ICOM 6034
Website Engineering

Dr. Roy Ho
Department of Computer Science, HKU

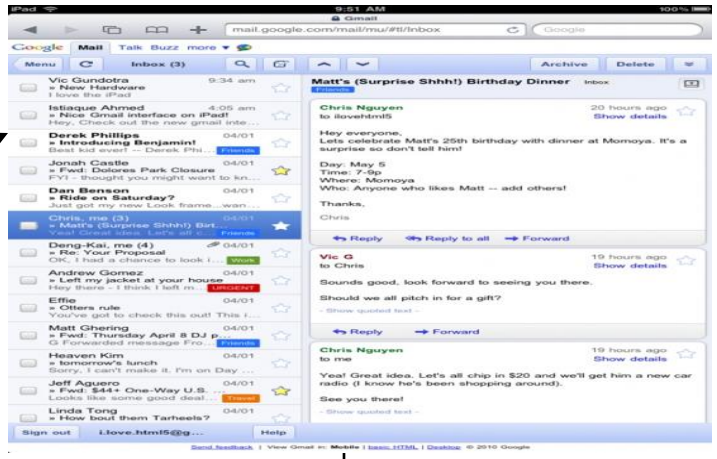
Session 7: Standard data formats and web API protocols (Part I)

ICOM6034: scope

Part 1 (done):

Websites becoming "web apps" -> more sophisticated, lots of interactions with users, "Web 2.0", etc.

Part 2 (done):



*Integration and interoperability issues -> how to reuse existing & remote **data**?*

Part 3 (sessions 7-9):

YouTube, Gmail, Amazon, online databases, Maps, updated event lists, YOUR websites, ……

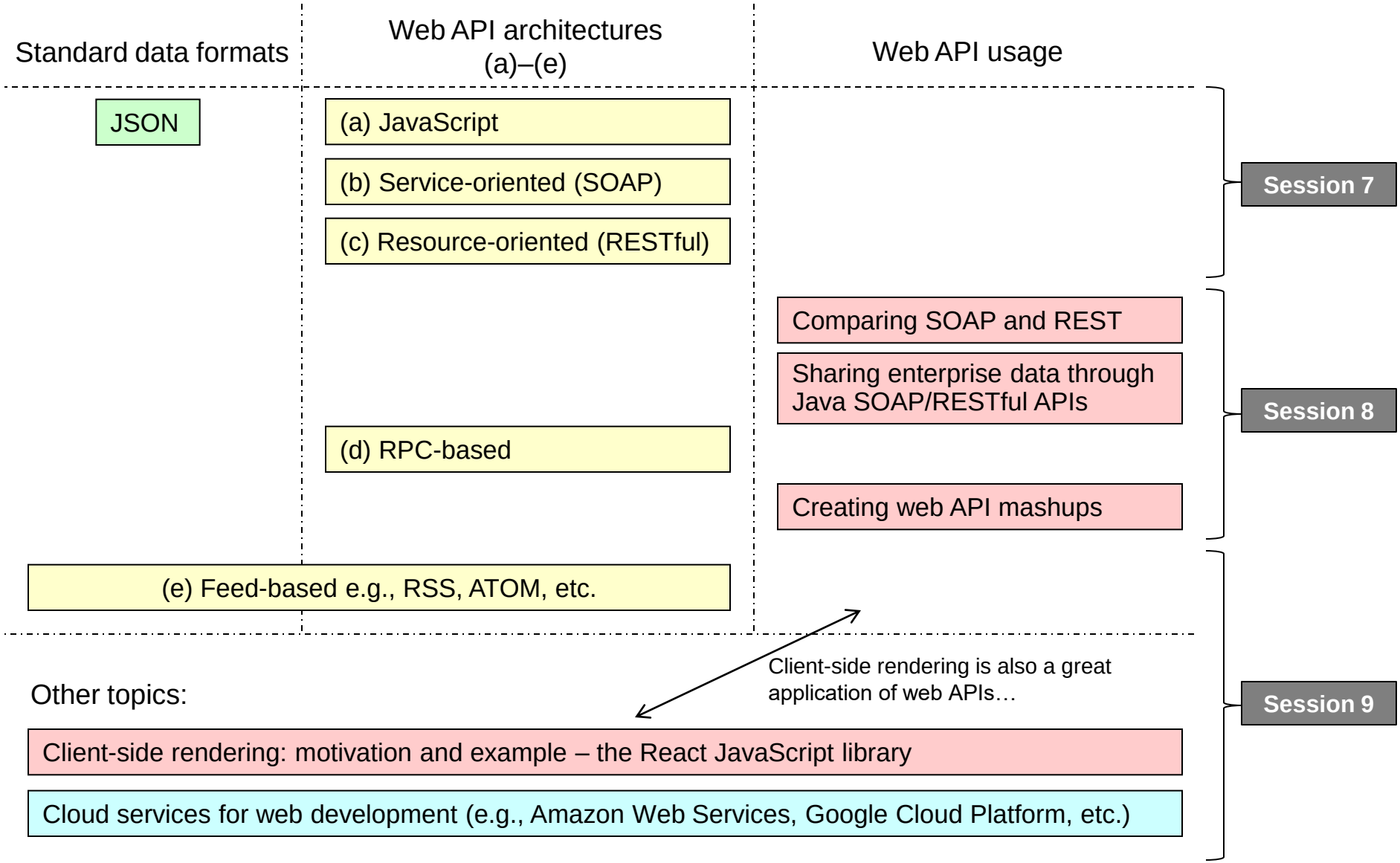
The web/cloud(s)

Optimizations

Part 4 (session 10):

You have a great website, how to make it loaded fast at users' devices, and most important… popular?

Organization of Part 3 (Sessions 7-9)



Session objectives

- JSON – a standard data format for data sharing
- Introduction to Web APIs and their main architectures
- Service-oriented and Resource-oriented web APIs
  - Service-orientation
    - Implementation of SOAP web APIs with Apache Axis
  - Resource-orientation
    - Design principles and implementation of RESTful web APIs
    - Modeling a RESTful APIs
    - Some good practices of designing RESTful APIs
- Lab 4A: Using web APIs at the client side

Web 2.0 revisited

- Evolved from “Web 1.0” with three trends:
  - Functionality enhancements – RIA, etc.
  - Socialization
  - Web data consolidation
    - Concerned about data sharing/exchange and website interoperability

Covered in Part 2

- Motivations to data sharing/exchange:
  - To reuse the well-developed, comprehensive contents on many “Web 1.0” sites (e.g., maps, database archives, etc.).
  - Data created in multiple websites can be combined, or transformed to form “value-added” web content. E.g., combining football match schedules (from different websites) and Google Calendar can allow clients to get informed of events… and many other examples.
  - Question: how to combine and reuse remote data?

Standard data formats and protocols

- "Combination" and "reuse" can only be possible if data sharing can be done effectively
  - We need **standard** protocols for data sharing/exchange
    - And, HTTP is too low-level for some complicated usage
    - => We need higher-level "web APIs" for supporting more dynamic data sharing
  - Also, HTML is designed for human's reading
    - Difficult to be "processed" -- e.g., difficult to extract news articles from a news website
    - => We need **standard** data formats
    - Examples of standard data formats: XML, JSON, feeds (e.g., RSS, ATOM, etc.)
- Lists of commonly used protocols and data formats: see www.programmableweb.com

JSON – a standard data format for data sharing

JSON – A text format and an alternative to XML

- JavaScript Object Notation
- A simple, textual data format
- Part of JavaScript (ECMA-262) standard
- JSON has the MIME type of "application/json"

- Easy to parse => great alternative to XML
  - Syntax much more concise and readable than XML
  - During the last decade, JSON has become much more popular than XML for sharing web data

- Encoding: strictly Unicode

- Data types in JSON:
  - Strings
  - Numbers
  - Booleans: true or false
  - Objects
  - Arrays
  - **`null:`** a value that isn't anything

Strings and numbers

- Strings:
  - Sequence of Unicode characters
  - Wrapped in `"double quotes"`
  - No “character” type
    - A character is a string with a length of 1
  - Backslash escape supported
- Numbers:
  - Supports integers, real numbers, and scientific notation
  - No **`NaN`** or **`Infinity`**
    - Use **`null`** instead

Objects

- Objects are unordered containers of key/value pairs wrapped in **{ }**
- **:** separates a key and its value
- **,** separates key/value pairs
- Keys are strings

```
{
    "name":      "Cargo",
    "packaged": true,
    "grade":     "A",
    "format": {
        "type":       "rect",
        "width":      1080,
        "height":     1020,
        "waterproof": false,
        "weight": 24
    }
}
```

Array

- Arrays are ordered sequences of values wrapped in `[]`
- `,` separates values

```
["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
   "Saturday"]
```

```
[
    [0, -1, 0],
    [1, 0, 0],
    [0, 0, 1]
]
```

- JSON does not care about array indexing.
  - Array indexing is determined by the programming language that processes JSON
  - E.g., for JavaScript, array indexes start from 0.

Handling JSON data in browsers

- JSON data can be converted to JS variables or objects very easily
- The magic: the JavaScript eval(***json_data***) method
- When applied to JSON, eval returns the described JavaScript object

Example: using JSON data received from AJAX's `responseText`

- `responseText` in an XMLHttpRequest contains the AJAX response in plain text
  - As compared to `responseXML` which contains XML data)

- Can be anything:
  - A JavaScript object/array in JSON
  - JavaScript code
  - HTML / text
  - …

Example: using `responseText` (in JSON)

```
{"CATALOG":
    {"CD":
        [
        {"TITLE":"Empire Burlesque",
         "ARTIST":"Bob Dylan",
         "COUNTRY":"USA",
         "COMPANY":"Columbia",
         "PRICE":"10.90",
         "YEAR":"1985"
        },{...}
        ]
    }
 }
```

```
// Get the responseText
txt = xhr.responseText;

// eval it to convert it back to a JS object
obj = eval(txt);

// Get the value "Empire Burlesque"
value = obj.CATALOG.CD[0].TITLE;
```

Comparison of JSON and XML

- Similarities:
  - Both are textual data formats
  - Both are language independent
  - Both can be used in Ajax

- Differences:
  - XML can be validated
  - JSON is less wordy
  - Many developers feel that JSON is easier to read and handle
  - JSON can be parsed by JavaScript's eval method
    - Using plain JavaScript to handle XML is less straightforward

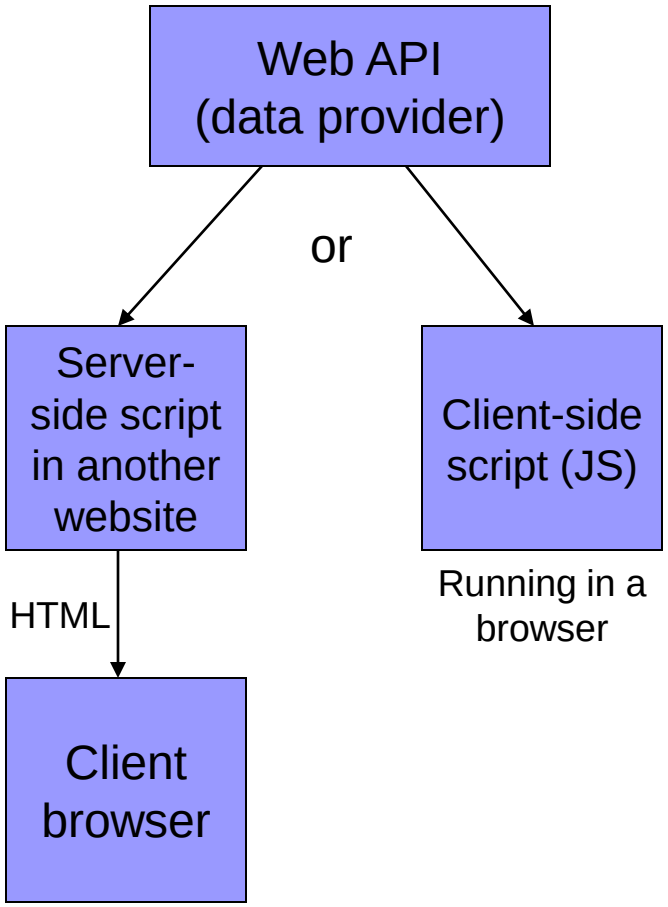
Introduction to Web APIs

Web APIs

- Application Programming Interfaces (APIs) that can be “called” by remote programs over the web for data sharing
- “Web services”
  - The term “web APIs” is often used interchangeably with “web services”
- Examples:
  - E.g., client programs (or websites) can use Google Maps’s services by remotely “calling” the Google APIs
  - Other examples: Amazon’s APIs, Facebook’s APIs, etc.
- Most APIs use standard protocols for the interactions between the data providers (i.e., the APIs) and the data consumers (the clients)
  - Example protocols: SOAP, RESTful, XML-RPC, JSON-RPC…
  - Most protocols internally use a standard data format (e.g., XML, JSON, etc.) for data exchange

Consumption of web services/APIs

- Web APIs can be “consumed” (i.e., called) by a server-side script (e.g., in PHP) or a client-side script (JS).
- Consider browser compatibility when choosing the protocols through which your web APIs are offered,
  - e.g., SOAP in general is easier to be processed at the server-side than by client-side, etc.
- Many large-scale API providers support both for best compatibility.



Five main architectures of web APIs

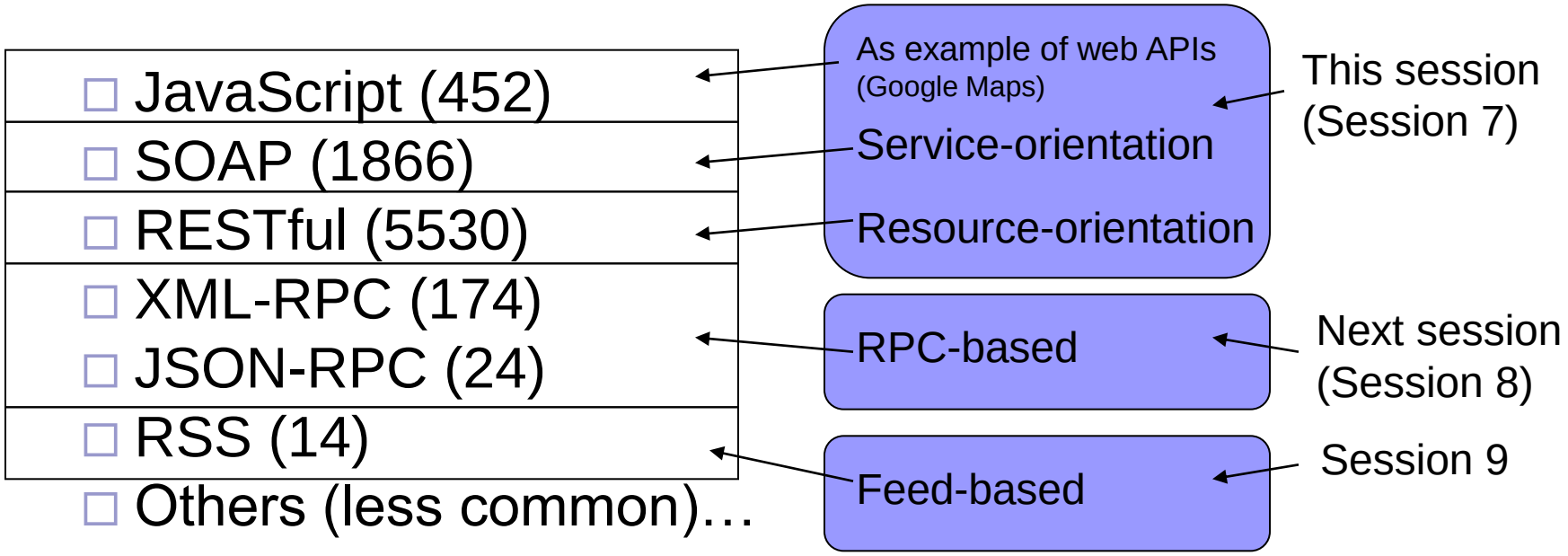
- (a) JavaScript: no fixed protocol; the data provider simply provides a JavaScript library that enables clients (mostly browsers) to connect to the web API through ordinary GET or POST (mostly through AJAX). Can use any data format for data sharing

We will use Google Maps as an example of this architecture

- (b) Service-oriented architecture: model a web API as a "service" – a service is a collection of operations/functions that can be invoked, and return the output (of the service) to the clients.
  - Main model: SOAP-based web APIs
- (c) Resource-oriented: model a web API as a "resource" – a "noun" that reflects a real-life object (a book, calendar, photo, etc.) with a limited set of operations (create, read, update, delete).
  - Main model: RESTful web APIs
- (d) RPC-based: a simpler form of the service-oriented architecture - model a web API as a "Remote Procedure Call" (RPC) service. Servers and clients exchange call parameters and return values, through standard data formats such as XML, JSON, etc.
- (e) Feed-based: the website provides (or accepts) a feed (e.g., RSS/ATOM) for data sharing
- There are other architectures/protocols, but are less commonly used or more application-specific (e.g., XMPP for IM applications, etc.)

Some statistics

- Programmableweb.com maintains a list of public web APIs:

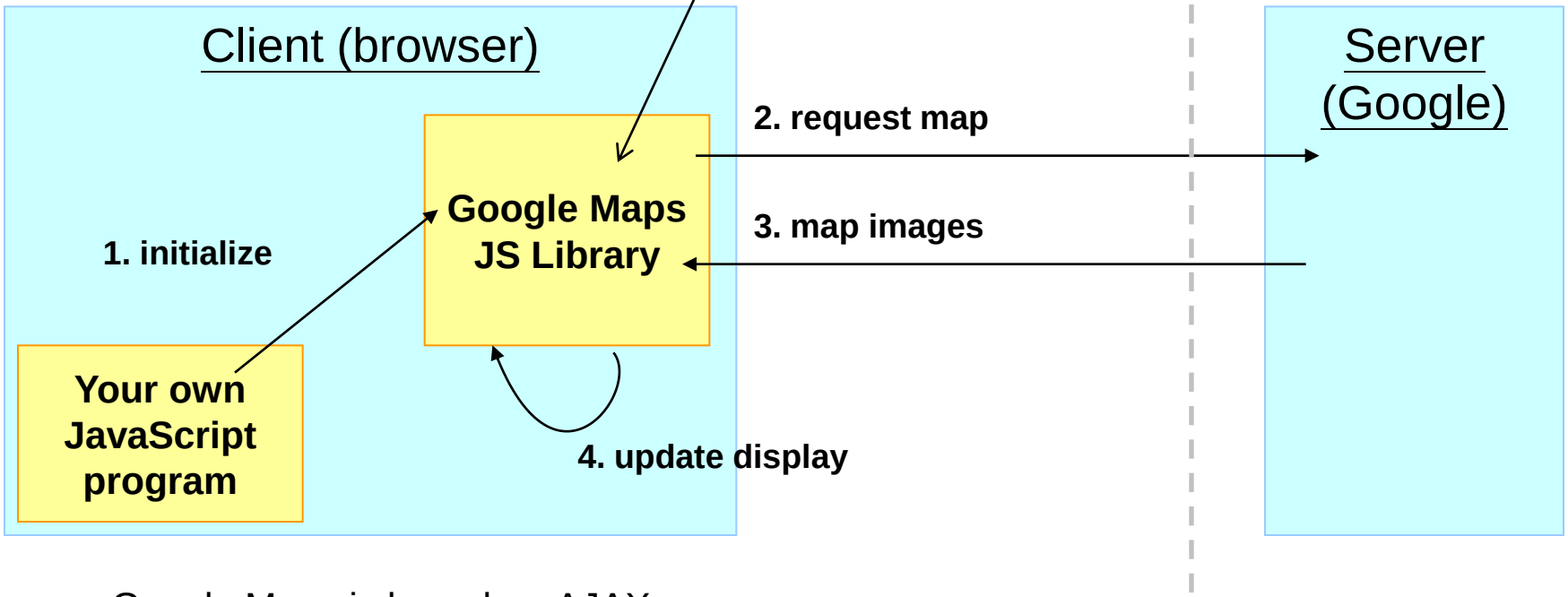


An example of web API protocol (a) – JavaScript: Geocoding in Google Maps

The 1st web API architecture – "JavaScript": the API provider simply provides a JS library for clients to use in browsers. It may not (but it can) use any specific standard protocols (e.g., SOAP, REST, etc.) in its underlying implementation.

Google Maps

- The Google Maps API provides library functions for displaying a map in a web page, and geocoding.
- Geocoding is the process of converting a physical address (like a street address) into geographic coordinates (i.e., latitude and longitude), which can be used for placing markers on a map.



- Google Maps is based on AJAX
  - Maps are retrieved asynchronously from Google while the user is interacting with the web page
  - => we have to register callback functions to handle the incoming data when it arrives

Geocoding in Google Maps

(Note: to keep this example short, let's assume that the initial map has been created. We will show how to create an initial map in Session 8 when we introduce "web API mashups".)

After the initial map is created, we can place a marker at a specific location on the map using geocoding.


```
function callback2(){
    var myHtml = "HKU using GClientGeocoder!";
    marker.openInfoWindowHtml(myHtml);
}
function callback1(latlong){
    if (latlong == null) {
        document.getElementById("msg").innerHTML = "address not found";
    }
    else {
        var marker = new GMarker(latlong);
        GEvent.addListener(marker, "click", callback2);
        map.addOverlay(marker);
    }
}
var geocoder = new GClientGeocoder();
geocoder.getLatLng("The University of Hong Kong, Hong Kong", callback1);
```

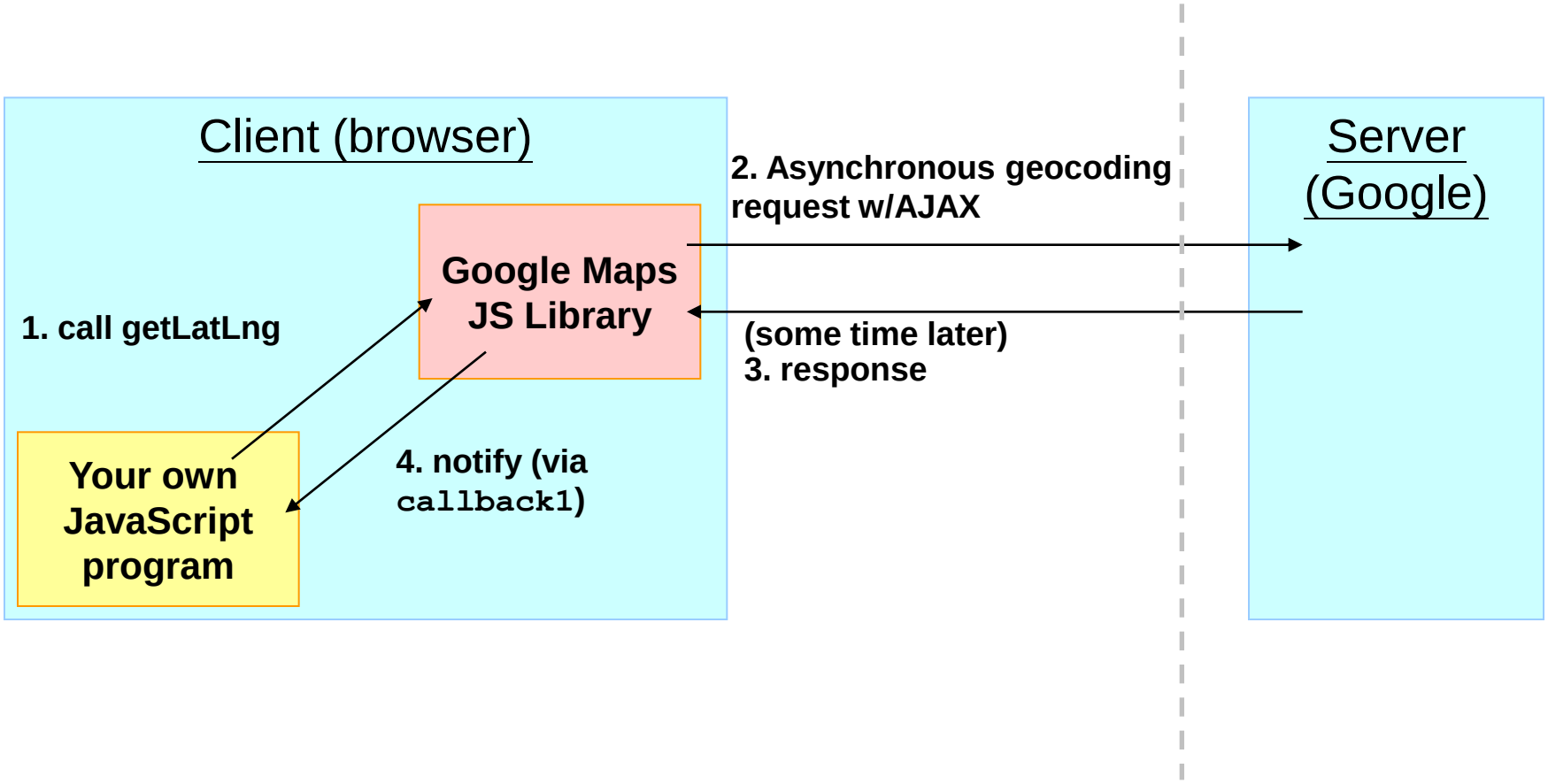
Library functions provided by the Google Maps Library

`callback2` would be called when `click` event happens

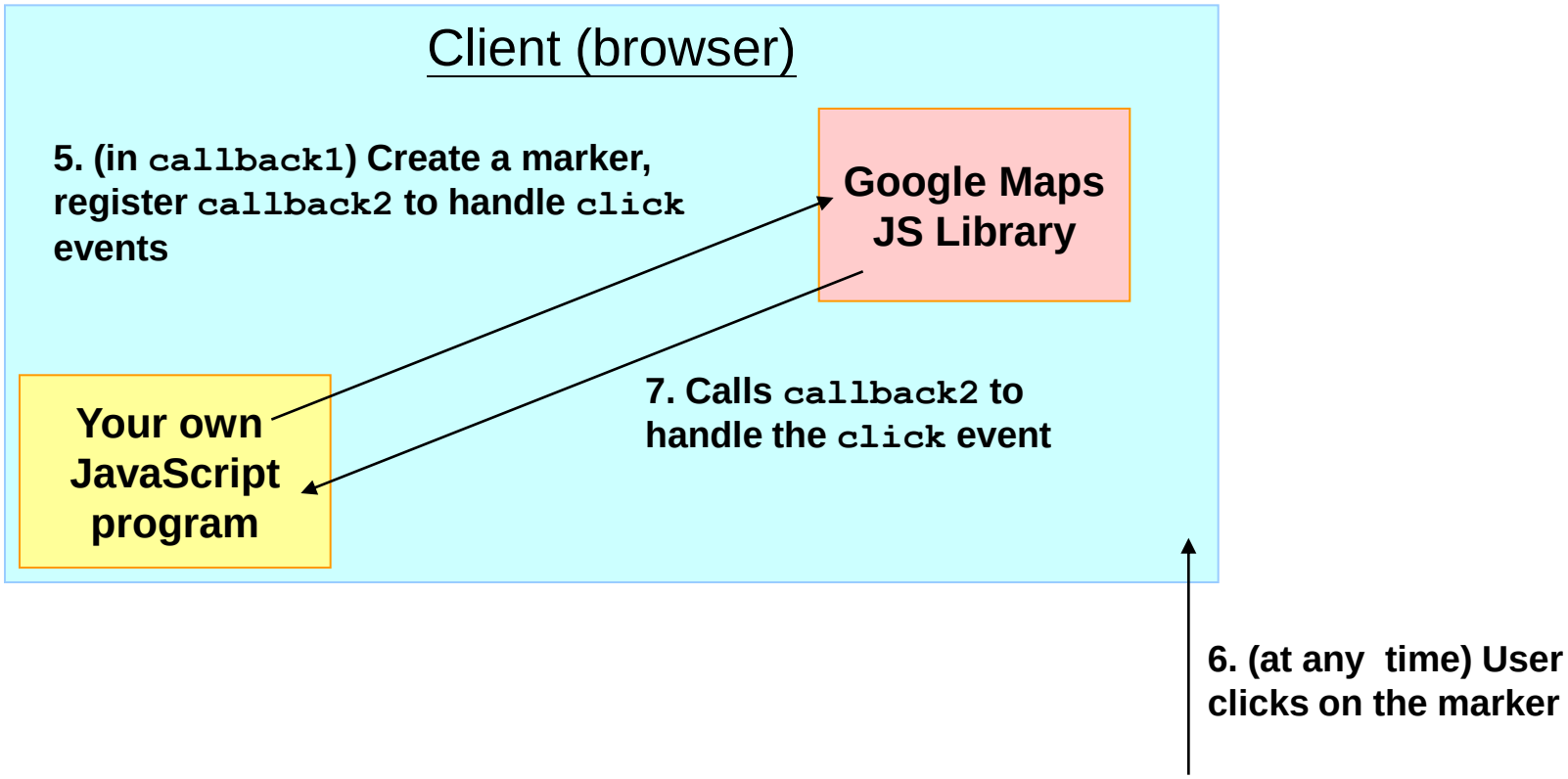
`geocoder.getLatLng` (latitude & longitude) uses `AJAX` to submit requests to Google to retrieve maps

calls `callback1` when `geocoder.getLatLng` receives the result from Google

Drawing a marker in Google Maps

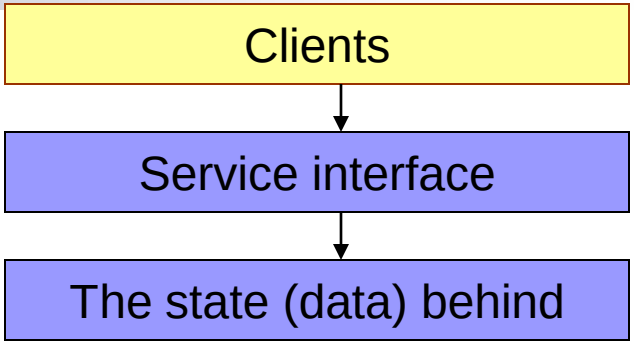


Opening an Info Window in Google Maps



Web API protocol (b): Service orientation and SOAP-based web APIs

Service orientation



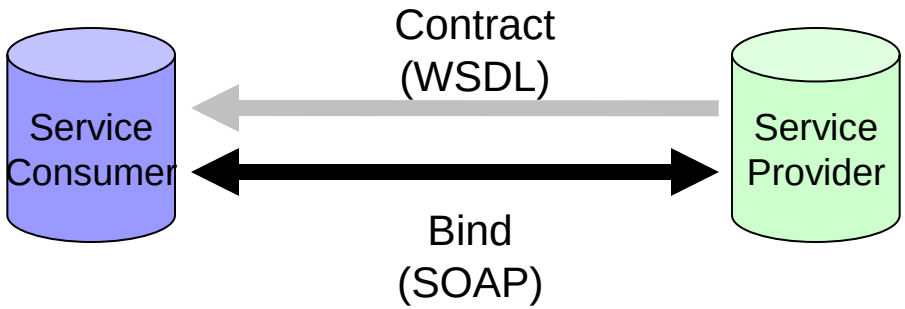
- A web API is modeled as a “service”
- A service is an interface of the state (data) behind; we don’t know the data unless the data is retrieved through the interface
- All clients can see is the interface which defines the available “operations”.
  - E.g., for a bookstore’s API, the interface may include operations like getBook(), getBookReviews(), newBook(), etc.
- The API clients do not have to know any implementation details of the API apart from the agreed interfaces.
- The interface itself is stateless (i.e., it does not have data/state in itself)
  - So, the interface can be easily replicated
  - => A client can use any service providers as long as they share the same interface => better scalability and fault resilience

Contract and SOA

- The service interface is usually defined in a contract
  - The contract describes the messages accepted (input) and sent out (output) by the service.
  - Clients **only** need to know the format of these messages, but not the API's internal logic.
- A system is said to adopt a service-oriented architecture (SOA) if it adopts this (service-oriented) model for data sharing.
- SOA is based on three operations:
  - {publish, discover, bind}
  - 1. A service provider publishes a contract to the network
  - 2. A service consumer discovers the contract, possibly through a URL
  - 3. The service consumer processes the contract and binds to (connects to and communicates with) the service according to the contract.

Building blocks: SOAP and WSDL

- The most common protocols for the service-oriented design are the "web service" (or "WS") protocol stack
- The stack uses WSDL (web service definition language) for contract specification, and SOAP (simple object access protocol) as a communication protocol
- The WS protocol stack:
  - Messaging
    - SOAP, XML-RPC, etc.
  - Service description
    - WSDL
  - Many other standards (seldom used)
    - WS-Addressing, WS-Security, BPEL, WSRF, etc.

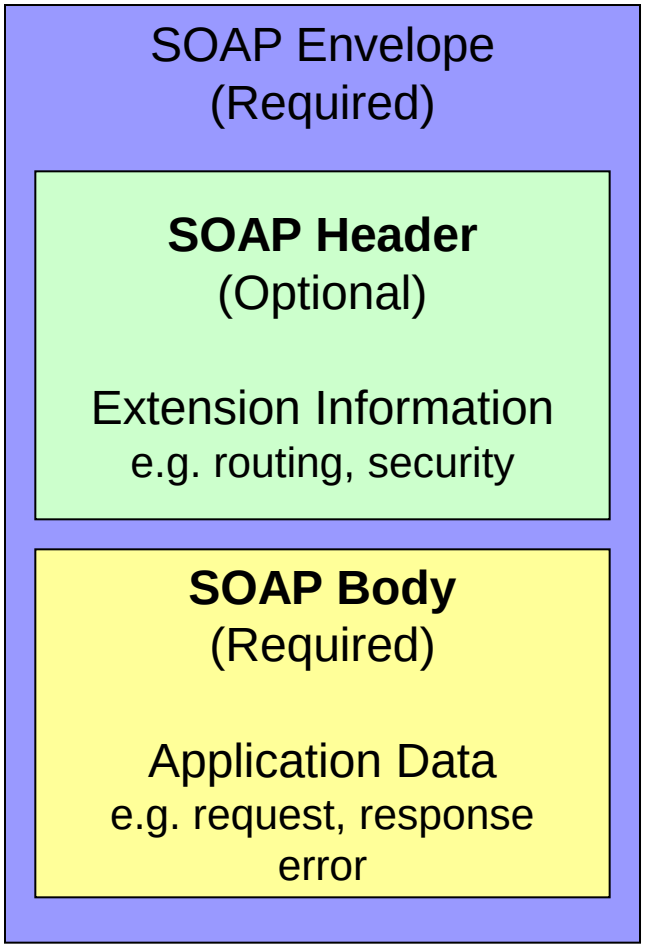


SOAP and WSDL are de-facto standards; the other standards are far less popular or obsolete due to their complexities.

SOAP

- Simple Object Access Protocol (SOAP)
  - Now just SOAP
- SOAP defines the XML messages sent between the API provider and consumer
- SOAP does not define the message exchange pattern
  - Instead, the pattern is defined in WSDL – as we will see later
- SOAP does not restrict the transport protocol
  - Normally HTTP, but can be any protocol that can carry a SOAP envelope
  - Again, the protocol is defined in WSDL

SOAP document



- Envelope
  - Top-level wrapper
- Header (optional)
  - Security and authentication information (WS-Security)
  - Routing information (WS-Addressing)
  - Resource information (WSRF)
  - …
- Body
  - Application data in XML
- Attachments (optional)
  - Additional non-XML data (e.g., binary, plain text, etc.)

SOAP request (in HTTP)

This SOAP msg is transferred on HTTP

```
POST /InStock HTTP/1.1
Host: www.stock.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
     xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
     soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

     <soap:Header>
          ... (optional header information)
     </soap:Header>

    <soap:Body xmlns:m="http://www.stock.org/stock">
          <m:GetStockPrice>
                    <m:StockName>IBM</m:StockName>
          </m:GetStockPrice>
     </soap:Body>
</soap:Envelope>
```

A custom namespace for avoiding naming conflicts

HTTP Header

SOAP Envelope

SOAP Header

SOAP Body

SOAP response (HTTP)

```
HTTP/1.1 200 OK
Content-Type: application/soap; charset=utf-8
Content-Length: nnn

<soap:Envelope
     xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
     soap:encodingStyle ="http://www.w3.org/2001/12/soapencoding">

     <soap:Header>
          ... (optional header information)
     </soap:Header>

     <soap:Body xmlns:m ="http://www.stock.org/stock">
          <m:GetStockPriceResponse>
                    <m:Price>119.52</m:Price>
          </m:GetStockPriceResponse>
     </soap:Body>
</soap:Envelope>
```

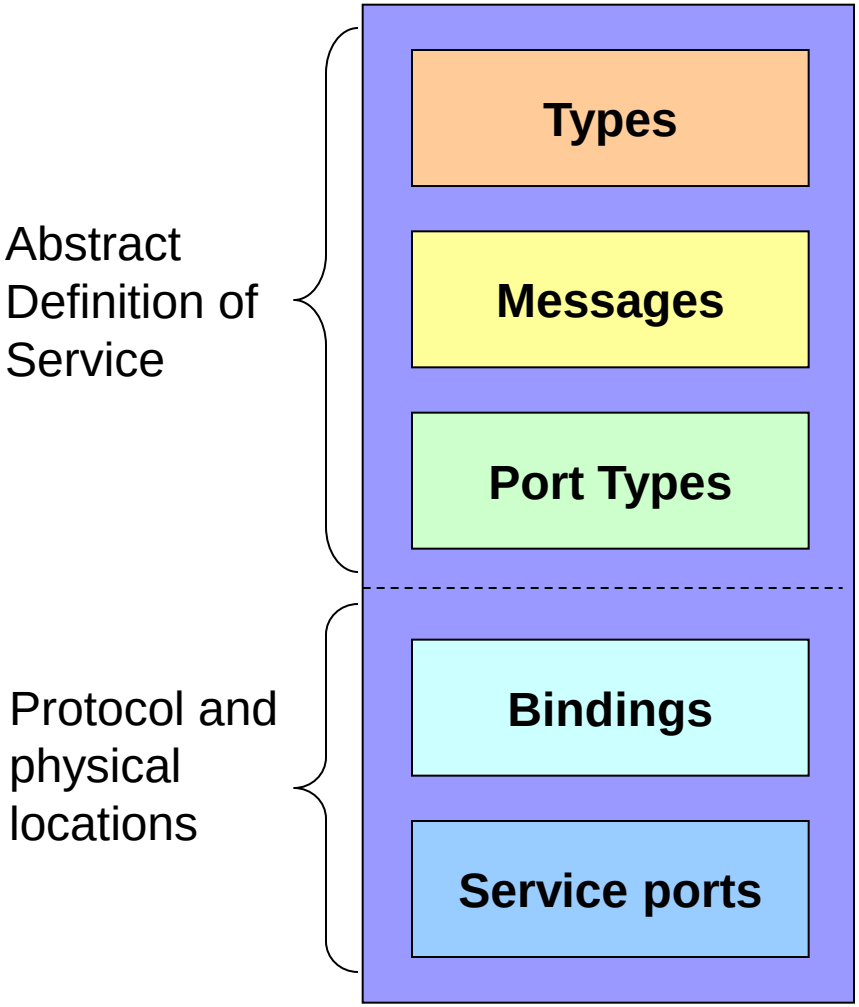
SOAP advantages

- Simple and lightweight
- Language- and OS-independent
  - Unlike RMI (Java) or DCOM (MS/Windows)
- Transport protocol independent
- Vendor support (an important advantage)
  - IBM, Microsoft, Apache, HP, Oracle, etc.
  - With better compatibility with backend data-stores (which are usually tied to vendors), SOA/SOAP is one of the most popular approaches for developing web-based enterprise applications

WSDL – the “contract”

- Web Service Definition Language (WSDL)
- A WSDL contract defines:
  - Where the service is located,
  - What the service can do, and
  - How to invoke the service
- Generally available as a single document
  - e.g. http://example.com/someService?WSDL
  - But: the WSDL doesn’t need to be hosted together with the web API, it can be hosted anywhere – for fault resilience

Structure of a WSDL document



- Types
  - What data types will be transmitted
- Messages
  - What messages will be transmitted
- Port Types
  - What business operations (functions) will be supported
  - And the input and output messages of each operation
- Bindings
  - The messaging protocol (e.g. SOAP) being used
- Service ports
  - The physical location (URL)
  - A service can have multiple, replicated ports

WSDL example – a “getBook” service

“Message” section:
This service will handle two types of message – getBookRequest and getBookResponse

“Port type” section:
One operation is available – “getBook”, which accepts getBookRequest as input and sends getBookResponse as output

“Binding” section:
The service is bound to use SOAP, where the input and output messages can be used literally

“Service port” section:
The physical location of the service, using the port binding “bookBind” defined above

```
<message name="getBookRequest">
 <part name="param" element="isbn"/>
</message>
<message name="getBookResponse">
 <part name="resp" element="book"/>
</message>
```

```
<portType name="bookPortType">
 <operation name="getBook">
  <input message="getBookRequest"/>
  <output message="getBookResponse"/>
 </operation>
</portType>
```

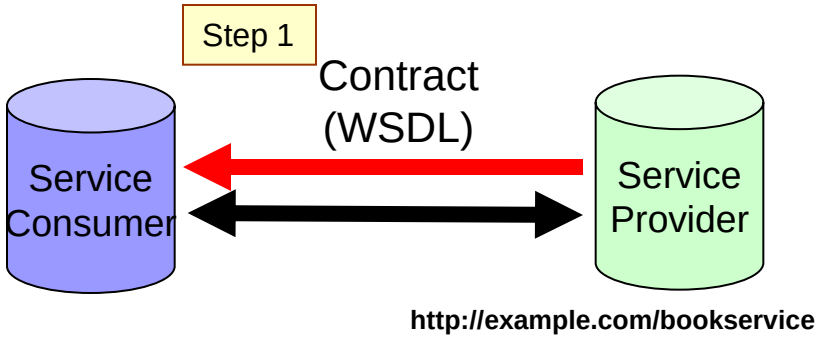
Abstract definition of the service

```
<binding type="bookPortType" name="bookBind">
 <soap:binding style="document"
   transport="http://schemas.xmlsoap.org/soap/http"/>
 <operation>
   <soap:operation soapAction="getBook"/>
   <input> <soap:body use="literal"/> </input>
   <output> <soap:body use="literal"/> </output>
 </operation>
</binding>
```

```
<service name="Hello_Service">
 <port binding="bookBind" name="bookPort">
  <soap:address
     location="http://example.com/bookservice"/>
 </port>
</service>
```

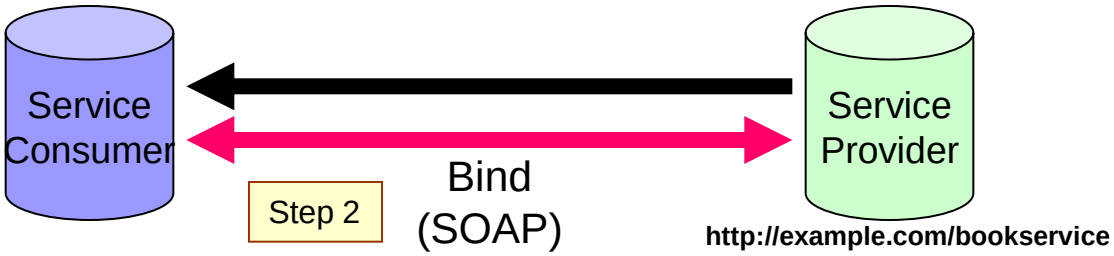
Protocol and physical locations

Putting it together




```
<message name="getBookRequest">
 <part name="param" element="isbn"/>
</message>
<message name="getBookResponse">
 <part name="resp" element="book"/>
</message>
<portType name="bookPortType">
 <operation name="getBook">
  <input message="getBookRequest"/>
  <output message="getBookResponse"/>
 </operation>
</portType>
<binding type="bookPortType" name="bookBind">
 <soap:binding style="document"
  transport=http://schemas.xmlsoap.org/soap/http
 <operation>
  <soap:operation soapAction="getBook"/>
   <input> <soap:body use="literal"/> </input>
   <output> <soap:body use="literal"/> </output>
 </operation>
</binding>
<service name="Hello_Service">
 <port binding="bookBind" name="bookPort">
  <soap:address
location="http://example.com/bookservice"/>
 </port>
</service>
```

Putting it together




```
<soap:Envelope
 <soap:Body>
  <getBookRequest>
   <isbn>0004702670</isbn>
  </getBookRequest>
 </soap:Body>
</soap:Envelope>
```

SOAP request

```
<soap:Envelope
 <soap:Body>
  <getBookResponse>
   <book>
    <title>Harry Potter</title>
    <author>J.K. Rowling</author>
    <date>2005-10-05</date>
   </book>
  </getBookResponse>
 </soap:Body>
</soap:Envelope>
```

SOAP response

Points to note

- SOAP and WSDL are used by many web APIs
  - Any client and any server can communicate as long as they "speak" SOAP and WSDL
  - However: SOAP APIs are more commonly consumed by server side scripts
    - Less easy for browsers to handle SOAP/WSDL unless a library is provided by the web API provider

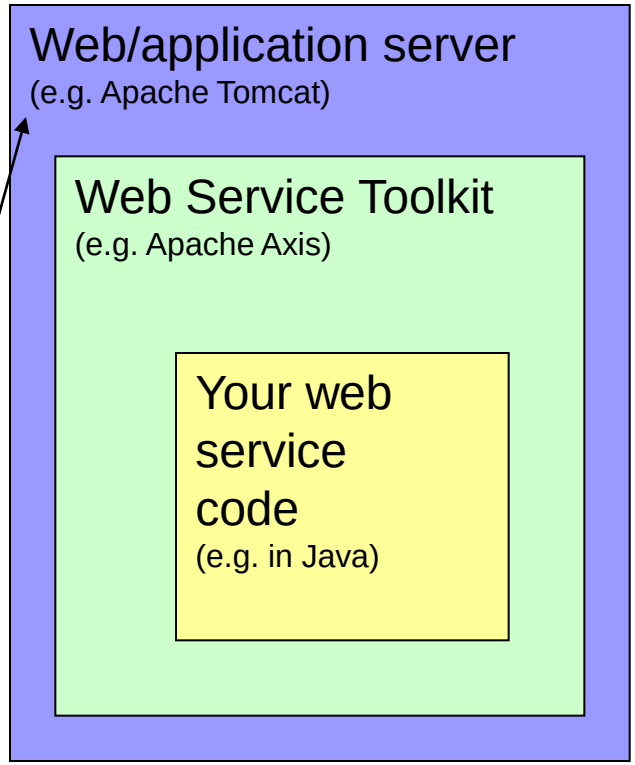
- SOAP-based web APIs are more commonly implemented in Java/.NET than the other languages (e.g., PHP, Ruby, etc.), probably because:
  - The SOAP+WSDL standards were first implemented in Java (then .NET followed)
  - SOAP-based APIs are more popular in the enterprise domain where Java (Enterprise) and .NET are commonly used.

- It is possible to write server-side programs to create a SOAP API manually, from scratch. However, for easy development, one should always use a web service toolkit instead of developing a SOAP API (or the WSDL document) from scratch
  - E.g., Apache Axis / Axis2

Implementing SOAP APIs with a web service toolkit

Web service toolkit

- With a Web Service Toolkit
  - XML, SOAP, WSDL and HTTP are all hidden from developers
  - Developers can focus on the application logic, not protocol/messaging details
- A Web Service Toolkit:
  - Automatically generates WSDL (e.g. from Java class)
  - Converts incoming SOAP requests into function calls (e.g. Java function call)
  - Converts outputs from functions into SOAP responses
- A web server (or an application server) hosts the web service, and handles all HTTP communication



Common web service toolkits

- Apache Axis/Axis2
  - Java, C++
  - Support Java application servers like Apache Tomcat
- Apache CXF – can handle multiple protocols (e.g., SOAP, REST, or even CORBA)
- JAX-WS and the Metro stack
  - “Official” Java technologies for SOAP APIs
  - More on this in Session 8
- Microsoft .Net
  - Visual Basic, C#, Perl…
  - Windows
  - Microsoft Internet Information Server (IIS)

Apache Axis/Axis2

- An open-source SOAP engine
- A framework for creating SOAP clients and servers
- Can be hosted within traditional application servers, e.g., Apache Tomcat

- Supports auto-generation of WSDL and SOAP messages, so developers can focus on application logic instead of these protocol details

- To run Axis, you need to install:
  - Apache Tomcat
    - http://tomcat.apache.org/
  - Java
  - Apache Axis
    - http://axis.apache.org/axis2/java/core/

Example: a calculator service

- The web service code
- e.g. **`Calculator.java`**

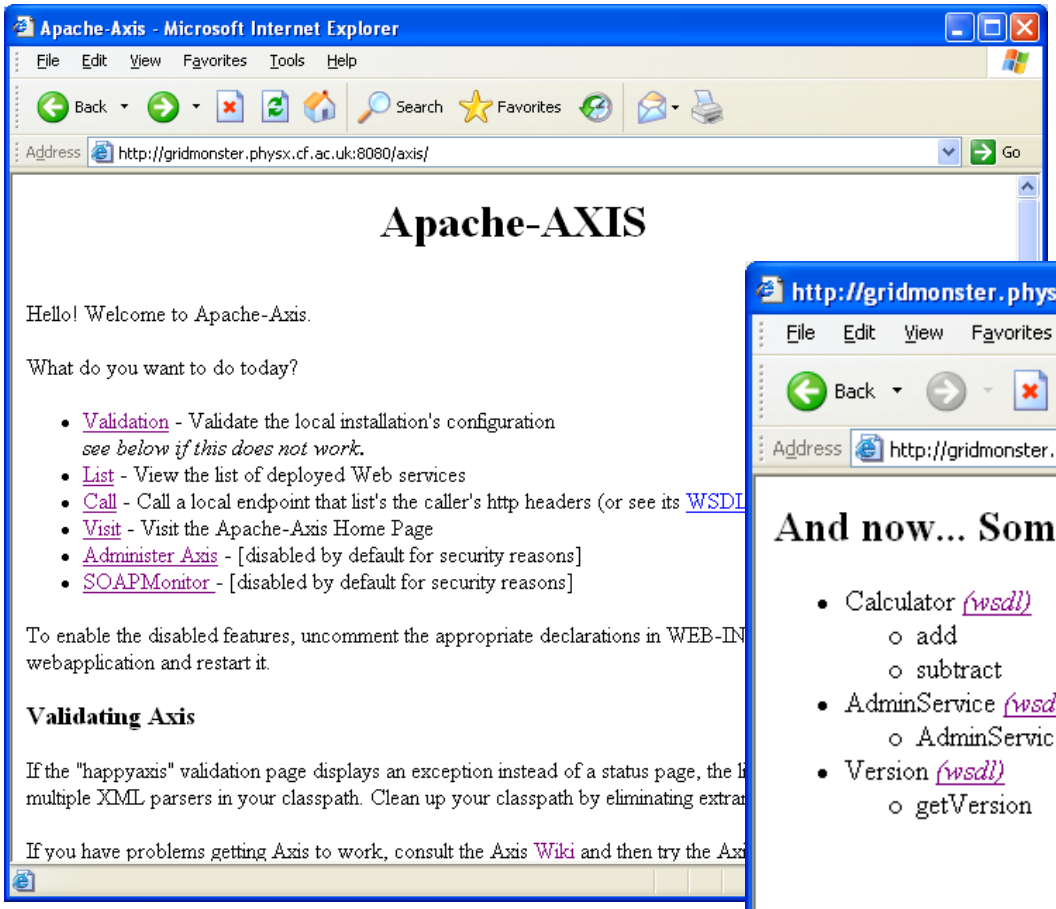
This is just an example showing how a web service toolkit works; a real web API should provide more meaningful functions or data.

```
public class Calculator {

    public int add(int i1, int i2)
        { return i1 + i2; }

    public int subtract(int i1, int i2)
        { return i1 - i2; }

}
```

Deploying the service

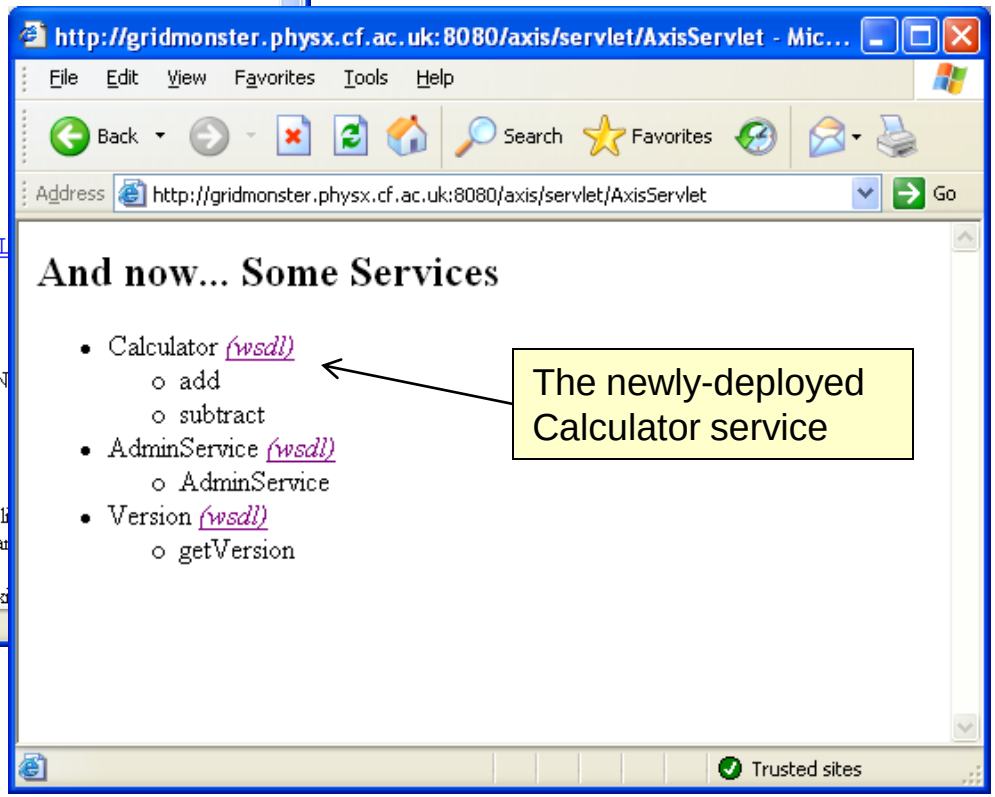
- Apache Axis supports instant deployment
  - Rename .java file as **`.jws`** file
    - .jws = “Java Web Service”
  - Copy **`.jws`** file into **`webapp`** directory
  - That’s it…
    - Axis would handle compilation, WSDL generation, etc.

AXIS Server

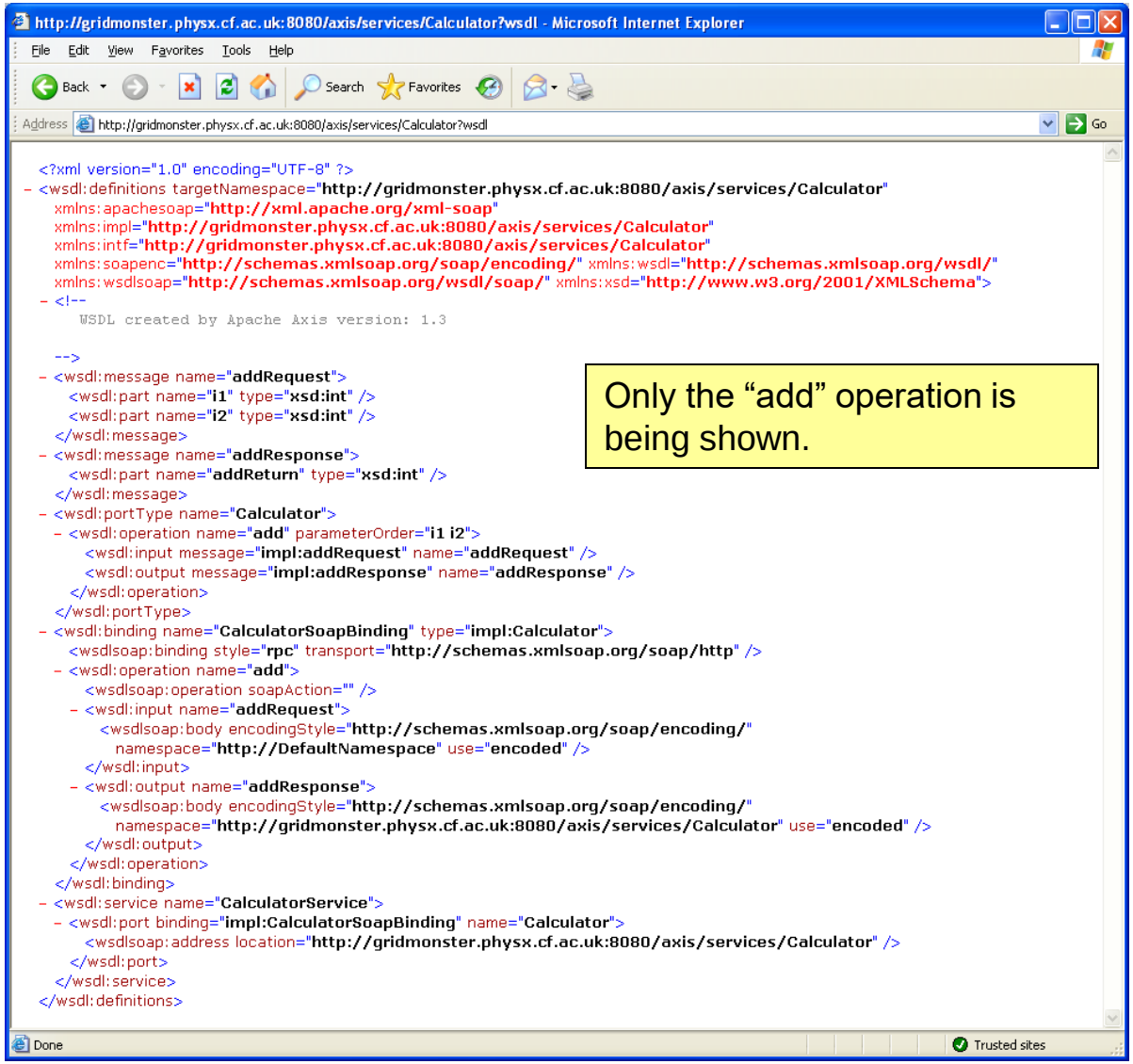


Welcome screen of Axis

The list of web services being hosted



The generated WSDL file



Only the “add” operation is being shown.

Development of client programs

- Apache Axis provides tools for generating client stubs (based on the WSDL) for developing client programs

- Step 1: Obtain the WSDL
  - Generally at web service URL + `?wsdl`
    - e.g. `http://<host>/axis/services/Calculator?wsdl`

- Step 2: Generate stubs
  - e.g. use WSDL2Java to create client stubs/bindings
    - e.g. `java org.apache.axis.wsdl.WSDL2Java Calculator.wsdl` generates:
    - `Calculator.java` (interface of calculator)
    - `CalculatorService.java` (the service interface)
    - `CalculatorServiceLocator.java`

      (service factory, implements `CalculatorService.java`)
    - `(Some other supporting files which implement the protocol communication and data conversion)`

Generated client stubs

- **Calculator.java**

Only the interface is shown. The detailed generated code is omitted for clarity.

```
public interface Calculator extends Remote {

    public int add(int i1, int i2) throws RemoteException;

    public int subtract(int i1, int i2) throws RemoteException;

}
```

- **CalculatorService.java**

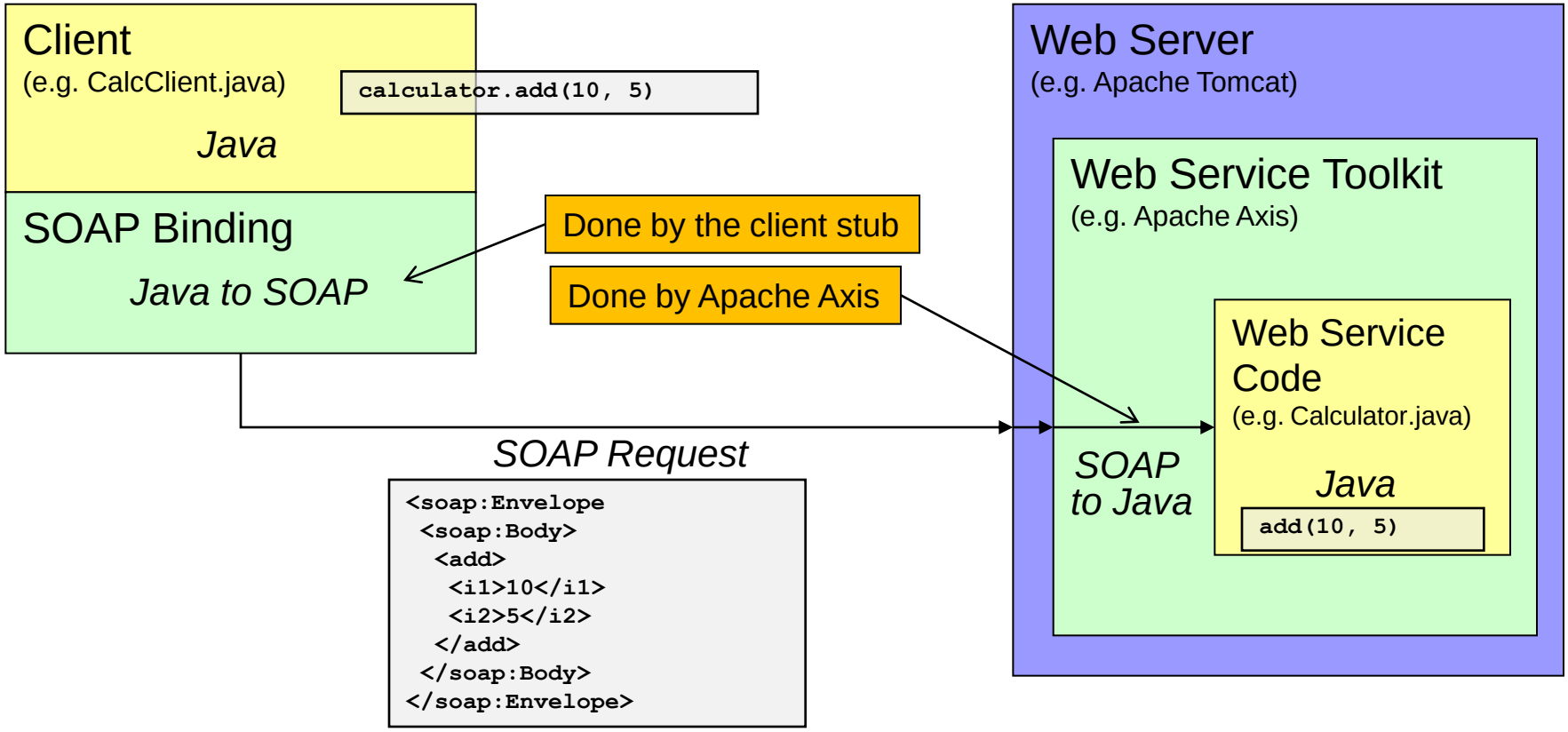
```
public interface CalculatorService extends Service {

    public String getCalculatorAddress();

    public Calculator getCalculator() throws ServiceException;

    public Calculator getCalculator(URL portAddress) throws ServiceException;

}
```

Invoking the service

- Step 3: Write client code

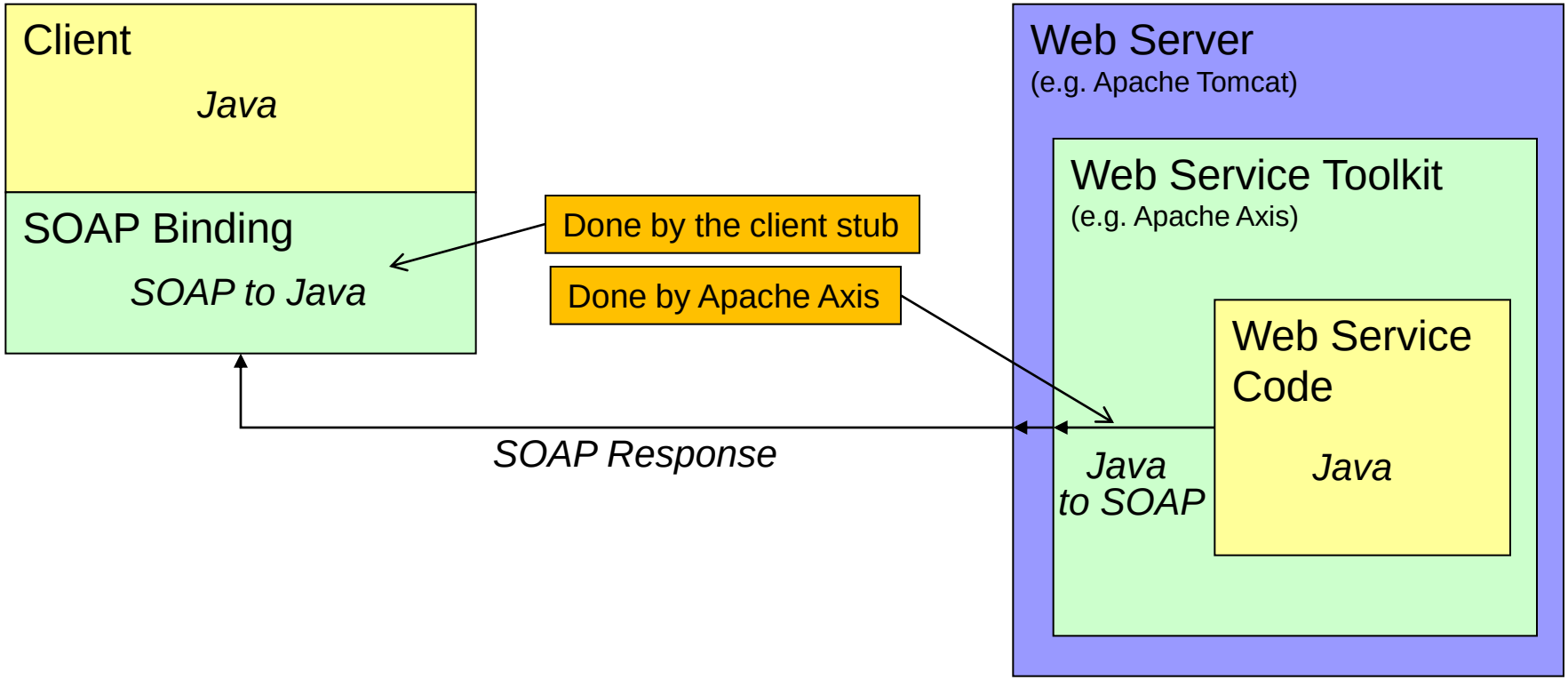
```
import Calculator.*;

public class CalcClient {

   public static void main(String[] args) {
      try {
         CalculatorService service = new CalculatorServiceLocator();
         Calculator calulator = service.getCalculator();

         int res = calculator.add(10, 5);

         System.out.println("Result=" + res);
      } catch(Exception except) {
         except.printStackTrace();
      }
   }
}
```

Complete picture



Web service response

Result: any clients can communicate with the web API as long as they obtain a copy of the WSDL document.



Web API protocol (c): resource orientation and RESTful web APIs

Resource orientation

- Resource orientation
  - Everything that can be named (i.e., it is a “noun”) is a resource.
  - Unlike SOA, where operations are defined in the contract (e.g., getBook() on p.37),
    - A resource has no implied/embedded operations.
    - The operations are not defined by the API developers either.
  - Instead, the operations are defined by the protocol through which the resource is made available
    - E.g., if a resource is located at http://example.com/abc, then the operations allowed on that resource would be defined by the HTTP protocol, i.e. GET, POST, PUT, DELETE, etc.
- As we will see later, the web itself is an example of resource-oriented system
- REST (REpresentational State Transfer) is the most common model for resource-oriented web API design

REST philosophy & architecture

- REST's philosophy:
  - The existing principles and protocols for the web are powerful enough to create robust web APIs – we probably don't need SOAP/WSDL or other protocols.

- A very simple architecture:
  - Application states/data are modeled as resources.
  - Each resource is uniquely addressable using a URL.
    - E.g., http://bookstore.com/book/ISBN/.....
  - All resources share a uniform interface defined by the web protocol - HTTP.
  - HTTP supports a **constrained** set of well-defined operations – the "verbs": **GET**, **PUT**, **POST**, and **DELETE**

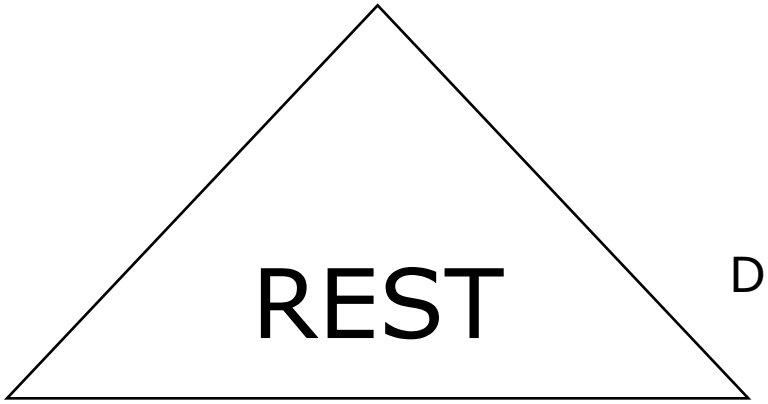
REST architecture

- The HTTP verbs are actually identical to the CRUD operations in traditional DBMS.
  - GET = READ
  - POST = CREATE
  - PUT = UPDATE
  - DELETE = DELETE
  - RESTful APIs support only these 4 operations for manipulating the resource identified by the URL.

- Based on the above definition, the web itself is a RESTful service
  - Every webpage/image/media file is a resource.
  - Each resource is accessible through a unique URL.
  - All resources in the web share the same interface defined by HTTP (GET/POST/PUT/DELETE)

The “REST Triangle”

Nouns
(unconstrained)
e.g. http://myservice.com/book/



Verbs
(constrained)
GET, POST,
PUT, DELETE

Data formats/content types
(constrained)
(X)HTML/XML/JSON,
Or other standard data format (e.g., text)

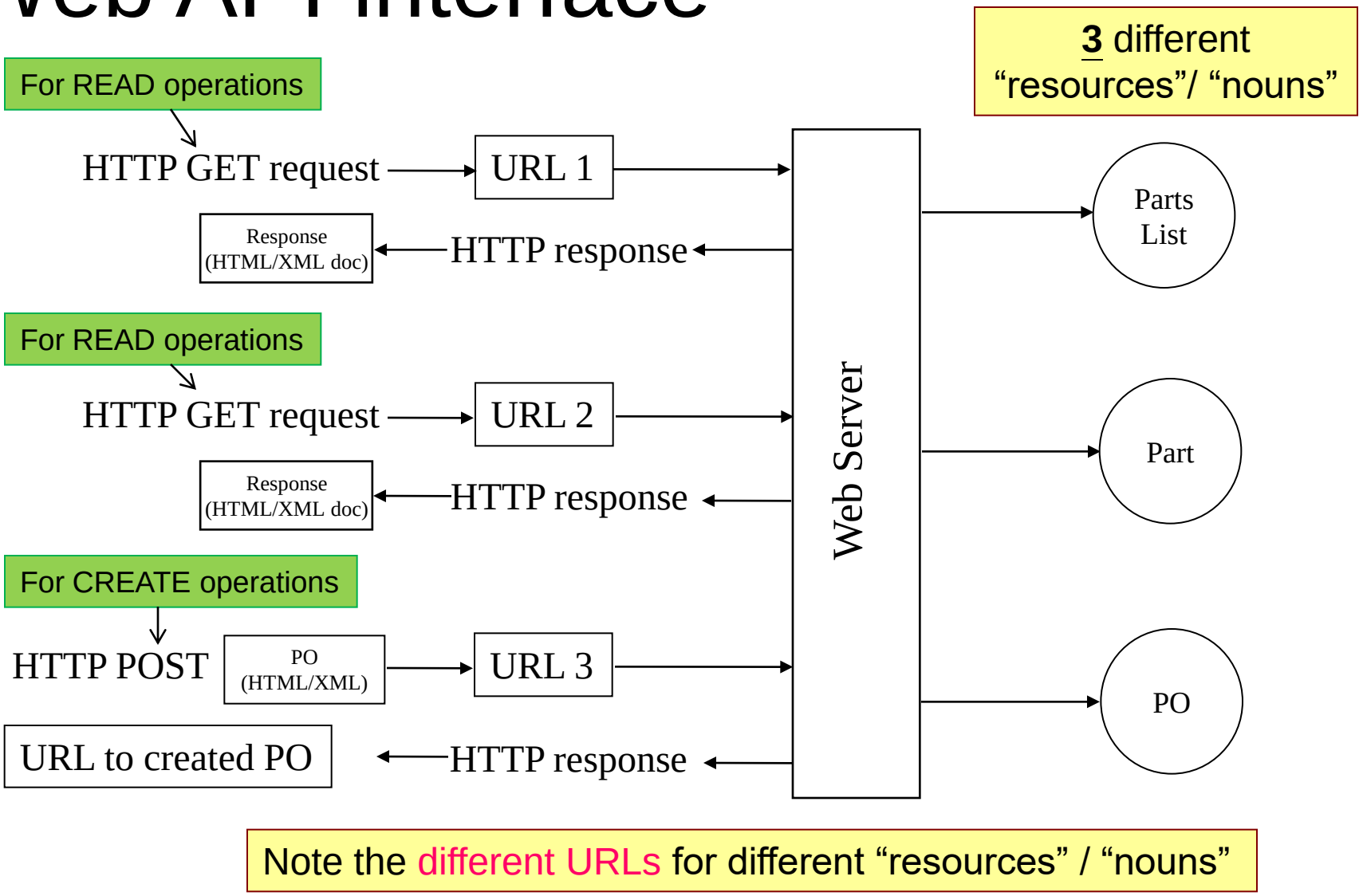
Example of RESTful service:
A parts depot service

A Parts Depot service

- Consider a Parts Depot Web API, which allows its clients to perform three operations:

1. Get a list of parts/products (through URL 1)
2. Get detailed information about a particular part (through URL 2)
3. Submit a purchase order (PO) (through URL 3)

Web API interface



Note the different URLs for different "resources" / "nouns"

GET operations

1. Get a list of parts

- The URL can be like this:

  http://www.parts-depot.com/parts

- If the API supports multiple data formats, it may offer URLs like these:

  http://www.parts-depot.com/parts?format=xml

  http://www.parts-depot.com/parts?format=json

  …

Response of GET

List of parts available

```
<?xml version="1.0"?>
<p:Parts xmlns:p="http://www.parts-depot.com"
         xmlns:xlink="http://www.w3.org/1999/xlink"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
                 "http://www.parts-depot.com
                  http://www.parts-depot.com/parts.xsd">
    <Part id="00345" xlink:href="http://www.parts-depot.com/parts/00345"/>
    <Part id="00346" xlink:href="http://www.parts-depot.com/parts/00346"/>
    <Part id="00347" xlink:href="http://www.parts-depot.com/parts/00347"/>
    <Part id="00348" xlink:href="http://www.parts-depot.com/parts/00348"/>
</p:Parts>
```

Note that the parts list has links to the detailed info about each part.
This is a key feature of REST: the clients can be transferred from one state to the next through the URLs contained in the response document.

That's why this API design is called "REpresentational State Transfer" (REST).

Getting more specific information

2. Get information about a particular part

- The RESTful API provides a URL to each part resource, such as:

  http://www.parts-depot.com/parts/00345?format=xml

Response

```
<?xml version="1.0"?>
<p:Part xmlns:p="http://www.parts-depot.com"
        xmlns:xlink="http://www.w3.org/1999/xlink"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation=
              "http://www.parts-depot.com
               http://www.parts-depot.com/part.xsd">
    <Part-ID>00345</Part-ID>
    <Name>Widget-A</Name>
    <Description>This part is used within the frap assembly</Description>
    <Specification xlink:href="http://www.parts-depot.com/parts/00345/specification"/>
    <UnitCost currency="USD">0.10</UnitCost>
    <Quantity>10</Quantity>
</p:Part>
```

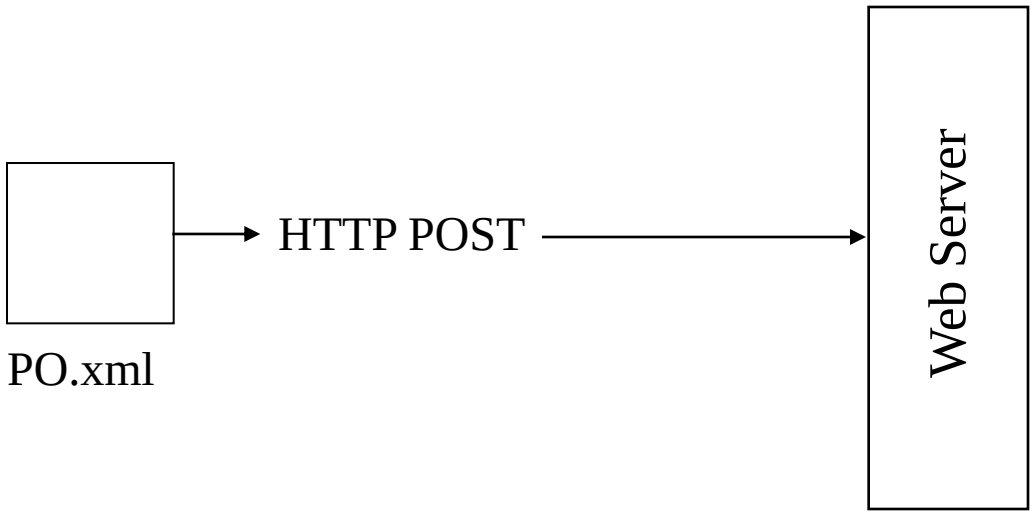
Link to further information
(e.g., the specification of a part)

Again, observe how the response allows the client to drill down to get further information.

POST operations

3. Submit a Purchase Order (PO)

- The web API provides a URL for creating new POs. The client submits a PO XML to that URL in a HTTP POST request.



Advantages of REST

- No need to formulate service interfaces or “contracts”
  - All communications are based on only 4 operations defined by HTTP (GET, POST, PUT, DELETE)
    - Just like the web
  - By contrast, every SOAP API has its own **custom interface** defined by WSDL
    - Flexible but inconvenient
    - Imagine if the web is designed in “the SOAP way”:
      - e.g., website A expects “GET index.html”; website B expects “GET_A_Page index.html”…
      - => much more inconvenient in retrieving web pages
      - => The web would be far less popular/scalable than it is

- => Provably **scalable** (consider the web)

  - Caching is simpler in REST than in SOAP
  - Because cache/proxy servers can cache data (e.g., obtained from GET) without the need to understand SOAP message format and WSDL

  - No need to use special application servers to host a RESTful API; current web servers can be used, which are highly-optimized
  - No vendor dependence

Modeling a RESTful service and some good practices

A stock broker example

- It is trivial to model a database-centric application as a RESTful service, e.g., the Amazon web APIs for books retrieval may look something like:
  - http://bookstore.com/ISBN/12345678
  - http://bookstore.com/book/harry_potter
- Less trivial to model web APIs that perform computations or operations
- Let's look at an example of modeling a simple web API provided by a stock broker (server) that allows traders' applications (clients) to perform stock trading.

Business operations

- Buy, sell, queueOrder, cancelOrder,
- getOrders, getClosedOrders,
- createQuote, getQuote, getAllQuotes,
- updateQuotePriceVolume,
- getHoldings, getHolding, getAccountData, getAccountProfileData, updateAccountProfile,
- Register

These operations can be easily put together to form a SOAP API, note the “verbs+nouns” nature, e.g., getOrders, just like getBook()

Less trivial to model them as a RESTful API

Modelling a RESTful API

1. Identify the nouns: account, profile, order, quote, holding, etc.

2. Apply the HTTP verbs: GET, POST, PUT, DELETE
   - Get = read; post = create; put = update; delete = delete

3. Design the URL space

The RESTful API

URL	HTTP Method (the verbs)	Nouns	CRUD	Business Operation
/acct/	POST	accounts	create	register
/acct/{acct_id}	GET	accounts	read	getAccountData
/acctprofile/{acct_id}	GET	profile	read	getAccountProfileData
/acctprofile/{acct_id}	PUT	profile	update	updateAccountProfile
/{acct_id}/open_orders/	POST	orders	create	buy/sell/queueOrder
/{acct_id}/open_orders/{order_id}	DELETE	orders	delete	cancelOrder
/{acct_id}/open_orders/	GET	orders	read	getOrders
/{acct_id}/closed_orders/	GET	orders	read	getClosedOrders
/{acct_id}/quotes/	POST	quotes	create	createQuote
/{acct_id}/quotes/{quote_id}	GET	quotes	read	getQuote
/{acct_id}/quotes/	GET	quotes	read	getAllQuotes
/{acct_id}/quotes/{quote_id}	PUT	quotes	update	updateQuotePriceVolume
/{acct_id}/holdings/	GET	holdings	read	getHoldings
/{acct_id}/holdings/{holding_id}	GET	holdings	read	getHolding

Authentication can be done using standard methods, e.g., sessions with SSL

Implementation

- A RESTful API, by definition, can be implemented from scratch by using the most basic technologies
- However, for rapid development, one should use the URL routing facilities supported in many web development frameworks (e.g., Laravel, Ruby on Rails, etc.) for implementing the URL mapping and routing
  - Laravel supports Resource Controllers (see Session 5) which handles GET/POST/PUT/DELETE
  - The scaffolding facilities of Rails support RESTful designs out of the box
- Standard data formats (e.g., XML, JSON, etc.) should always be used for better interoperability

Some good practices

- Provide a unique URL for each resource (e.g., account, profile, quote, holding, etc.) that you want to expose.

- Prefer URLs that are logical instead of URLs that are physical. For example:

  Prefer:

  http://www.boeing.com/airplanes/747

  Instead of:

  http://www.boeing.com/airplanes/747.html , or

  http://www.boeing.com/airplanes/747.xml , etc.

  The desired data type (e.g., HTML, XML, JSON, text, etc.) is better specified within the HTTP request header (in the ACCEPT field).

- Use **only** nouns in the logical URL, not verbs. Resources are "things" not "actions".
  - Always leave the “verbs” to the protocol (i.e., HTTP)

Some good practices

- Minimize the use of query strings. For example:

  Prefer:
  http://www.parts-depot.com/parts/00345
  Instead of:
  http://www.parts-depot.com/parts?part-id=00345

  - Use the slash "/" in a URL to represent a parent-child relationship.
  - Reason: the “parent-child” relationship between 'parts' and '00345' is clearer (like sub-folders) to both human and search engine robots, and you can further introduce sub-resources of '00345' easily.

- Use a "gradual unfolding methodology" for exposing data to clients. That is, a returned resource, whenever appropriate, should include links to further details.
  - Can reduce the number of subsequent requests

HTTP PUT and DELETE

- Not supported in browsers' user interface
  - But supported in JavaScript => can be used in AJAX.
  - So, RESTful APIs are well supported in AJAX

- Server-side programs need to check HTTP commands (GET/POST/PUT/DELETE) for proper handling of the requests
  - => routes in Laravel have to be defined separately for GET/POST/PUT/DELETE

Summary

- Web 2.0 encourages data reuse across websites, which needs standard protocols and standard data formats
- JSON – a popular alternative to XML for data sharing
- Web APIs can be "called" by remote programs (a server-side or a client-side script) for data sharing
- Five main architectures of web APIs: JavaScript, service-oriented, resource-oriented, RPC-based, and feed-based
- Service-oriented and resource-oriented web APIs
  - Service-orientation
    - A web API is modeled as a "service", which has operations that clients can invoke
    - SOAP is used as a messaging protocol; WSDL for formulating the contract of a service
    - Web service toolkits like Apache Axis/Axis2 simplify the development of SOAP APIs
  - Resource-orientation and RESTful web APIs
    - A web API is modeled as a "resource"
    - Existing web standards (e.g., HTTP) are used; no additional protocols needed
    - Unlike SOAP:
      - The operations allowed on a resource is constrained by the protocol (i.e., HTTP)
      - No need to formulate "contracts"

Post-class readings and references

Post-class readings:

- Take a look at the list of web APIs at Programmableweb.com
- Representational State Transfer (Wikipedia)

(Please see Moodle for links)

Reminder:
Group project

Reminder: group project

- Please form the project groups **by next Saturday (Feb 20, 2021)** and send your names to Steven.
- If you plan to work on your own, please also inform Steven or he will form groups for you later.
- If you have any questions, please feel free to post to Moodle or email Steven or me.

Optional consultation session

Optional, informal, consultation session – **please vote**…

- **It is optional**
  - For those who have questions on the labs, the assignment/project or any other course materials.
  - => Feel free to skip this session if you don't have any questions about the course.
- Date - two possibilities: **February 21 (Sun)**, 2pm-5pm, or **March 6 (Sat)**, 2pm-5pm
  - **=> Please vote (in Moodle) for the date you prefer by this Thursday (Feb 11)**
  - **The consultation session will be held on the date that is preferred by more students**
- Venue: Room P6-03, Graduate House (GH) **and** online (through a Zoom meeting). Steven and I will be there.
- Format: you may drop by in person, **or** join our online Zoom meeting
- Exact date and the Zoom link will be announced in due course.
- You are recommended to take a look at the labs and the assignment before the consultation session so that we can discuss if you have any questions about them.