - **Boundaries.** Tags `<section>` and `</section>` surround collection of text and markup
 - **Roles.** `<sectionname>` .. `</sectionname>` has a different purpose from `<ref>` .. `</ref>`.
 - **Positions.** The first `<ref>` .. `</ref>` is placed logically after `<sectionname>` .. `</sectionname>` and sensibly would be printed to a browser like this.
 - **Containment.** Both `<sectionname>` and `<ref>` elements are nested within the `<section>` element.

XML Documents...

- All XML documents need to be
 - Well-formed (compulsory)
 - Valid against a Schema/DTD (optional)
- Parsers will check if document is well-formed
- There are Validating parsers and Non-Validating parsers
- With a validating parser, the validity is checked if the document refers to a specified DTD or XML Schema

A Few Definitions...

- Content
 - "The mobile phone revolution has just begun"
- Tags
 - `<statement>` The mobile phone ... `</statement>`
 - `<statement>` = **Start tag / Opening tag**
 - `</statement>` = **End tag / Closing tag**
- Tags are case sensitive: `<name>`, `<NAME>`, `<Name>` are treated as different (**CAUSES MANY ERRORS!**)
- Element = Tag + Content

- An Element in XML consists of:
 - A start tag, Content & an end tag
 - e.g. `<dateOfBirth>2003-01-01</dateOfBirth>`
- A start tag may include one or more **attributes** of the element
 - `<address type="permanent"> ... </address>`
- Empty elements without any content are allowed
 - `<applause />` -- same as `<applause> </applause>`
 - XHTML: `
` -- same as `
 </br>`
 - Note: an empty element may have attributes

What can we store in elements?

- Elements are used to contain
 - Elements
 - Data (Typical case)
 - Character references
 - Entity references
 - Comments
 - Processing instructions
 - CDATA Sections

XML Character data

- Character data is text in the document, not markup tags
 - May be of any allowable Unicode value
 - Certain characters are reserved or they are not part of the ASCII character set and must be entered using character or entity references
- Element content is often referred to as **parsed-character data** (PCDATA)
- PCDATA is any “well-formed” text string, most text strings are “well-formed” except those that contain symbols reserved for XML, such as < > &

Entity References

- Used to place *string literals* into elements/attributes
 - Start with &, End with ;
 - e.g. **&**;
- Some pre-defined entity references are:

<code>&amp;</code>	<code>&</code>
<code>&lt;</code>	<code><</code>
<code>&gt;</code>	<code>></code>
<code>&apos;</code>	<code>'</code>
<code>&quot;</code>	<code>"</code>

Can define own entity references

Character References...

- A *character reference* is where use is made of a decimal or hexadecimal number to represent a character able to be stored in XML data and is displayable (for instance, in a Web browser), e.g., ©
- Such a character is not able to be placed in its displayable form into a document because it is not available from the input device, e.g., a Japanese character trying to be entered in a Turkish word processor

Character References

- Allows to represent Unicode characters
- Allowed forms of references are:
 - Decimal: `&#DDDDD;` (1 to 5 digits), e.g., `©`
 - Hexadecimal: `&#xHHHH;` (1 to 4 digits), e.g., `©`
 - Both the above represent ©
- Hexadecimal form is preferred
- Cannot use character references for element/attribute names

- Comments are like in HTML
 - `<!-- This is a comment -->`
- Comments should be useful and aimed to improve communication
- Avoid comment noise (i.e. too many trivial comments)
- Short & targeted comments are ideal

Processing Instructions (PIs)

- Processing hint, script code or presentational info can be indicated to a parser
 - Is parser-specific, so not all parsers will respond to a PI's in the same way
- Syntax: `<? Some processing instruction ?>`
 - Examples:

```
<?xml-stylesheet href="style.css"
  type="text/css"?>
<? Setup-templates ?>
```

CDATA (Char Data) sections

- These are needed to capture any data that may confuse the parser
 - This entire section will not be parsed
- CDATA sections cannot be empty
- Example:

```
<![CDATA[  
    <html><body>Content</body></html>  
]]>
```
- Here we have some HTML as an unparsed string within XML; we don't want the XML parser to identify `<html>` and `<body>` as XML tags!

- XML documents contain,

- An optional *prolog*

```
<?xml version=1.0 encoding="UTF-8" standalone="no" ?>  
<!DOCTYPE book SYSTEM "book.dtd">
```

- A body with a root element

```
<book>
```

```
  <title>Long live XML</title>
```

```
</book>
```

- An optional *epilog* – seldom used

XML Documents - Prolog

- A prolog contains
 - XML declaration
 - Miscellaneous statements or comments
 - Document type declaration
- This order has to be followed or the parser will generate an error message
- Example:

```
<?xml version=1.0 encoding="UTF-8" standalone="no" ?>  
<!-- This document describes one book -->  
<!DOCTYPE book SYSTEM "book.dtd">
```

Well-formedness Rules

- An XML document must follow “Document” *production*, i.e., document contains a *prolog*, a root element and a miscellaneous part to which the following rules apply
- One root element containing all other elements
- Elements must have both a start and end tag, except that *empty* elements end in “/”
- Elements must nest, i.e., elements do not overlap, e.g.: `<section><sectionname> ...</sectionname> ... </section>`

Well-formedness Rules (Cont'd)

- Attribute values in double or single quotes, e.g.:


```
<book settext="no">Castro E., XML for the World Wide Web, Peachpit Press, 2000.</book>
```
- Markup characters (eg, `<` `>` `&`) do not appear in parsed content, e.g.: `<eqn>1 < 3</eqn>` for showing `1 < 3`
- Element names may begin with letter or underscore or ":" and remaining characters include alphanumeric, "_", "-", ":", and "."
- Cannot use the same name for two or more attributes of the same element

- Write an XML document to describe the following information:
 - Web Services - Concepts, Architectures and Applications, Gustavo Alonso et al., Springer, 2004
 - Web Services and SOA: Principles and Technology, 2/E, Michael Papazoglou, Pearson, 2012

From XML to JSON

Deserialization: convert json string to JS object

```
who = {  
  "name": 'Chuck',      String  
  
  "age": 29,             Integer  
  
  "college" : true,      Boolean  
  
  "offices" : [ '3350DMC', '3437NQ' ], List/Array  
  
  "skills" : { "fortran": 10,  
               "C": 10,           Object  
               "C++": 5,  
               "python" : '7'  
             }  
};
```



JSON is hierarchical!

*In Javascript,
"alert(who.name)" will display "Chuck"*

JSON example

```
{ "employees": [  
    { "firstName": "John", "lastName": "Doe" },  
    { "firstName": "Anna", "lastName": "Smith" },  
    { "firstName": "Peter", "lastName": "Jones" }  
  ] }
```

Ede | [arq1w](#)

If assigned to variable x , then x becomes an object and we can use $x.employees[0].firstName$ to fetch the string "John"

XML DTD – a brief introduction

- For validate xml documents
 - What **type** of data can be contained in an element
 - What **elements** must be included
 - The **order** of the elements
 - The number of **times** each element can **occur** in the document
 - Any **attributes** of the elements

- `<!ELEMENT br EMPTY>`: `
`
- `<!ELEMENT from (#PCDATA)>`
- `<!ELEMENT note ANY>`
- `<!ELEMENT note (to,from,heading,body)>` : in sequence
- `<!ELEMENT note (message)>`: one occurrence
- `<!ELEMENT note (message+)>`: more than one
- `<!ELEMENT note (message*)>`: zero or more
- `<!ELEMENT note (message?)>`: zero or one
- `<!ELEMENT note (#PCDATA|to|from|header)*>`: mixed content

A DTD for the XML

Persons.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT Persons ((Person+))>
<!ELEMENT Person ((Name, Age))>
<!ELEMENT Name ((First, Last))>
<!ELEMENT Last (#PCDATA)>
<!ELEMENT First (#PCDATA)>
<!ELEMENT Age (#PCDATA)>
```

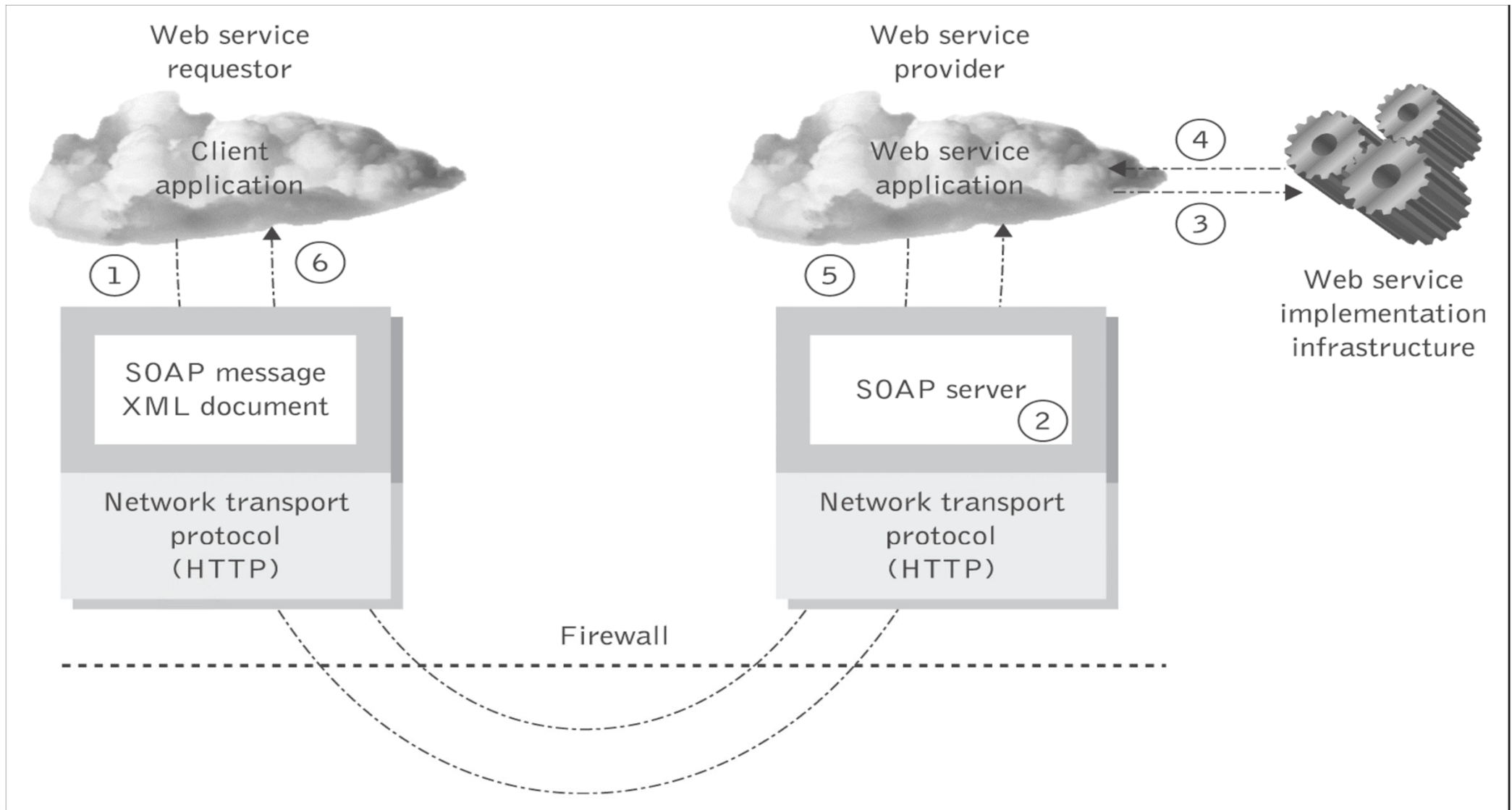
Parsed char data
'<' will be changed
to '<'

XML for Persons with a DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Persons SYSTEM "Persons.dtd">
<Persons>
  <Person>
    <Name>
      <First>Thomas</First>
      <Last>Atkins</Last>
    </Name>
    <Age>30</Age>
  </Person>
  <Person>
    <Name>
      <First>Sachin</First>
      <Last>Tendulkar</Last>
    </Name>
    <Age>38</Age>
  </Person>
</Persons>
```

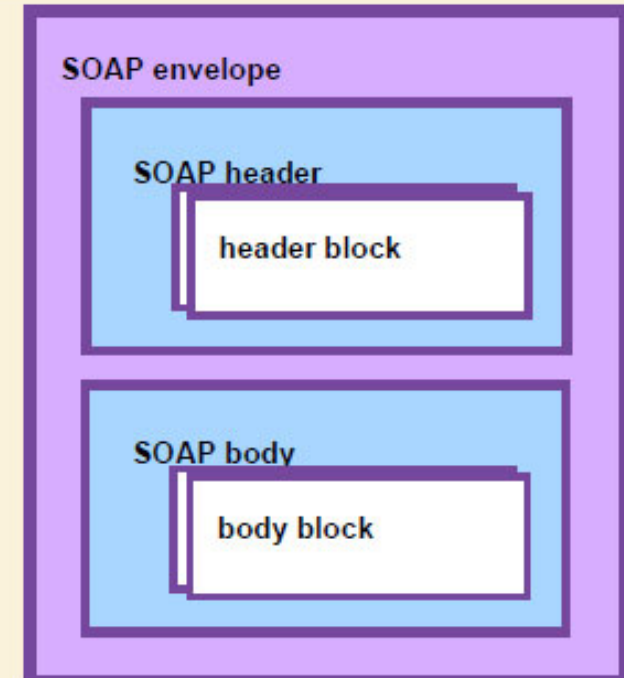
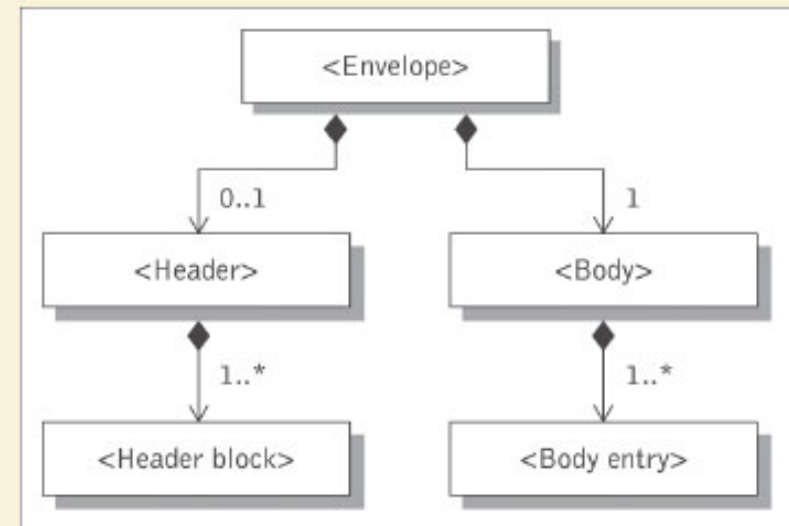
- SOAP: Simple Object Access Protocol

Distributed messaging using SOAP



SOAP messages

- SOAP is based on **message exchanges**.
- Messages are seen as **envelopes** where the application encloses the data to be sent.
- A SOAP message consists of an `<Envelope>` element containing an optional `<Header>` and a mandatory `<Body>` element.
- The contents of these elements are application defined and not a part of the SOAP specification.
- A SOAP `<Header>` contains blocks of information relevant to how the message is to be processed. This helps pass information in SOAP messages that is not for the application but for the SOAP engine.
- The SOAP `<Body>` is where the main end-to-end information conveyed in a SOAP message must be carried.



SOAP envelope and header

```
<env:Envelope
  xmlns:env="http://www.w3.org/2003/05/soap-envelope"
  env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  .....
</env:Envelope>
```

Example of SOAP envelope

```
<env:Envelope
  xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
  ...
  <env:Header>
    <tx:transaction-id
      xmlns:tx="http://www.transaction.com/transaction"
      env:mustUnderstand="true">
      512
    </tx:transaction-id>
    <notary:token xmlns:notary="http://www.notarization-services.com/token"
      env:mustUnderstand="true">
      GRAAL-5YF3
    </notary:token>
  </env:Header>
  .....
</env:Envelope>
```

Example of SOAP header

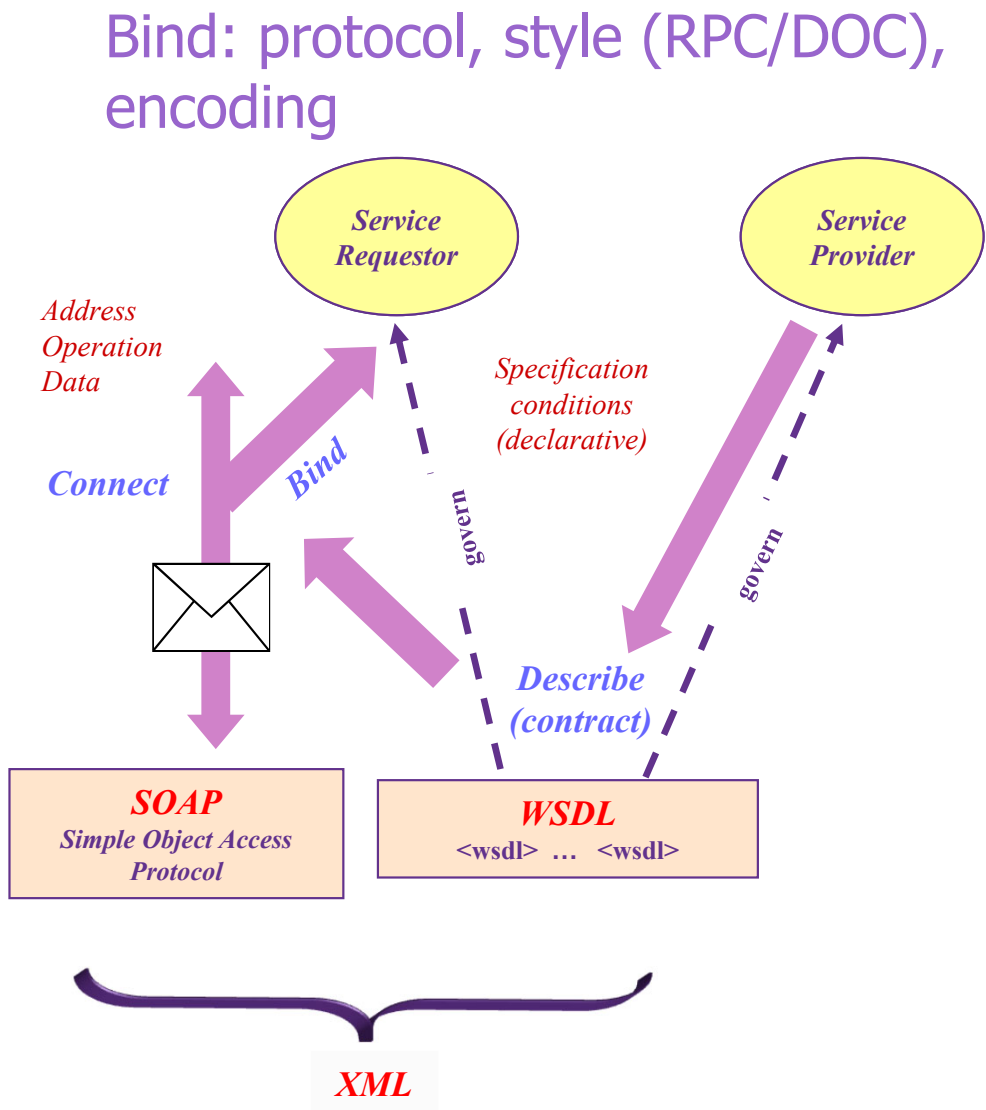
- *WSDL: Web Service Description Language*

Web Services Description Language

- The Web Services Description Language (WSDL) is the XML-based service representation language used to describe the details of the complete interfaces exposed by Web services and thus is the means to accessing a Web service.
 - For instance, neither the service requester nor the provider should be aware of each other's technical infrastructure, programming language or distributed object framework (if any).

WSDL as a contract

- A Web service description in WSDL is an XML document that describes the mechanics of interacting with a particular Web service.
- It is inherently intended to constrain both the service provider and the service requester that makes use of that service. This implies that **WSDL represents a “contract” between the service requester and the service provider**
- WSDL is platform and language-independent and is used primarily (but not exclusively) to describe SOAP-enabled services. Essentially, WSDL is used to describe precisely
 - **What** a service does, i.e., the operations the service provides,
 - **Where** it resides, i.e., details of the protocol-specific address, e.g., a URL, and
 - **How** to invoke it, i.e., details of the data formats and protocols necessary to access the service’s operations.



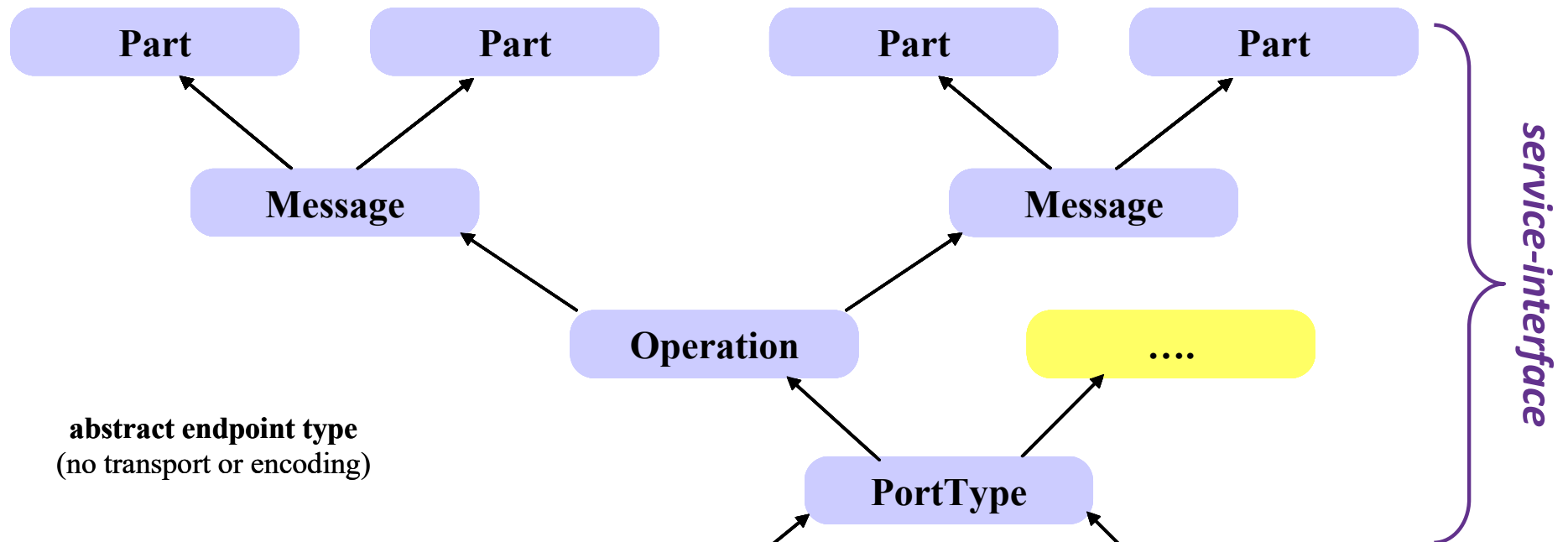
Structure of WSDL documents

- WSDL documents can be separated into distinct sections:
 - The **service-interface definition** describes the general web service interface structure. This contains all the operations supported by the service, the operation parameters and abstract data types.
 - The **service implementation part** binds the abstract interface to a concrete network address, to a specific protocol and to concrete data structures.
 - A web service client may bind to such an implementation and invoke the service in question.
- This enables each part to be defined separately and independently, and **reused** by other parts
- The combination of these two parts contains **sufficient information** to describe to the service requester how to invoke and interact with the web service at a provider's site.
 - Using WSDL, a requester can locate a web service and invoke any of the publicly available operations.

WSDL document content

- Abstract (interface) definitions
 - `<types>` data type definitions
 - `<message>` operation parameters
 - `<operation>` abstract description of service actions
 - `<portType>` set of operation definitions
- Concrete (implementation) definitions
 - `<binding>` operation bindings
 - `<port>` association of an network address with a binding
 - `<service>` contains one to many port(s)
- Also:
 - `<import>` used to reference other XML documents

WSDL interface elements hierarchy



```

<wsdl:definitions name="PurchaseOrderService"
  targetNamespace="http://supply.com/PurchaseService/wsdl"
  xmlns:tns="http://supply.com/ PurchaseService/wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <xsd:schema
      targetNamespace="http://supply.com/PurchaseService/wsdl"
      <xsd:complexType name="CustomerInfoType">
        <xsd:sequence>
          <xsd:element name="CusNamer" type="xsd:string"/>
          <xsd:element name="CusAddress" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="POType">
        <xsd:sequence>
          <xsd:element name="PONumber" type="integer"/>
          <xsd:element name="PODate" type="string"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="InvoiceType">
        <xsd:all>
          <xsd:element name="InvPrice" type="float"/>
          <xsd:element name="InvDate" type="string"/>
        </xsd:all>
      </xsd:complexType>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="POMessage">
    <wsdl:part name="PurchaseOrder" type="tns:POType"/>
    <wsdl:part name="CustomerInfo" type="tns:CustomerInfoType"/>
  </wsdl:message>
  <wsdl:message name="InvMessage">
    <wsdl:part name="Invoice" type="tns:InvoiceType"/>
  </wsdl:message>
  <wsdl:portType name="PurchaseOrderPortType">
    <wsdl:operation name="SendPurchase">
      <wsdl:input message="tns:POMessage"/>
      <wsdl:output message="tns:InvMessage"/>
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>

```

Abstract data type
definitions

Listing 1:
Example of WSDL
Interface definition

targetNamesapce:
All elements
defined will be
within that
namespace

Data that is sent

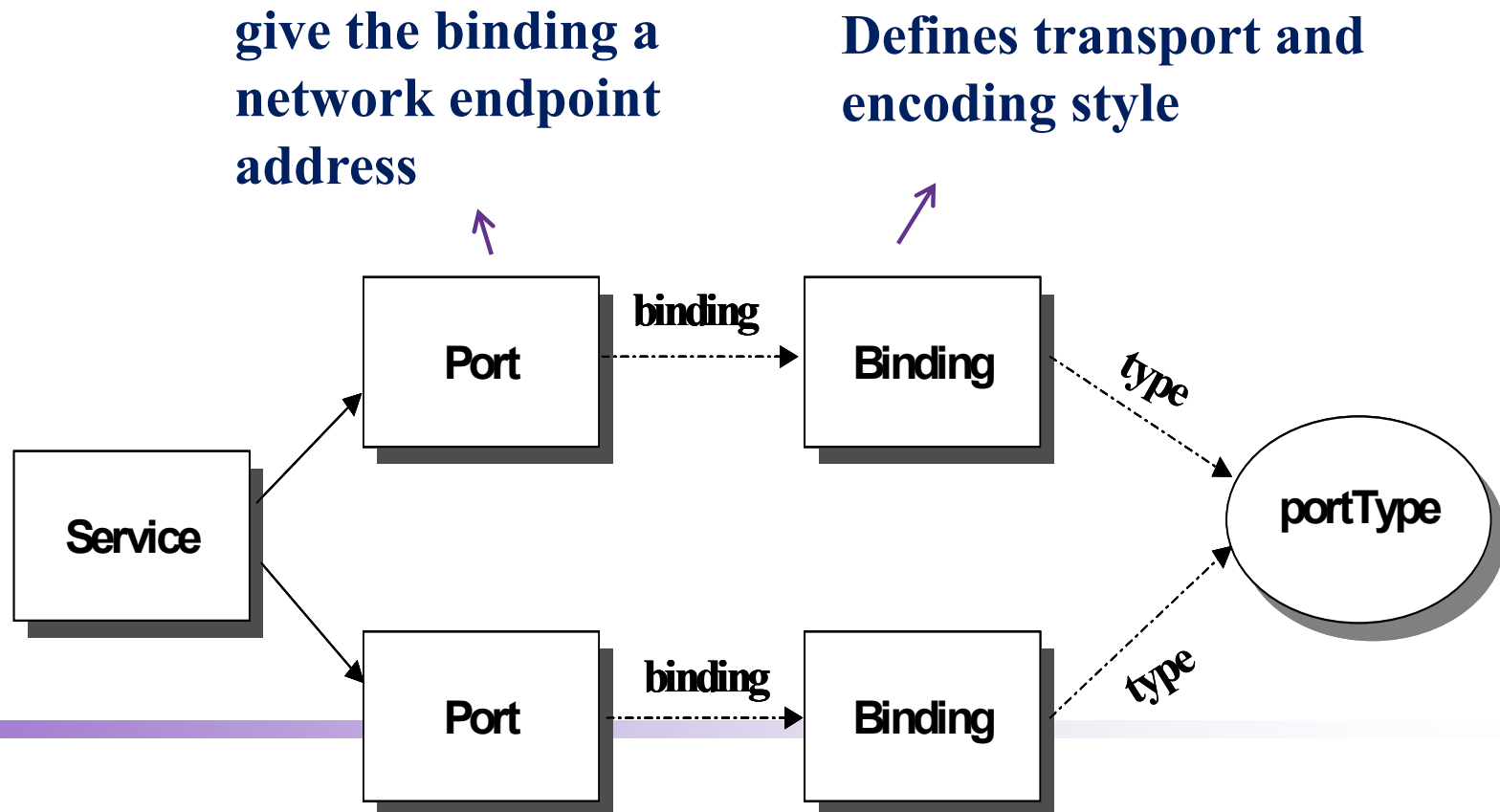
Data that is returned

Port type with
one operation

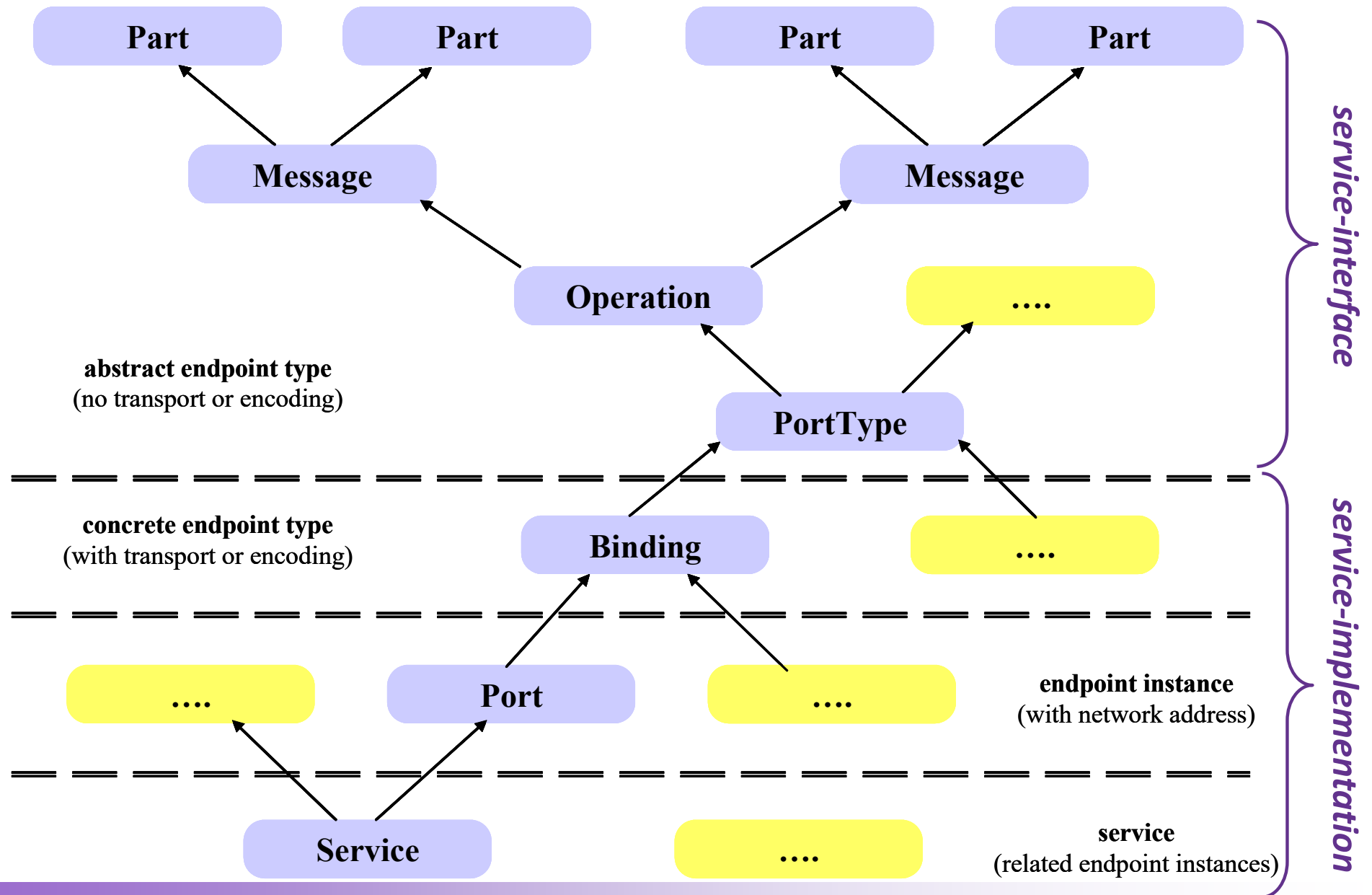
An operation with
request (input) &
response (output)
message

WSDL Implementation

- The purpose of WSDL is to specify a web service abstractly and then to define how the WSDL developer will reach the implementation of these services.
- The service implementation part of WSDL contains the elements **<binding>**, **<port>** and **<service>** and describes how a particular service interface is implemented by a given service provider.

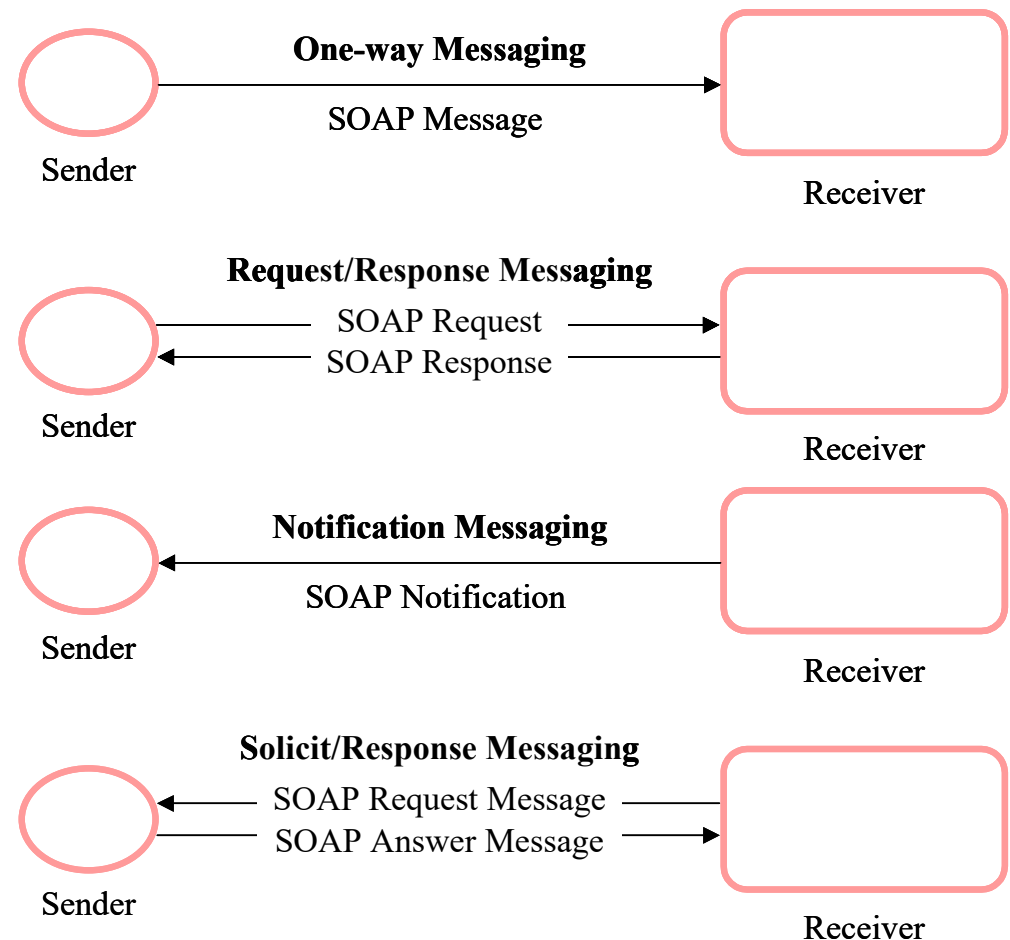


WSDL elements hierarchy



WSDL Message Exchange Patterns

- WSDL interfaces support four common types of operations that represent possible combinations of input and output messages
- The WSDL operations correspond to the incoming and outgoing versions of two basic operation types:
 - an incoming single message passing operation and its outgoing counterpart (“one-way” and “notification” operations),
 - the incoming and outgoing versions of a synchronous two-way message exchange (“request-response” and “solicit response”).
- any combination of incoming and outgoing operations can be included in a single WSDL interface
 - these four types of operations provide support for both push and pull interaction models at the interface level.



DL for Restful web services

- JSON Schema-based
 - RAML
 - OpenAPI
 - I/O Docs
 - Google APIs Discovery Service schemas
 - **APIS.JSON**: <http://apisjson.org/format.html>

- Similar to XML Schema, JSON Schema is for defining the structure and data types for a JSON document
- JSON Schema document itself is also in JSON
- Reference: <https://json-schema.org/learn/getting-started-step-by-step.html>

The Example JSON doc

- Suppose we want to use JSON to describe a product:

```
{  
  "productId": 1,  
  "productName": "A green door",  
  "price": 12.50,  
  "tags": [ "home", "green" ]  
}
```

Starting the schema

Type is the only constraint

```
{  
  "$schema": "http://json-schema.org/draft-07/schema#",  
  "$id": "http://example.com/product.schema.json",  
  "title": "Product",  
  "description": "A product in the catalog",  
  "type": "object"  
}
```

Defining the properties

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "http://example.com/product.schema.json",
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",
  "properties": {
    "productId": {
      "description": "The unique identifier for a product",
      "type": "integer"
    }
  },
  "required": [ "productId" ]
}
```

Nesting data structures

```
{  
  ...  
  "dimensions": {  
    "type": "object",  
    "properties": {  
      "length": {  
        "type": "number"  
      },  
      "width": {  
        "type": "number"  
      },  
    }  
  }  
}
```

The complete examp

- See `product.schema.json`

Developing RESTful web services

Benefits of REST Over SOAP

- REST allows a greater variety of data formats, whereas SOAP only allows XML.
- Coupled with JSON (which typically works better with data and offers faster parsing), REST is generally considered easier to work with.
- Thanks to JSON, REST offers better support for browser clients.
- REST provides superior performance, particularly through caching for information that's not altered and not dynamic.
- It is the protocol used most often for major services such as Yahoo, Ebay, Amazon, and even Google.
- REST is generally faster and uses less bandwidth. It's also easier to integrate with existing websites

REST interface constraints

According to Fielding, REST is defined by four interface constraints:

1. Identification of resources
 - Every interesting resource must have its own unique URI
2. Manipulation of resources through representations
 - Resources are manipulated using HTTP requests (GET, POST, PUT, DELETE)
3. Self-descriptive messages
 - Each identified resource can be returned in various formats (such as HTML, XML or JSON),
4. Hypermedia as the engine of application state
 - Communication must be state-less (follow HTTP)

Restful strategy of handling CRUD

- GET – Retrieve
- POST – Create
- PUT – Update
- PATCH – partial update
- Delete – delete

Examples

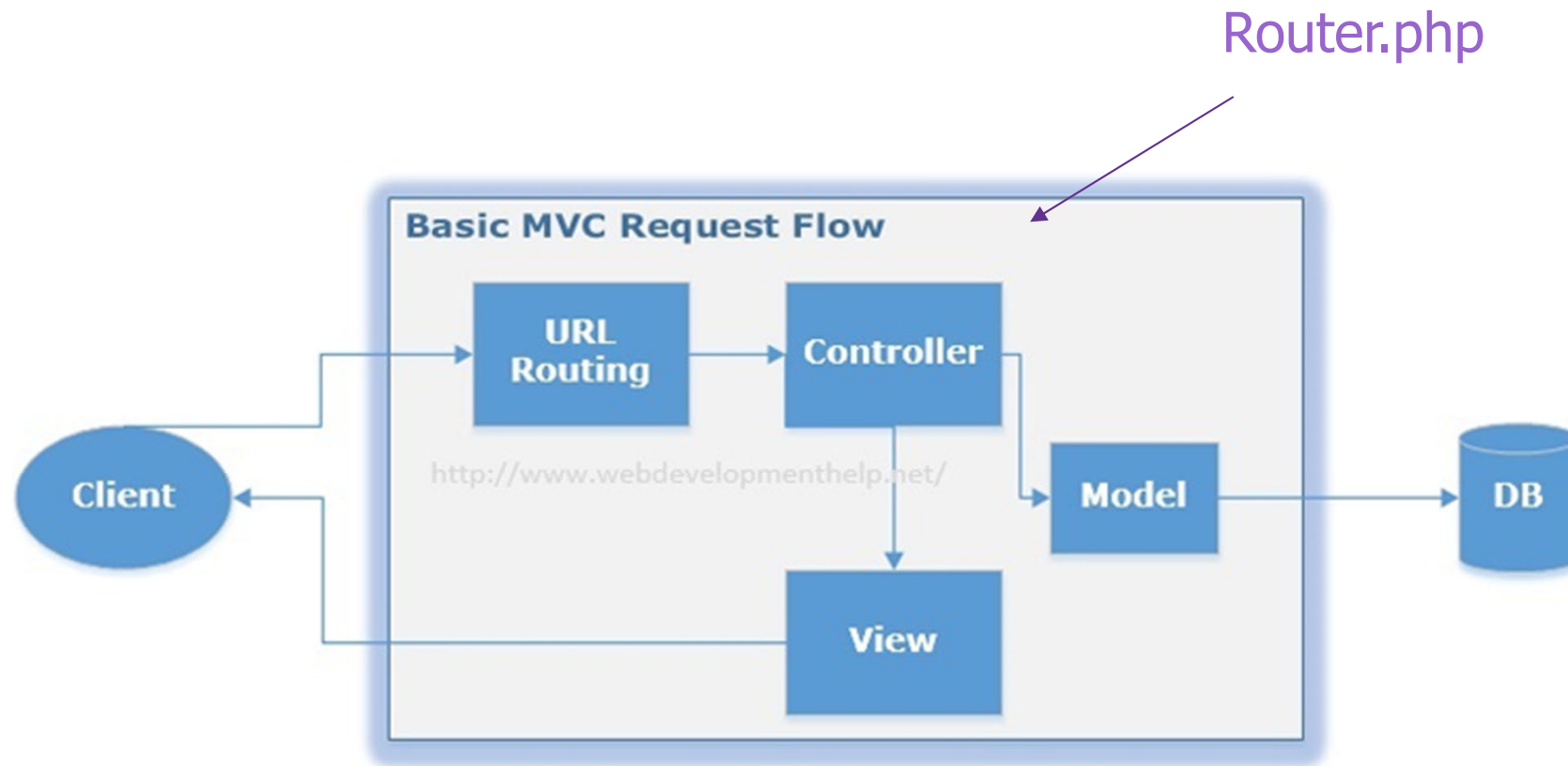
- GET /videos - Retrieves a list of videos (We won't use this one because it wouldn't make sense for our app.)
- GET /videos/tutorial - Retrieves the list of videos that match the keyword "tutorial"
- GET /videos/id/:id - Retrieves a specific video--in particular, the one whose id is *:id*
- POST /videos - Creates a new video on the server
- PUT /videos/:id - Updates video whose id is *:id*
- PATCH / videos/:id - **Partially Updates** video whose id is *:id*
- DELETE / videos/:id - Deletes the video whose id is *:id*

Restful Web service frameworks

For PHP

- *Slim*
- *Lumen*
- *Laravel*
- ...

SLIM Architecture



URL routing to PHP methods

```
3 <?php
4 // Routes
5 // API group
6 $app->group('/api', function () use ($app) {
7     // Version group
8     $app->group('/v1', function () use ($app) {
9         $app->get('/employees', 'getEmployees');
10        $app->get('/employee/{id}', 'getEmployee');
11        $app->post('/create', 'addEmployee');
12        $app->put('/update/{id}', 'updateEmployee');
13        $app->delete('/delete/{id}', 'deleteEmployee');
14    });
15 }
16 }
```

<http://localhost/api/v1/employees> => invoke getEmployees()
PHP method using HTTP GET method

Routing Design

#	Route	Method	Type	Full route	Description
1	/employee	GET	JSON	http://yourhost/api/v1/employees/	Get all employee data
2	/employee/{id}	GET	JSON	http://yourhost/api/v1/employee/1	Get a single employee data
3	/create	POST	JSON	http://yourhost/api/v1/create	Create new record in database
4	/update/{id}	PUT	JSON	http://yourhost/api/v1/update/21	Update an employee record
5	/delete/{id}	DELETE	JSON	http://yourhost/api/v1/update/2	Delete an employee record

Implement the functions

```
function getEmployees($response) {  
2   $sql = "select * FROM employee";  
3   try {  
4       $stmt = getConnection()->query($sql);  
5       $wines = $stmt->fetchAll(PDO::FETCH_OBJ);  
6       $db = null;  
7       return json_encode($wines);  
8   } catch(PDOException $e) {  
9       echo '{"error":{"text":'. $e->getMessage() .'}}';  
1  }  
0 }  
1  
1  
1  
1  
2
```

JSON Encode



Invoke Restful Web services

cURL
code

```
<?php
$url = 'http://yourhost/api/v1/employees/';

$curl = curl_init();
curl_setopt($curl, CURLOPT_URL, $url);
curl_setopt($curl, CURLOPT_RETURNTRANSFER, 1);
curl_setopt($curl, CURLOPT_PROXY, 'cache.aut.ac.nz:3128');
$pageContent = curl_exec($curl);
curl_close($curl);

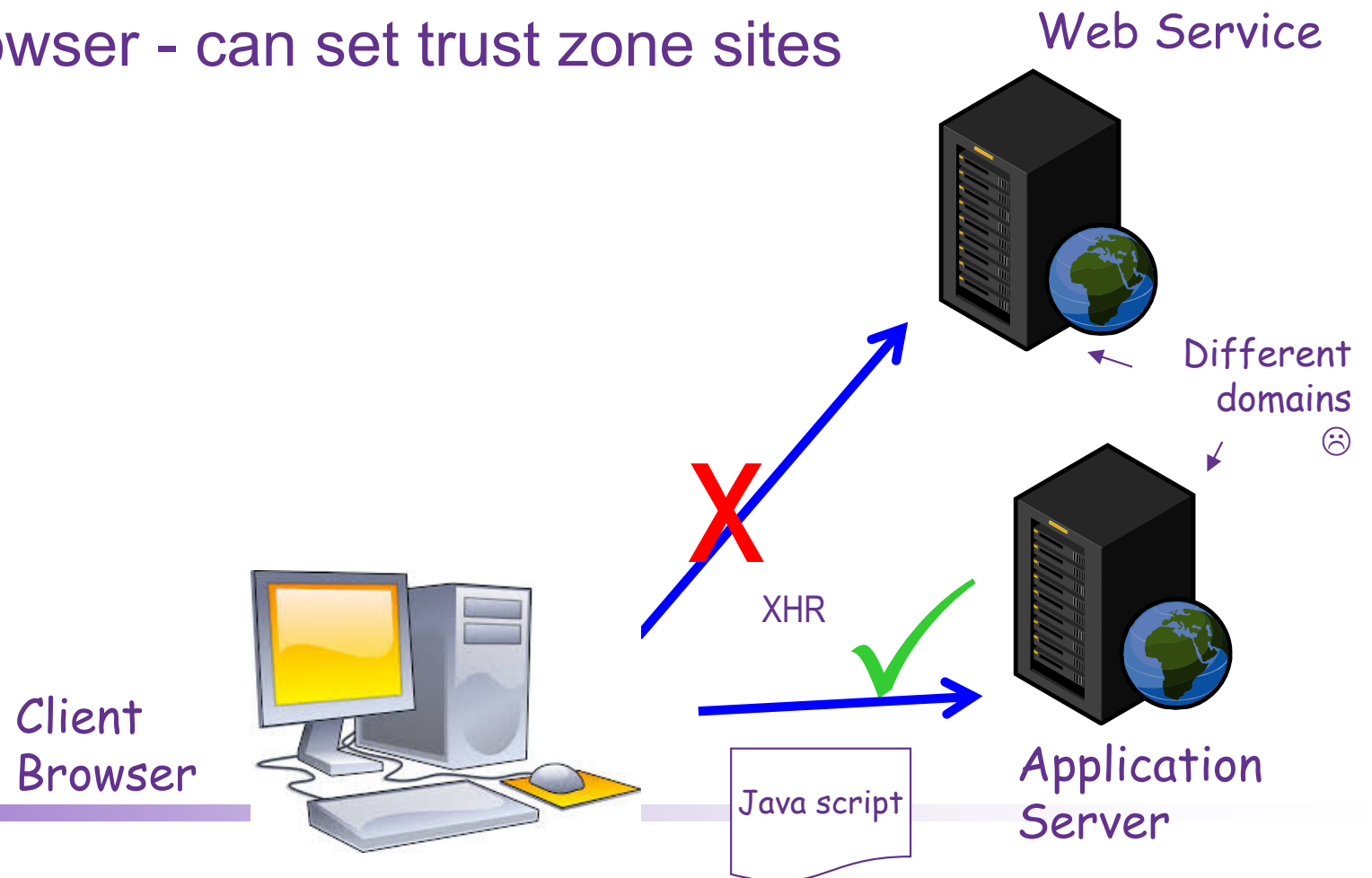
echo $pageContent

?>
```

proxy

Same Origin Policy

- Same origin policy
 - May not work across domains
 - sensible security restriction
 - Browser - can set trust zone sites



Same Origin Policy

- Browser considers two pages to have the same origin if *protocol*, *port* (if given), and *host* (*domain*) are the same.
- Example: Javascript from <http://store.company.com/dir/page.html>.

URL	Outcome	Reason
http://store.company.com/dir2/other.html	Success	
http://store.company.com/dir/inner/another.html	Success	
https://store.company.com/secure.html	Failure	Different protocol
http://store.company.com:81/dir/etc.html	Failure	Different port
http://news.company.com/dir/other.html	Failure	Different host

There are some solutions for cross-domain invocation...