



# ICOM 6034

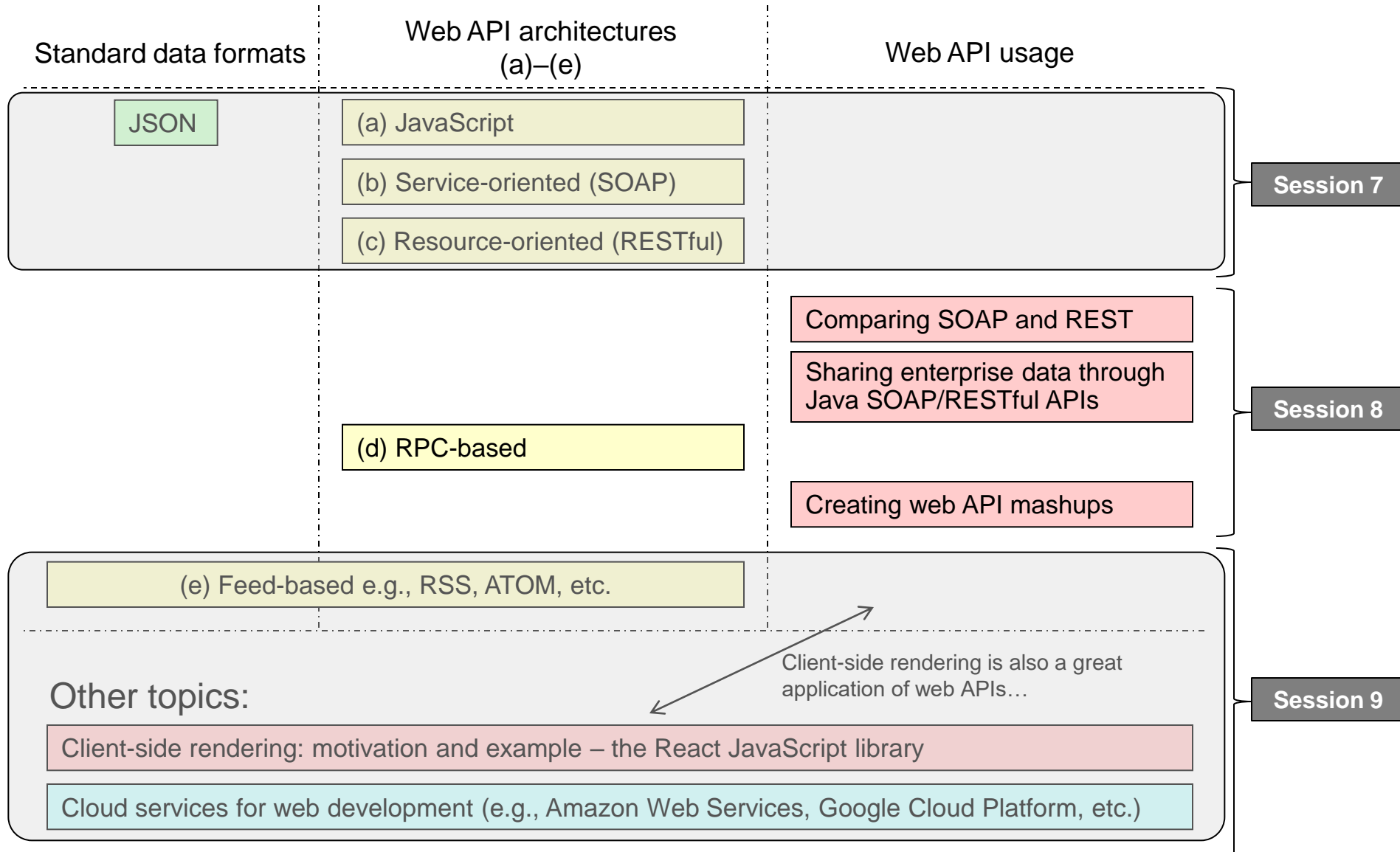
# Website Engineering

Dr. Roy Ho

Department of Computer Science, HKU

Session 8: Web API protocols (Part II) and service mashups

# Organization of Part 3 (Sessions 7-9)



# Session objectives

- Comparing SOAP and REST
- Sharing enterprise data through Java SOAP/RESTful APIs
- RPC-based web APIs
  - XML-RPC and JSON-RPC
- Mashups of web APIs
- Examples of client- and server-side mashups
  - Flickr and Google Maps
- Lab 4B: Using web APIs at the server side



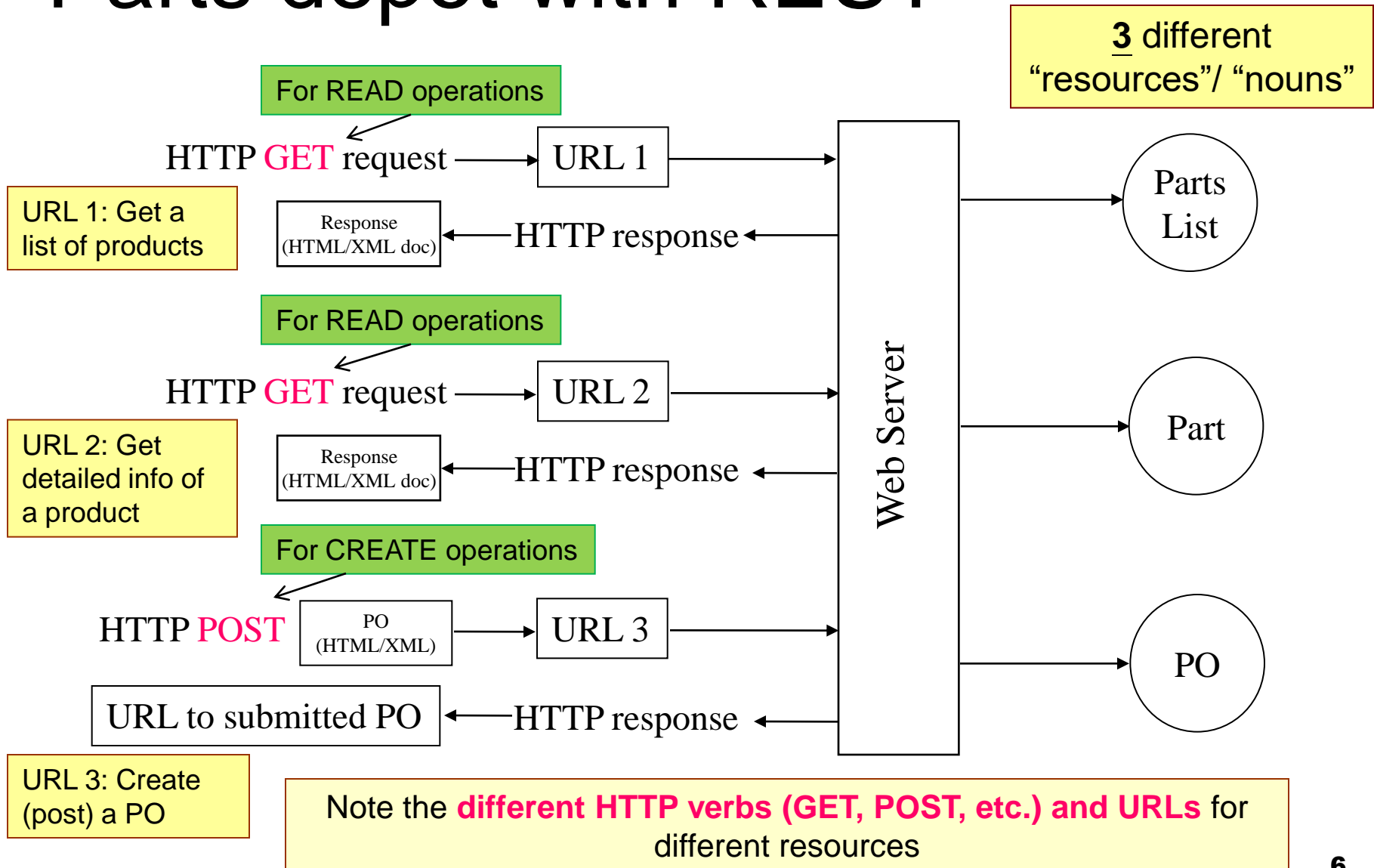
# Comparing SOAP and REST



# The Parts Depot service revisited

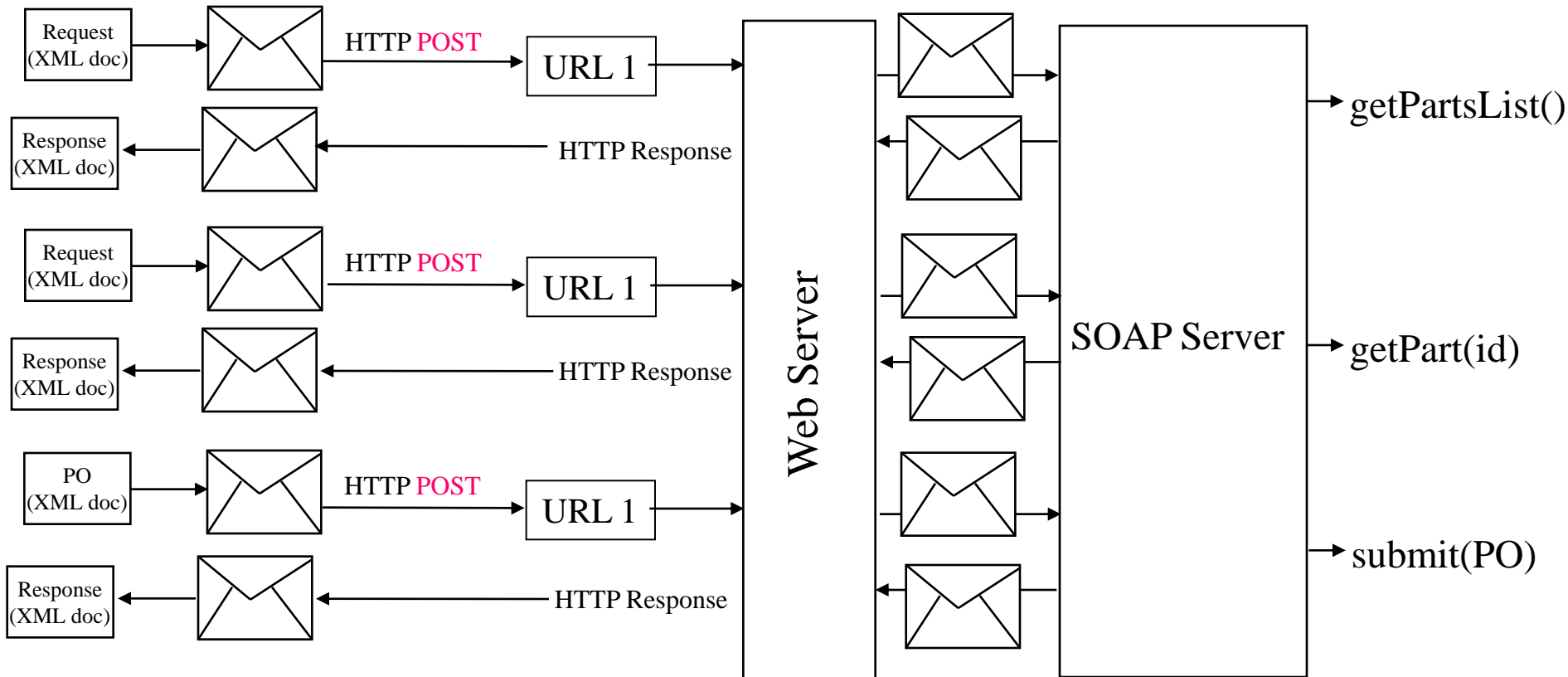
- Consider a Parts Depot Web API, which allows its clients to perform three operations:
  1. Get a list of parts/products
  2. Get detailed information about a particular part
  3. Submit a purchase order (PO)

# Parts depot with REST



# Parts depot with SOAP

## SOAP envelopes



Note the use of the **same** URL (URL 1) for all transactions. The SOAP server parses the SOAP request to determine which method to invoke. **All SOAP messages are sent using HTTP POST.**

# Similarities and differences

- Similarities:

- ☐ SOAP and REST are the same in their goals
  - Both allow API providers and clients to share data over a network.
- ☐ In both cases, you need prior agreement on data formats between client and server.

- Differences – two main aspects:

- ☐ Request handling and caching
- ☐ Scalability and interoperability



# Request handling and caching

- With REST, every resource has a unique URL.
  - SOAP requests for different operations are funneled through a single URL to the SOAP server.
- => A SOAP server must look inside a SOAP envelope (the payload) to find what information/operation is being requested. With REST, all decisions can be made based on the URL and HTTP command (i.e., GET/POST/PUT/DELETE).
  - REST is more efficient in request handling
- If an organization has a gateway service that provides custom filtering (e.g., for security) of certain requests, REST (based on URL) can be analyzed much more easily
  - But in SOAP, the envelope would need to be inspected, and the gateway would have to understand the interface of every SOAP application.
- With REST, GET is used for all “READ” operations => results can be cached at the client-side or in proxies
  - By contrast, when submitting a SOAP message, HTTP POST is always used (no matter a request is for “reading” or not); the response is not cached by default
  - So, caching SOAP responses with existing cache servers is not quite possible
  - No/less caching = higher server loadings + longer response time

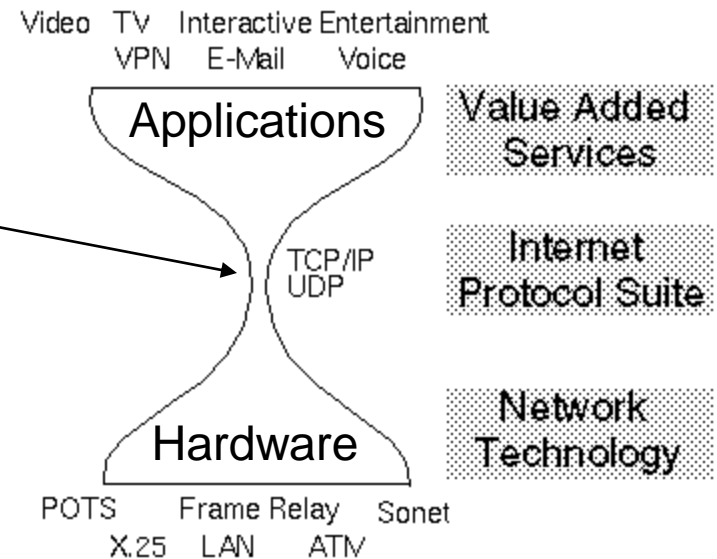
# Scalability and interoperability

- REST is based on well-established and widely-implemented standards: URLs, HTTP GET/POST/PUT/DELETE, etc.
  - Client and server software supporting these standards are everywhere
  - REST in general has better interoperability.
- **REST has a limited set of “verbs” (CRUD); SOAP has no defined set of methods.**
  - Consequently, tools and clients must be customized on a per-application basis with SOAP. **This is not scalable.**
  - (See the next slide on the network protocol stack analogy)

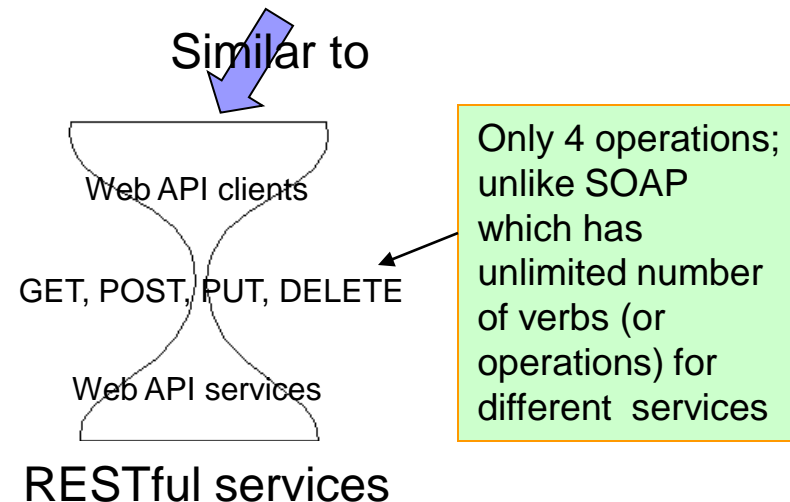
# Why limiting the verbs?

## Network protocol stack as an analogy

- **Simple** protocols (TCP/IP) with limited operations (verbs) such as connect(), route(), etc. that “**glue**” applications and different network hardware together.
- “**Simple**” => more likely that both the applications side and the hardware side have **complete support** and **stable implementation** of the protocol standards
  - Interoperability and compatibility
  - Lower implementation cost => hence more “gluing agents” (e.g., routers) can be deployed => scalability
  - Less bugs (they are critical components of the system)
- Imagine if TCP/IP is implemented differently on different OS and routers, the Internet would never be so scalable



**Figure 1: Hourglass-model of the Internet**



# Discussions

- REST is simple, cost-effective (can use existing web servers for hosting web APIs) and highly scalable.
- Amazon, which supported both SOAP and REST, reported that their usage was 80% REST, and 20% SOAP.
- REST is often faster than SOAP (Amazon claimed that REST is up to six times faster)
  - (I believe this is not a problem of the protocol itself but how much the software is optimized - HTTP clients/servers have been highly optimized, while SOAP-related stuff are far less popular => less optimized.)
- REST mimic traditional DBMS operations (CRUD) => easier to understand and model.

# Discussions

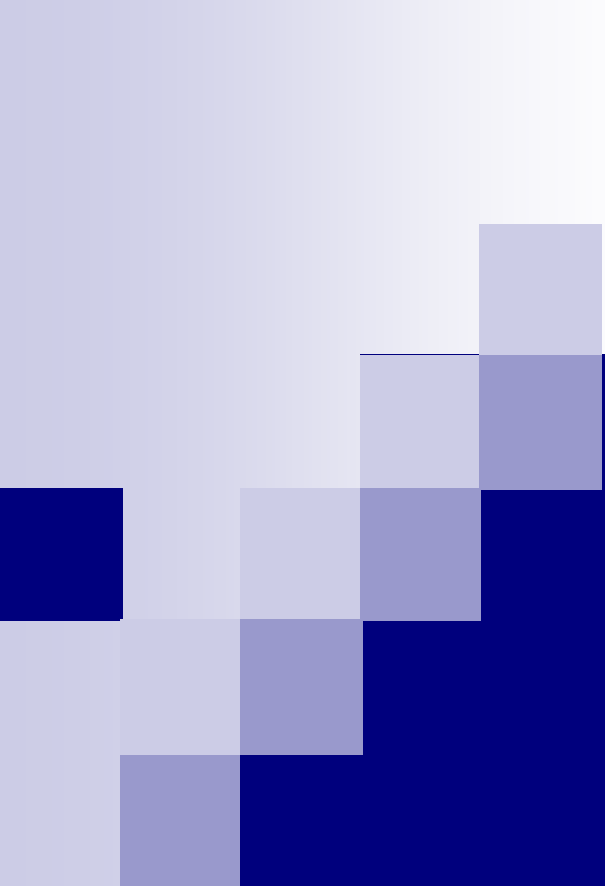
- However, SOAP has greater flexibility in modeling how a service looks like and behaves (the interaction pattern)
  - By contrast, REST has to model everything as a “resource” (i.e., a noun)
  - REST is bounded by the (HTTP’s) request/response pattern and the CRUD operations
- If you have an existing set of business operations running in the backend (e.g., the Stock Broker service we discussed in Session 7), it might be easier to implement a SOAP service as a “direct mapping” of those internal operations
  - By contrast, some re-modeling effort would be needed for turning these internal operations into a RESTful API.
  - => a tradeoff between interoperability and implementation effort.
- SOAP can have richer functionality with the support from vendors (which support other (non-web-based) backend enterprise applications as well).
  - => SOAP might be suitable for applications that integrate tightly with backend systems and require advanced capabilities
- Some developers believe that SOAP is best used in closed systems within an enterprise because of the better vendor support



# Sharing enterprise data with Java SOAP/ RESTful APIs

# Java SOAP/RESTful services

- Enterprise data often needs to be exposed to outside parties:
  - The web frontend of the company
  - Business partners
  - Other websites that want to consume/display the data (e.g., many websites help promote Amazon's products through its affiliate program/APIs)
- => Web APIs are needed to allow these parties to obtain the data
- Although many web frameworks support implementation of web APIs (e.g., it's easy to create SOAP/RESTful APIs with Laravel/RoR), many backend enterprise systems or data stores are implemented in Java Enterprise Edition (Java EE)
  - In these cases, it might be easier to implement the web APIs in Java as well
- Java Enterprise (or Java EE) supports SOAP/RESTful web APIs:
  - SOAP: JAX-WS and the Metro protocol stack implementation
  - RESTful: JAX-RS and the Jersey reference implementation



# Implementing SOAP APIs in Java (Metro / JAX-WS)

(Note: program code in this section will not be included in the examination)



# JAX-WS, JAXB and Metro

- Core Java standards for SOAP APIs:
  - **JAXB**: a set of APIs and annotations for converting XML documents to/from Java objects.
  - **JAX-WS**: a set of APIs and annotations that allow building and consuming SOAP APIs with Java.
- **Metro** is an open source library that supports JAXB and JAX-WS.

# Java Architecture for XML Binding (JAXB)

**JAXB** defines a standard to convert Java objects to XML and vice versa. It also manages **XML Schema Definitions** (XSD) transparently.

All data exchanged between SOAP API and clients is in XML

## @XmlRootElement

```
public class CreditCard {
    private String number;
    private String expiryDate;
    private Integer controlNumber;
    private String type;
    // Constructors, getters, setters
}
```

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

```
<creditCard>
```

```
    <controlNumber>6398</controlNumber>
```

```
    <expiryDate>12/19</expiryDate>
```

```
    <number>1234</number>
```

```
    <type>Visa</type>
```

```
</creditCard>
```

Conversions done by JAXB

XML validations using schema

JAXB also generates the XML schema for validating any objects converted as XML

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

```
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

```
    <xs:element name="creditCard" type="creditCard"/>
```

```
    <xs:complexType name="creditCard">
```

```
        <xs:sequence>
```

```
            <xs:element name="controlNumber" type="xs:int" minOccurs="0"/>
```

```
            <xs:element name="expiryDate" type="xs:string" minOccurs="0"/>
```

```
            <xs:element name="number" type="xs:string" minOccurs="0"/>
```

```
            <xs:element name="type" type="xs:string" minOccurs="0"/>
```

```
        </xs:sequence>
```

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

```
</xs:schema>
```

The corresponding XML schema of CreditCard

# The JAX-WS model

- To turn a class/interface into a SOAP service, simply **annotate** it with `@javax.jws.WebService`.

**@WebService**

```
public interface ItemWeb {  
    List<Book> findBooks();  
    List<CD> findCDs();  
    Book createBook(Book book);  
    CD createCD(CD cd);  
}
```

- A stateless bean can also be a SOAP service at the same time.

**@Stateless**

```
public class ItemEJB implements ItemWeb {  
    ...  
}
```

# A simple example

## @WebService

```
public class CardValidator {
    public boolean validate(CreditCard creditCard) {
        String lastDigit = creditCard.getNumber().substring(
            creditCard.getNumber().length() - 1,
            creditCard.getNumber().length());
        if (Integer.parseInt(lastDigit) % 2 != 0) {
            return true;
        } else {
            return false;
        }
    }
}
```

This is a SOAP service

## @XmlElement

```
public class CreditCard {
    private String number;
    private String expiryDate;
    private Integer controlNumber;
    private String type;
    // Constructors, getters, setters
}
```

This object will be converted into XML (and vice versa) by JAXB upon service invocation. Handling of XML by developers is not required.

// The web service consumer

```
public class Main {
    public static void main(String[] args) {
        CreditCard creditCard = new CreditCard();
        creditCard.setNumber("12341234");
        creditCard.setExpiryDate("10/20");
        creditCard.setType("VISA");
        creditCard.setControlNumber(1234);
        CardValidator cardValidator =
            new CardValidatorService().getCardValidatorPort();
        cardValidator.validate(creditCard);
    }
}
```

This is the client

Given this API implementation, Metro is able to generate the **WSDL** automatically.

Based on the published **WSDL**, clients can use Metro's tools to generate a **CardValidatorService** interface to obtain a CardValidator (similar to Apache Axis) --- see the following slides.

# The generated WSDL file

```
<definitions targetNamespace="http://javaee6.org/" ➡
  name="CardValidatorService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://javaee6.org/" schemaLocation ➡
        ="http://localhost:8080/CardValidatorService?xsd=1"/>
      </xsd:schema>
    </types>
    <message name="validate">
      <part name="parameters" element="tns:validate"/>
    </message>
    <message name="validateResponse">
      <part name="parameters" element="tns:validateResponse"/>
    </message>
    <portType name="CardValidator">
      <operation name="validate">
        <input message="tns:validate"/>
        <output message="tns:validateResponse"/>
      </operation>
    </portType>
```

URL of the XML schema for CreditCard  
(see next slide...)

<portType> makes  
references to <message>;

<message> to <types>;

<binding> to <portType> and  
<operation>;

<service> to <binding>

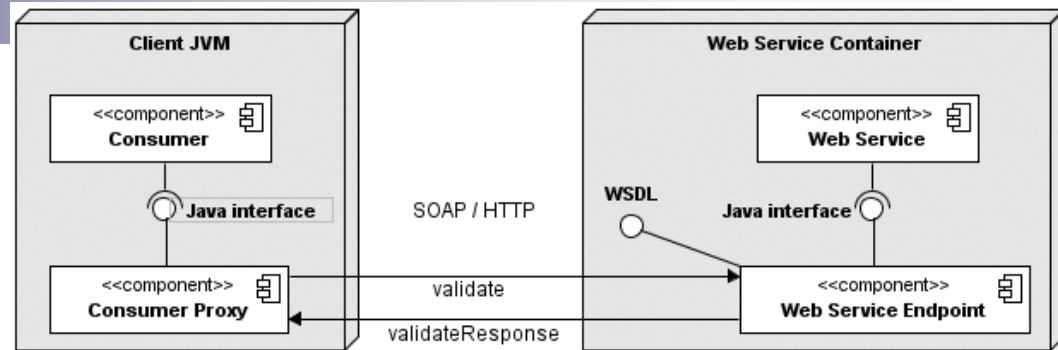
```
<binding name="CardValidatorPortBinding" type="tns:CardValidator">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" ➡
    style="document"/>
  <operation name="validate">
    <soap:operation soapAction=""/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
<service name="CardValidatorService">
  <port name="CardValidatorPort" binding="tns:CardValidatorPortBinding">
    <soap:address location = ➡
      "http://localhost:8080/CardValidatorService"/>
  </port>
</service>
</definitions>
```

Public location of this API (the endpoint)

# The generated XML schema

```
<xs:schema version="1.0" targetNamespace="http://javaee6.org/">
  <xs:element name="creditCard" type="tns:creditCard"/>
  <xs:element name="validate" type="tns:validate"/>
  <xs:element name="validateResponse" type="tns:validateResponse"/>
  <xs:complexType name="validate">
    <xs:sequence>
      <xs:element name="arg0" type="tns:creditCard" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="creditCard">
    <xs:sequence>
      <xs:element name="controlNumber" type="xs:int" minOccurs="0"/>
      <xs:element name="expiryDate" type="xs:string" minOccurs="0"/>
      <xs:element name="number" type="xs:string" minOccurs="0"/>
      <xs:element name="type" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="validateResponse">
    <xs:sequence>
      <xs:element name="return" type="xs:boolean"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

# Metro's client-side supports



- Metro provides a WSDL-to-Java utility tool (**wsimport**) that generates Java interfaces from a WSDL.
- Such interfaces are called **service endpoint interfaces (SEI)**, which acts like a proxy that routes the local Java call to the remote SOAP API using HTTP.
- The proxy is similar to the “client stubs” in Apache Axis which we introduced in Session 7.

```
CardValidatorService cardValidatorService = new CardValidatorService();
CardValidator cardValidator = cardValidatorService.getCardValidatorPort();
cardValidator.validate(creditCard);
```

Client code

The CardValidatorService, CardValidator, and CreditCard classes do not have to be implemented at the client side.

They are generated from the WSDL by the wsimport tool.

# The SOAP messages

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"  
  xmlns:cc="http://javaee6.org/">  
  <soap:Header/>  
  <soap:Body>  
    <cc:validate>  
      <arg0>  
        <controlNumber>1234</controlNumber>  
        <expiryDate>10/20</expiryDate>  
        <number>9999</number>  
        <type>VISA</type>  
      </arg0>  
    </cc:validate>  
  </soap:Body>  
</soap:Envelope>
```

The request

Both could be validated  
against the XML  
schema generated by  
JAXB

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">  
  <soap:Body>  
    <ns2:validateResponse xmlns:ns2="http://javaee6.org/">  
      <return>true</return>  
    </ns2:validateResponse>  
  </soap:Body>  
</soap:Envelope>
```

The response



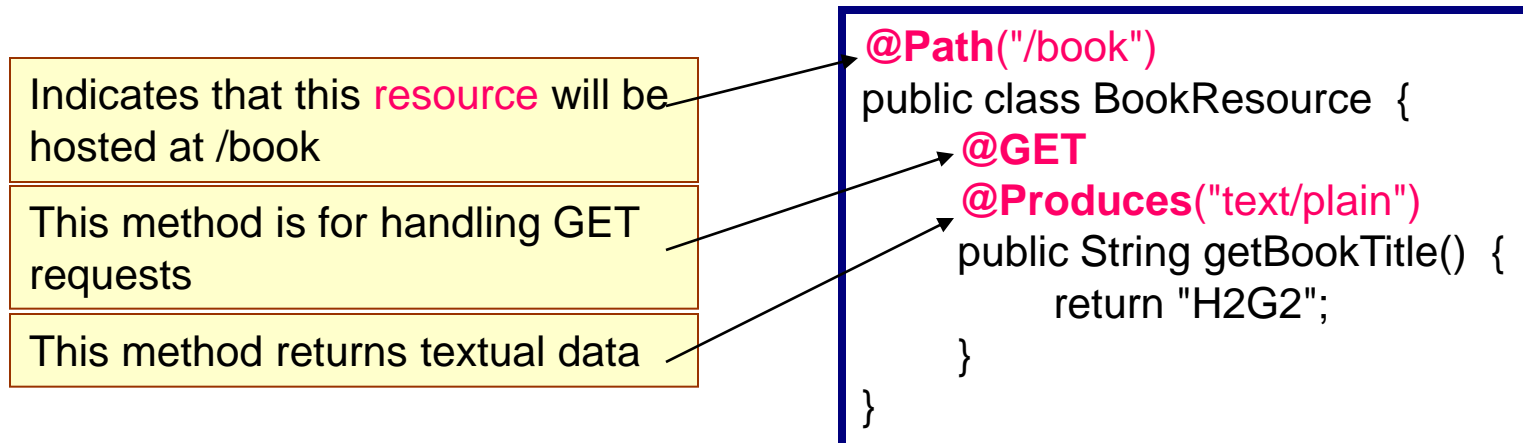


# Implementing RESTful APIs in Java (Jersey / JAX-RS)

(Note: program code in this section will not be included in the examination)

# Java API for RESTful web APIs

- **JAX-RS** defines a set of APIs that simplify implementation of RESTful APIs. **Jersey** is a reference implementation of JAX-RS.
- JAX-RS also relies on **JAXB** to convert XML and **JSON** data into objects, and vice versa.
- With JAX-RS:
  - **Resources** are Java objects that have methods annotated with `@javax.ws.rs.Path` (=> the URL of a resource)
  - Developers do not need to parse HTTP requests or create HTTP responses manually



# URL space

The **root path** where the resource is located. This is the root of the “tree” of all sub-resources

Can also embed parameters in the URL:  
e.g., `@Path("{itemid}")`

`@Path` may also be defined for individual methods.

If it exists in both class and method, the final path is a **concatenation** of the two.

E.g., this path would become:

`/items/books/1234`

```
@Path("/items")
public class ItemsResource {
    @GET
    public List<Item> getListOfItems() {
        // ...
    }
    @GET
    @Path("{itemid}")
    public Item getItem(@PathParam("itemid") String itemid) {
        // ...
    }
    @PUT
    @Path("{itemid}")
    public void putItem(@PathParam("itemid") String itemid,
        Item item) {
        // ...
    }
    @DELETE
    @Path("{itemid}")
    public void deleteItem(@PathParam("itemid") String itemid) {
        // ...
    }
    @GET
    @Path("/books/")
    public List<Book> getListOfBooks() {
        // ...
    }
    @GET
    @Path("/books/{bookid}")
    public Book getBook(@PathParam("bookid") String bookid) {
        // ...
    }
}
```

# Extracting parameters

```
@Path("/customers")
public class CustomersResource {
    @GET
    @Path("{customerid}")
    public Customer getCustomer(@PathParam("customerid") int customerid) {
        // ...
    }
}
```

http://www.abc.com/customers/12345

```
@Path("/customers")
public class CustomersResource {
    @GET
    public Customer getCustomerByZipCode(@QueryParam("zip") Long zip) {
        // ...
    }
}
```

http://www.myserver.com/customers?zip=19870

Extracting variables stored in cookies.

```
@Path("/products")
public class ItemsResource {
    @GET
    public Book getBook(@CookieParam("sessionid") int sessionid) {
        // ...
    }
}
```

```
@Path("/customers")
public class CustomersResource {
    @GET
    public Response getCustomers(@DefaultValue("50") @QueryParam("age")
        int age) {
        // ...
    }
}
```

Default value to be used when the variable is unspecified.

# Content types

With REST, a resource can have multiple representations; a “book” can be returned as a webpage (html), XML/JSON data, or an image showing the book cover.

If a resource is capable of producing more than one data type, the resource method chosen will depend on the most acceptable media type declared by the client in the “Accept” header in the HTTP request, e.g.,:

Accept: application/json

```
@Path("/customers")
@Produces("text/plain")
public class CustomersResource {
    @GET
    public String getAsPlainText() {
        // ...
    }
    @GET
    @Produces("text/html")
    public String getAsHtml() {
        // ...
    }
    @GET
    @Produces("application/json")
    public List<Customer> getAsJson() {
        // ...
    }
    @PUT
    @Consumes("text/plain")
    public void putBasic(String customer) {
        // ...
    }
    @POST
    @Consumes("application/xml")
    public Response createCustomer(InputStream is) {
        // ...
    }
}
```

# Example: an online bookstore

This is the “Book” entity class representing the database through ORM.

JAXB will do the object ⇔ XML/JSON conversion automatically

```
@Entity
@XmlRootElement
@NamedQuery(name = "findAllBooks", query = "SELECT b FROM Book b")
public class Book {
    @Id @GeneratedValue
    private Long id;
    @Column(nullable = false)
    private String title;
    private Float price;
    @Column(length = 2000)
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;
    // Constructors, getters, setters
}
```

# BookResource: the header

Indicates the path of this WS:  
http://example.com/books

This is also a stateless bean

This APIs produces and  
consumes XML or JSON data

```
@Path("/books")
@Produces({"application/xml", "application/json"})
@Consumes({"application/xml", "application/json"})
@Stateless
public class BookResource {
    @PersistenceContext(unitName = "book_example")
    private EntityManager em;
    @Context
    private UriInfo uriInfo;
    .....
}
```

# Getting a book

```
@GET
@Path("/{id}")
public Book getBookById(@PathParam("id") Long id) {
    Book book = em.find(Book.class, id);
    return book;
}
```

```
curl -X GET -H "Accept: application/xml" ➡
http://localhost:8080/books/601
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<book><description>Scifi IT book</description>
<illustrations>>false</illustrations><isbn>1234-234</isbn>
<nbOfPage>241</nbOfPage><price>24.0</price><title>H2G2</title></book>
```

```
curl -X GET -H "Accept: application/json" ➡
http://localhost:8080/books/601
```

```
{"description":"Scifi IT book","illustrations":"false",➡
"isbn":"1234-234","nbOfPage":"241","price":"24.0","title":"H2G2"}
```



# Creating a book

**@POST**

```
public Response createNewBook(JAXBElement<Book> bookJaxb) {  
    Book book = bookJaxb.getValue();  
    em.persist(book);  
    URI bookUri = uriInfo.getAbsolutePathBuilder().  
        path(book.getId().toString()).build();  
    return Response.created(bookUri).build();  
}
```

JAXB will do the  
"XML/JSON =>  
object" conversion  
automatically

curl -X **POST** --data-binary ➡

```
"{ \"title\": \"H2G2\", \"description\": \"Scifi IT book\",  
  \"illustrations\": \"false\", \"isbn\": \"134-234\", \"nbOfPage\": \"241\",  
  \"price\": \"24.0\" }" -H "Content-Type: application/json" ➡  
http://localhost:8080/books -v
```

The command to test

HTTP request

```
> POST /books HTTP/1.1  
> User-Agent: curl/7.19.0 (i586-pc-mingw32msvc) libcurl/7.19.0 zlib/1.2.3  
> Host: localhost:8080  
> Accept: */*  
> Content-Type: application/json  
> Content-Length: 127  
>  
< HTTP/1.1 201 Created  
< X-Powered-By: Servlet/3.0  
< Server: GlassFish/v3  
< Location: http://localhost:8080/books/601  
< Content-Type: application/xml  
< Content-Length: 0
```

Status code 201  
indicates that the  
resource has been  
created.

The URL of the  
newly-created  
book.

HTTP response

# Deleting a book

**@DELETE**

**@Path("{id}")**

```
public void deleteBook(@PathParam("id") Long id) {  
    Book book = em.find(Book.class, id);  
    em.remove(book);  
}
```

```
curl -X DELETE http://localhost:8080/books/601 -v
```

```
> DELETE /books/601 HTTP/1.1
```

```
> User-Agent: curl/7.19.0 (i586-pc-mingw32msvc) libcurl/7.19.0 zlib/1.2.3
```

```
> Host: localhost:8080
```

```
> Accept: */*
```

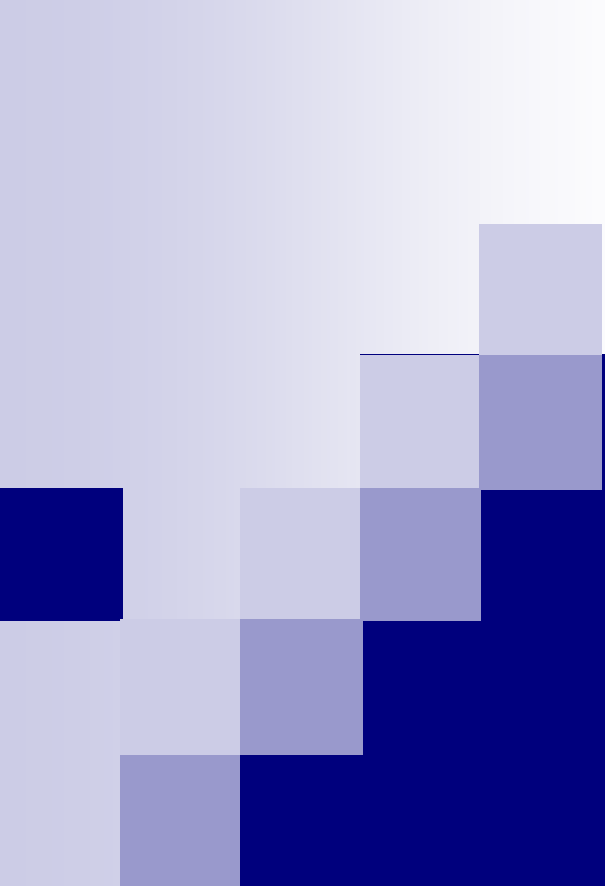
```
>
```

```
< HTTP/1.1 204 No Content
```

```
< X-Powered-By: Servlet/3.0
```

```
< Server: GlassFish/v3
```

Status code 204 indicates that this resource does not exist anymore.

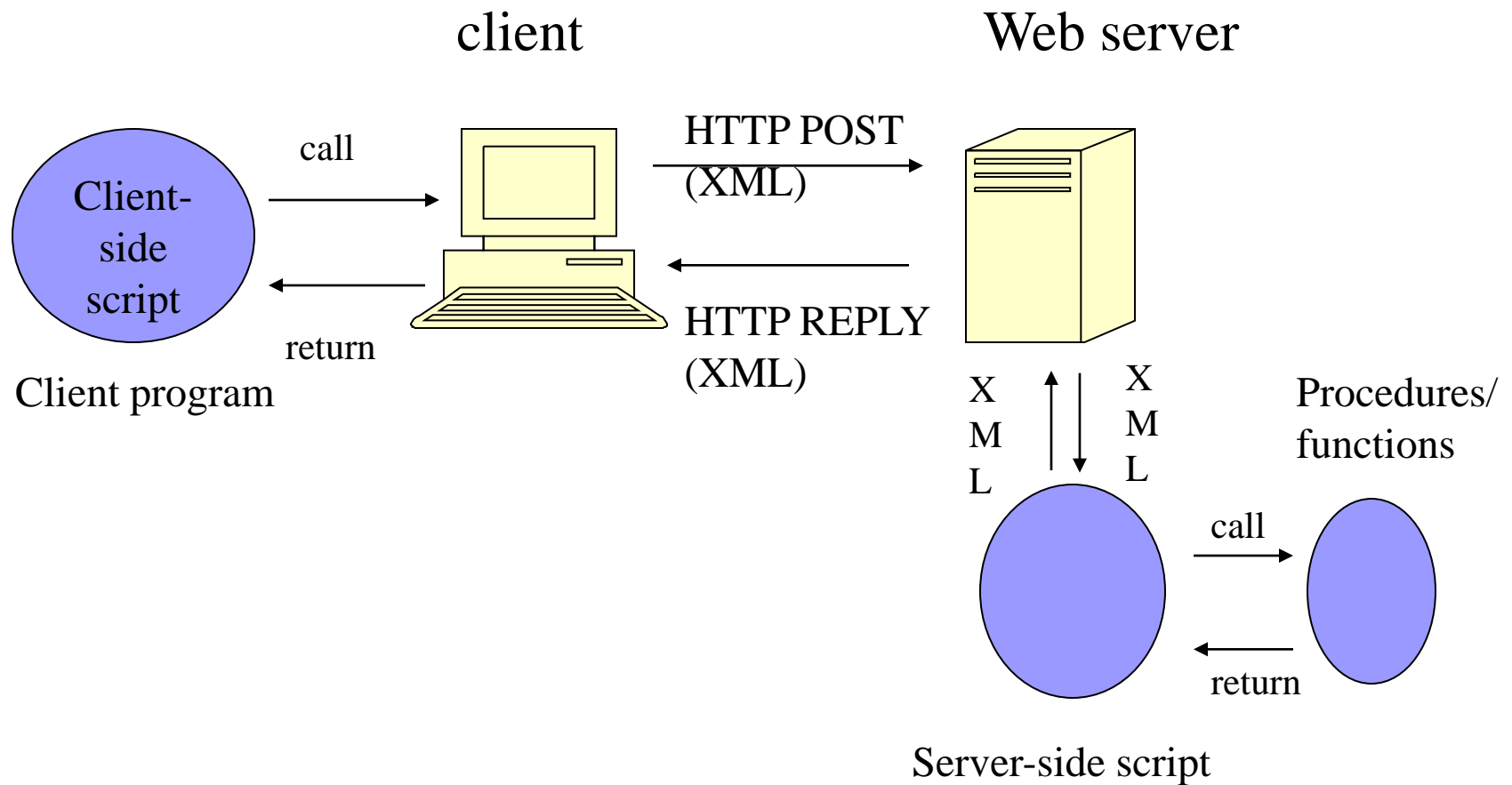


## Web API protocol (d): RPC-based protocols (XML-RPC and JSON-RPC)

# XML-RPC

- Remote procedure calling (**RPC**) with **XML** through **HTTP**
  - RPC is conceptually similar to RMI in Java
  - A mechanism which allows a program running on one computer to execute a function on another computer
- Both client-side (JavaScript) and server-side libraries for handling XML-RPC requests and replies are available in popular web languages/frameworks (e.g., PHP, Rails, etc.).
- A lightweight form of SOAP:
  - XML-RPC also adopts the **SOA** (service-oriented architecture) – the web API is modeled as a “service” which provides a number of “operations” that remote users can invoke
  - XML-RPC, however, doesn’t require formulation of contracts
- A typical XML-RPC call:
  - A client sends a RPC request through HTTP POST
  - The server handles the request and sends the return value to the client
    - Server’s address is a standard URL
    - <http://example.org:8080/rpcserver/>
- A request should indicate a **method name** and its **parameters**
  - Response should contain a return value and indicate if it is a successful return or an error message


# XML-RPC flow



# Request example

```
POST /RPC2 HTTP/1.1
User-Agent: Frontier/5.1.2
Host: example.com
Content-Type: text/xml
Content-length: 181

<?xml version="1.0"?>
<methodCall>
  <methodName>examples.getStateName</methodName>
  <params> <param>
    <value><i4>41</i4></value>
  </param></params>
</methodCall>
```



HTTP header

- A single <methodCall> structure to indicate this is an API request
  - A single <methodName>, which contains the calling method's name
  - A single <params>, which can contain any number of
    - <param>s. Each <param> has a <value>



# Data types

<i4> or <int>	four-byte integer	-12
<boolean>	0 (false) or 1 (true)	1
<string>	ASCII string	Hello
<double>	double-precision	3.1415
<dateTime.iso8601>	date/time	20190717T14:08:55
<base64>	base64-encoded binary	eW91IGNhbid

(default type is string)

# Data types (2) - <struct>

<struct> contains <member>s, each <member> contains a <name> and a <value>

```
<struct>
```

```
  <member>
```

```
    <name>lowerBound</name> <value><i4>18</i4></value>
```

```
  </member>
```

```
  <member>
```

```
    <name>upperBound</name> <value><i4>122</i4></value>
```

```
  </member>
```

```
</struct>
```



# Data types (3) - <array>

An <array> contains a single <data> element, which can contain any number of <value>s

```
<array>
```

```
  <data>
```

```
    <value><i4>12</i4></value>
```

```
    <value><string>Egypt</string></value>
```

```
    <value><boolean>0</boolean></value>
```

```
    <value><i4>-31</i4></value>
```

```
  </data>
```

```
</array>
```

# Response example

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 158
Content-Type: text/xml
Date: Wed, 26 Sep 2015 10:10:28 GMT
Server: UserLand Frontier/5.1.2-WinNT
```

HTTP header

```
<?xml version="1.0"?>
<methodResponse>
  <params><param>
    <value><string>South Dakota</string></value>
  </param></params>
</methodResponse>
```

- One `<methodResponse>`, it may contain either
  - A single `<params>` -- a successful procedure return, which contains a single `<param>` with a single `<value>` (but the `<value>` can be a struct or an array)
  - A `<fault>` -- a failure procedure return

# JSON-RPC

- Very similar to XML-RPC, but replaces XML with JSON
- There are JavaScript clients for XML-RPC and JSON-RPC, so browsers can consume such APIs directly.
- Libraries for XML-RPC/JSON-RPC are also available for server-side (e.g., there is a library for implementing JSON-RPC APIs in Laravel)
- Many web APIs nowadays are using either **plain JSON** (sent through JavaScript) or **JSON-RPC**
- If your web application or its backend is not XML-based, it should be easier to use JSON.

# Five main architectures of web APIs

- **(e) JavaScript**: no fixed protocol; the data provider simply provides a JavaScript library that enables clients (mostly browsers) to connect to the website through POST operations in plain JavaScript or AJAX. Can use XML, JSON or plain text for data sharing
- **(b) Service-oriented architecture**: model a web API as a “service” – a service is a collection of operations/functions that you can invoke, and return the output (of the service) to the clients.
  - Main model: SOAP-based services
- **(c) Resource-oriented**: model a web API as a “resource” – resources are some “nouns” that reflect real-life objects (books, calendar, photos, etc.) with a limited set of operations (create, read, update, delete).
  - Main model: RESTful services

Introduced in  
last lecture  
(Session 7)

- **(d) RPC-based**: a simpler form of the service-oriented architecture - model a web API as “Remote Procedure Call”. APIs and clients exchange call parameters and return values, through standard data formats such as XML, JSON, etc.

Introduced in  
this lecture

- **(e) Feed-based**: the website provides (or accepts) a feed (e.g., RSS/ATOM) for data sharing

Next lecture  
(Session 9)



# Mashup of web APIs

# Mashups

- Our discussions so far are mainly about the design and implementations of web APIs (i.e., about **providing** web API services)
  - Mashup is about **making use of, or consuming, web APIs** in websites
- **Mashup** is a web page (or a web page component) that consumes data or functionality from one or more web APIs to create a new service, or add values to existing data
- The data sources for mashups can be **any web API outputs or web feeds**
  - The number of possibilities is virtually unlimited
- Main motivations:
  - Combination
  - Visualization
  - Aggregation

# Combination

- Join across dimensions

- E.g., Subject + Time/Place + ...

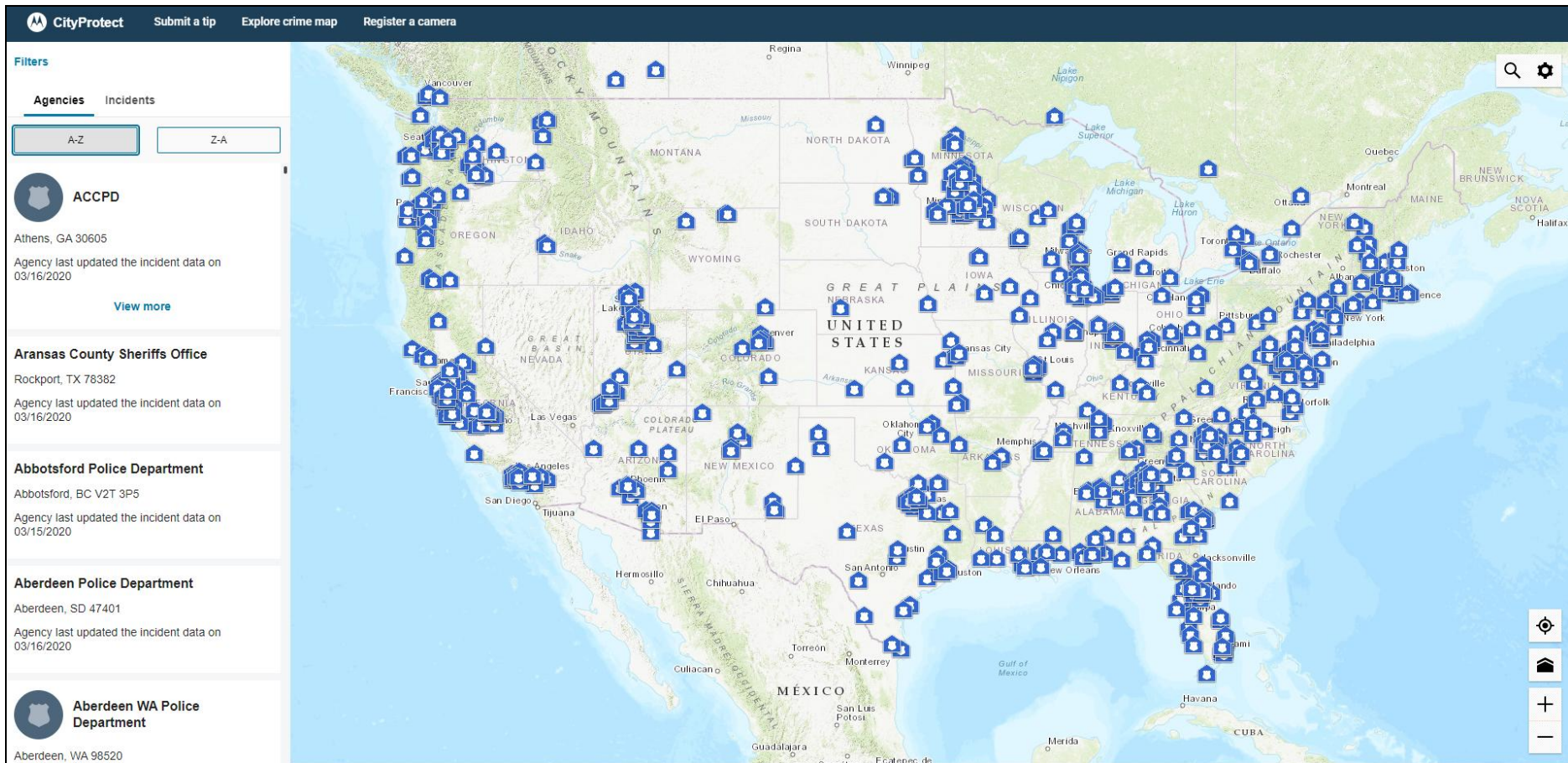
Online bookmark service

Related music clips or albums

IP address checking service

- Bands I like + Where I live + Spotify's API  
= your new mashup (e.g., a personalized music portal)

# Visualization



CityProtect.com

(Updated crime records from multiple agencies + online maps)

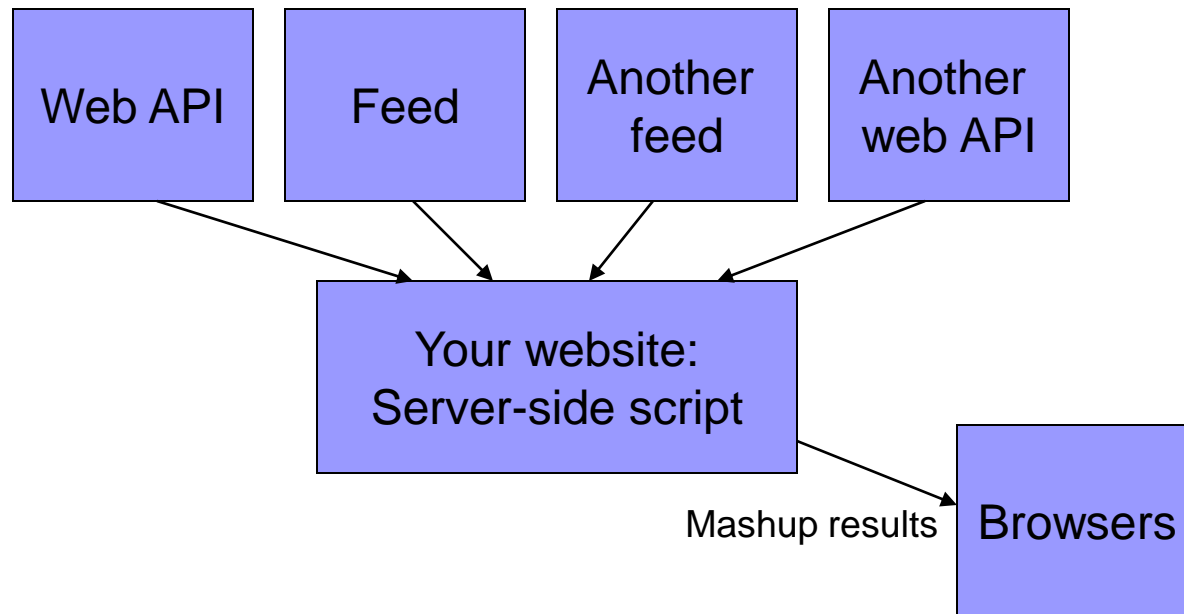


# Aggregation

- Group the data and take a measure or statistical analysis
  - Sum, Avg, Min, Max
  - Classification, prediction, clustering, etc.
- => The goal is to create new information or knowledge from the data
- E.g., combine, analyze, and/or display Google and Flickr image search results

# Server-side mashups

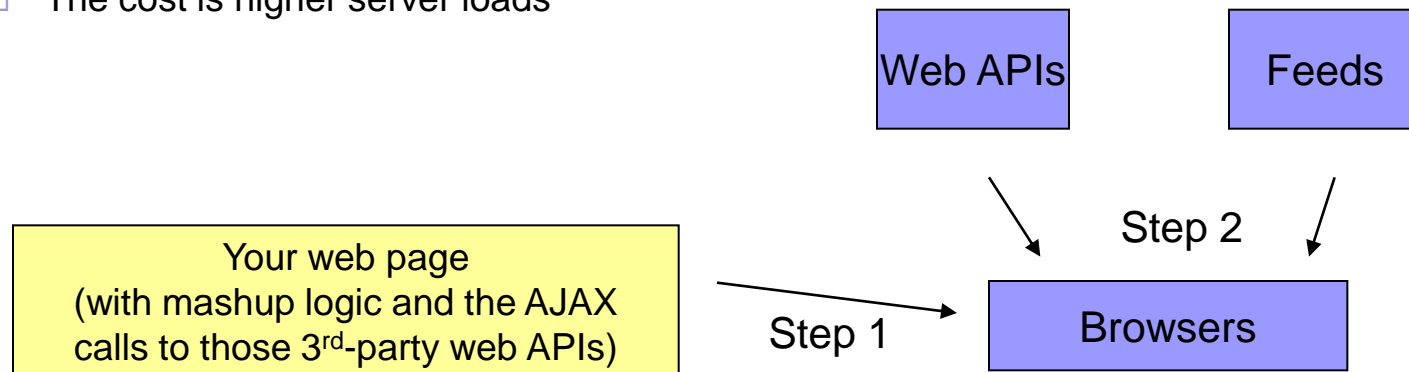
- The data from multiple sources is gathered at the web server side (i.e., the server-side is the **consumer** of those web APIs).
- The combined data (after processed by the mashup logic) are delivered to the clients in a HTML page



# Client-side mashups

The “**same-origin principle**”.

- The data is gathered at the browser, usually done with **AJAX**
- **Note however:** for security, most browsers do **not** allow XMLHttpRequest to connect to websites other than the domain your current webpage is retrieved from.
- There are some workarounds (e.g., IFRAME), but in general the server-side approach is more flexible
  - E.g., you can display a custom page to convey connection error, etc.
  - The cost is higher server loads



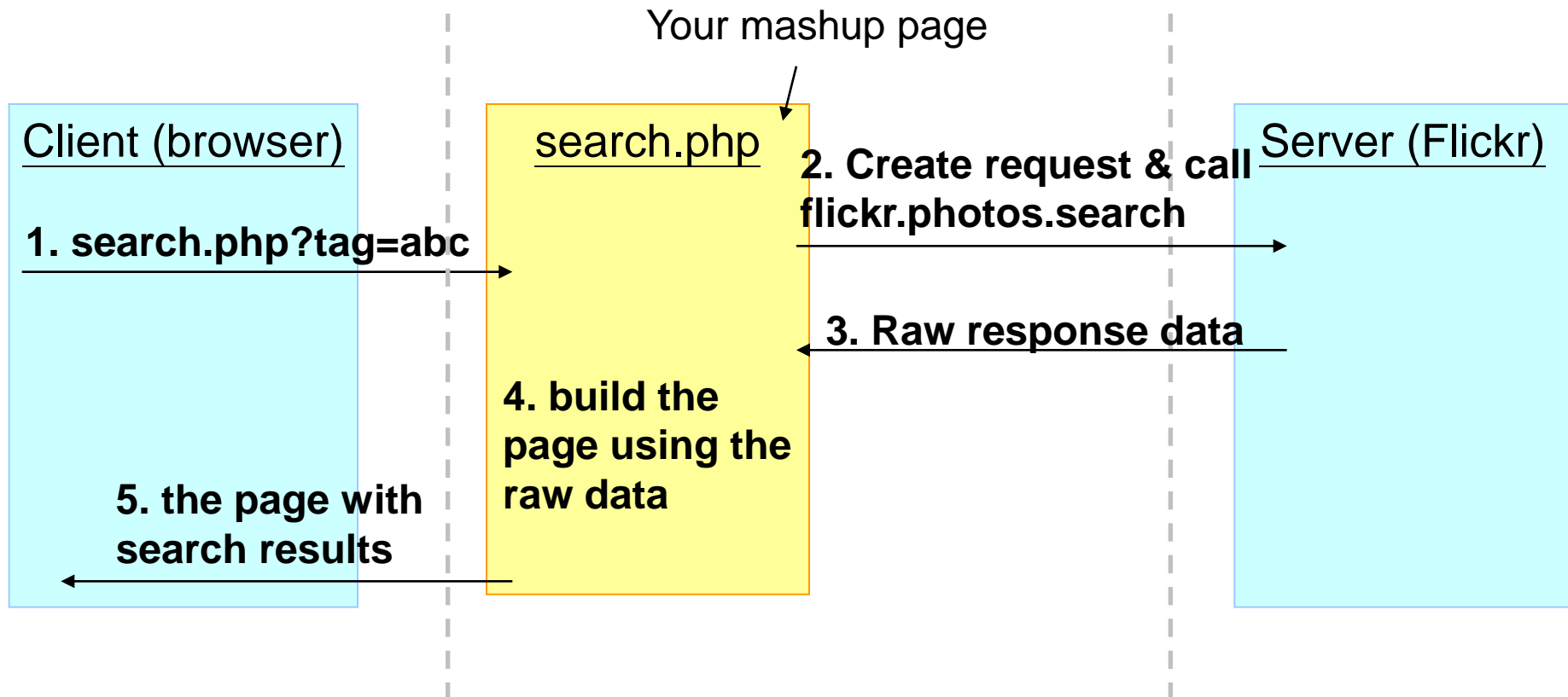
- Example: the **Google Maps application** described in **Session 7** is an example of this **client-side approach**.
- The client-side can be **combined** with the server-side approach
  - E.g., the server retrieves data from outside web APIs, the client connects to your own server to retrieve updates through AJAX
  - => best flexibility but more complicated implementation



# A server-side example: Using Flickr

# Searching photos from Flickr

- Flickr is an online photo-hosting service
- In this example, connection to the web API and data manipulations happen at the **server side**
- Flickr supports XML-RPC, SOAP and REST; we will use REST in this example



## 2. Create request (in REST) & call flickr.photos.search

```
<?php
#
# build the API URL to call
#
```

```
$params = array(
    'api_key'      => 'your_api_key',
    'format'       => 'php_serial',
    'method'       => 'flickr.photos.search',
    'tags'         => 'Mickey Mouse',
    'per_page'     => '10',
    'page'         => '1'
);
```

Calling method and arguments, format result in **php\_serial** (=> so that the result can be “unserialized” into PHP objects)

```
$encoded_params = array();
```

```
foreach ($params as $k => $v) {
    $encoded_params[] = urlencode($k).'='.urlencode($v);
}
```

Escape each (name, value) pair, and put it as name=value

```
#
# call the API and decode the response
#
```

```
$url = "http://api.flickr.com/services/rest/?".implode('&', $encoded_params);
```

```
$rsp = file_get_contents($url);
```

```
$rsp_obj = unserialize($rsp);
```

Build the request URL in REST  
implode(' & ', \$array):  
concatenates each item with '&'

Submit the request and  
“unserialize” the result

## 4. build the page using raw data (eg: showing the photo)

The **unserialized**, raw, response data:

An array of info which can be used to  
reconstruct links to photos

```
[ "photo" ] => array(10) {  
  [0] => array(9) {  
    [ "id" ] => string(10) "2379135927"  
    [ "owner" ] => string(12) "33994435@N00"  
    [ "secret" ] => string(10) "2281e8b620"  
    [ "server" ] => string(4) "2002"  
    [ "farm" ] => float(3)  
    [ "title" ] => string(31) "Mickey Mouse on acid, this one."  
    [ "ispublic" ] => int(1)  
    [ "isfriend" ] => int(0)  
    [ "isfamily" ] => int(0)  
  }  
}
```

+

URL template for a photo

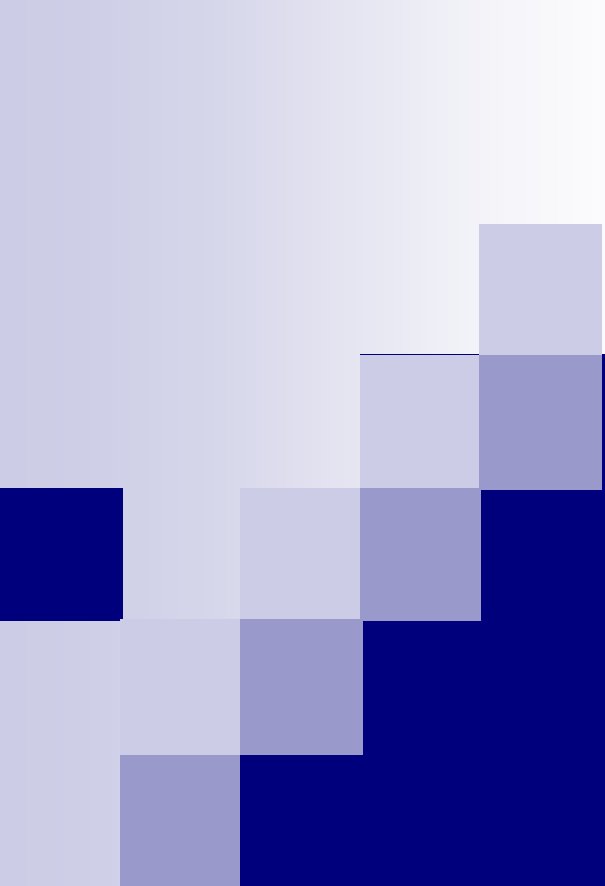
```
http://farm{farm-id}.static.flickr.com/{server-id}/{id}_{secret}.jpg
```

||

Useful output

```

```



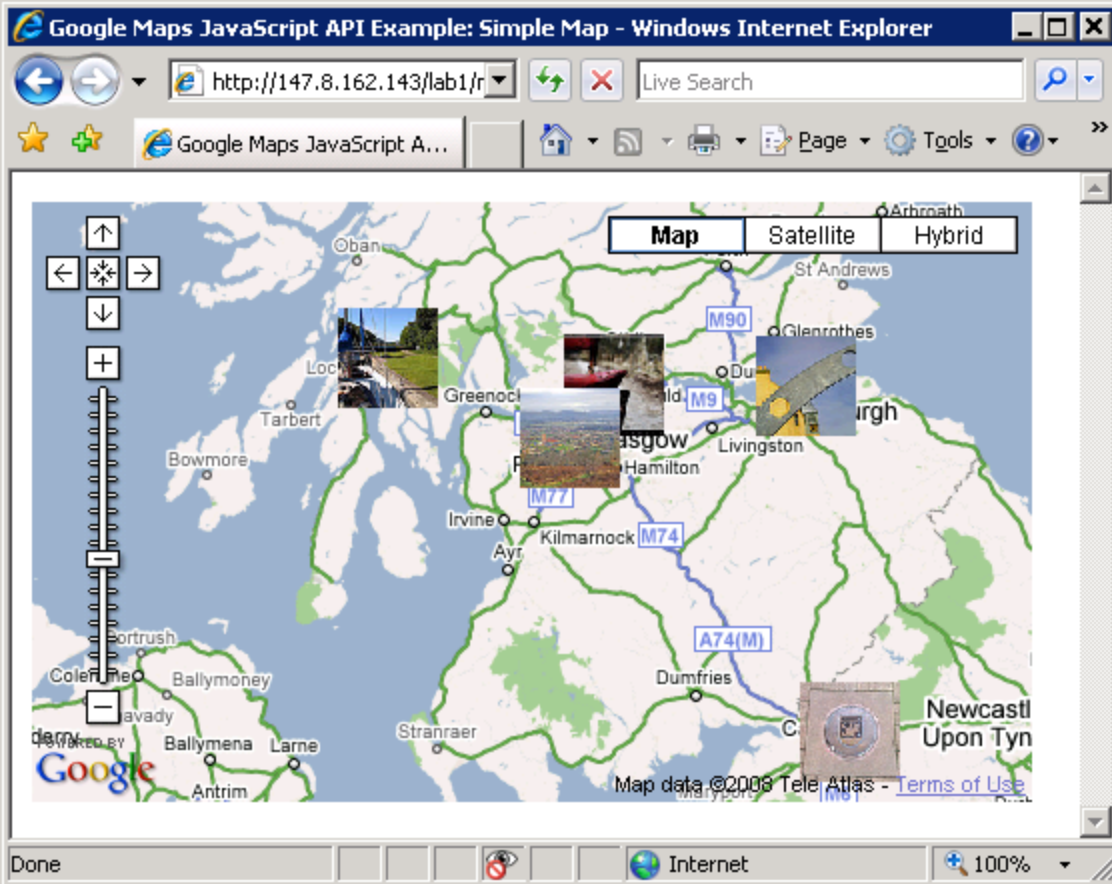
# Client-side + server- side mashups (Google Maps + Flickr)



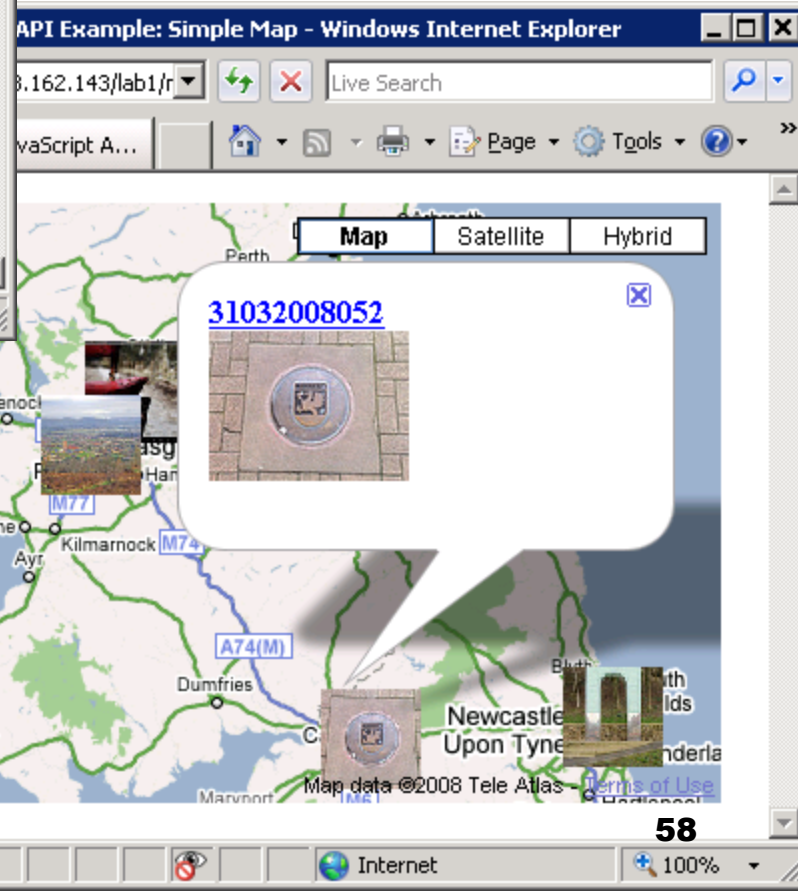
# Example:

## Google Maps + Flickr w/AJAX

- (Note: all program code in this section will not be covered in the exam)
- We will make a mashup by using both the Google Maps and Flickr service
- An initial map will be displayed
- Whenever the user moves (e.g., dragging or zooming) the map:
  - Search Flickr for photos taken within the geographic boundary shown on the map
    - When performing the search, a “loading” overlay is shown on the map
  - Display the photos as thumbnail markers on the map
  - Show photo’s details when the user clicks on a marker

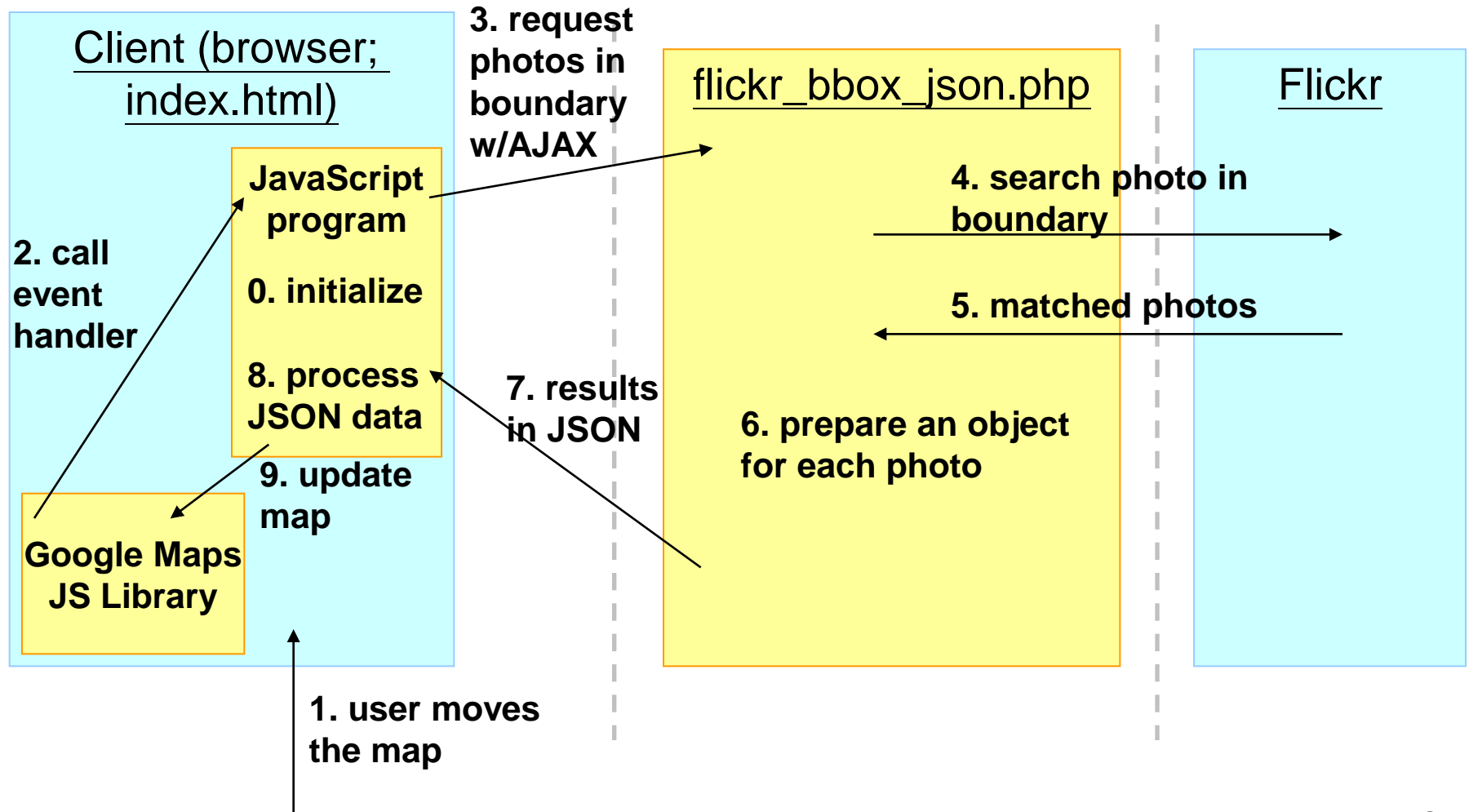


# Example: Google Maps + Flickr w/AJAX



# Example:

## Google Maps + Flickr w/AJAX



# (client) 0: initialize

Basic map setup

```
function initialize() {  
  if (GBrowserIsCompatible()) {  
    map = new GMap2(document.getElementById("map_canvas"));  
    map.setCenter(new GLatLng(51.4, 0), 7);  
    map.addControl(new GLargeMapControl());  
    map.addControl(new GMapTypeControl());  
  
    loading = new GScreenOverlay("loading.png", new GScreenPoint(100, 0),  
    new GScreenPoint(0, 0), new GScreenSize(100, 30));  
  
    GEvent.addListener(map, "dragend", sendRequest);  
    GEvent.addListener(map, "zoomend", sendRequest);  
  
    // send the first search request  
    sendRequest();  
  }  
}
```

Send the first search request to show some photos in the area

Register event handlers

Create a "loading" overlay (but not added to the map yet)

# (client) 1&2: Calls event handler sendRequest when the user moves the map

```
// prepare and send a search request using AJAX
function sendRequest(){
    // get the boundaries of the current view, and prepare the bbox argument
    bounds = map.getBounds();
    sw = bounds.getSouthWest();
    ne = bounds.getNorthEast();
    bbox = sw.lng() + "," + sw.lat() + "," + ne.lng() + "," + ne.lat();

    // destroy any existing XMLHttpRequest objects, to free resources
    if (xmlHttp) delete xmlHttp;

    // get a new XMLHttpRequest object for each request
    xmlHttp = getXMLHTTP();

    // specify the event handler
    xmlHttp.onreadystatechange = updateDisplay;

    // send the request, using GET method
    xmlHttp.open("GET", "flickr_bbox_json.php?bbox=" + bbox);
    xmlHttp.send(null);

    // add the "loading" overlay on map
    map.addOverlay(loading);
};
```

Call updateDisplay if  
readystatechange changes

Here we delete any  
existing XHR object  
and set up a new one

Set up parameters  
and send the  
request. This is an  
AJAX request, so  
the function  
continues without  
waiting

Show the  
"loading"  
overlay now

# (mashup site) 3&4: Send a request to Flickr

```
<?php
// build the API URL to call
$bbox = @$_REQUEST["bbox"];
```

```
$params = array(
    'api_key'      => 'YOUR_API_KEY',
    'format'       => 'php_serial',
    'method'       => 'flickr.photos.search',
    'bbox'         => $bbox,
    'per_page'     => '10',
    'page'         => '1',
    'sort'         => 'interestingness-desc',
    'extras'       => 'geo,title'
);
```

```
$encoded_params = array();
```

```
foreach ($params as $k => $v) {
    $encoded_params[] = urlencode($k).'='.urlencode($v);
}
```

```
// call the API and decode the response
$url = "http://api.flickr.com/services/rest/?".implode('&', $encoded_params);
$rsp = file_get_contents($url);
$rsp_obj = unserialize($rsp);
```

Set the bbox parameter to search for photos within a geographic boundary

Show most “interesting” photos first

Build and send the request to Flickr using REST

(mashup site) 5&6:

## Process the response data

```
$rsp_obj = unserialize($rsp);
```

```
// prepare the photo objects
```

```
$output = array();
```

```
$size = "t";
```

```
if ($rsp_obj['stat'] == 'ok'){
```

```
    // $key here is 0-9 (the original keys for the array $rsp_obj["photos"]["photo"])
```

```
    foreach ($rsp_obj["photos"]["photo"] as $key=>$photo) {
```

```
        $farmid = $photo["farm"];
```

```
        $serverid = $photo["server"];
```

```
        $photoid = $photo["id"];
```

```
        $secret = $photo["secret"];
```

```
        $owner = $photo["owner"];
```

```
        $lat = $photo["latitude"];
```

```
        $lng = $photo["longitude"];
```

```
        $title = $photo["title"];
```

```
        $output[$key] = array();
```

```
        // prepare the necessary URLs and location information
```

```
        $output[$key]["link"] =
```

```
            "http://www.flickr.com/photos/$owner/$photoid/";
```

```
        $output[$key]["icon"] =
```

```
            "http://farm$farmid.static.flickr.com/$serverid/${photoid}_${secret}_s.jpg";
```

```
        $output[$key]["thumb"] =
```

```
            "http://farm$farmid.static.flickr.com/$serverid/${photoid}_${secret}_t.jpg";
```

```
        $output[$key]["lat"] = $lat;
```

```
        $output[$key]["lng"] = $lng;
```

```
        $output[$key]["title"] = $title;
```

**“unserialize” the  
response from Flickr**

**Create an array storing the  
photos, each with the info we  
want: the image URL, site  
URL, geographic location, etc**

## (mashup site) 7: Return a JSON object to the client

```
// convert the array to JSON
// arrays with numeric keys will be arrays
// arrays with named keys will be objects

// for failed searches, output would be empty
echo json_encode($output);
```

- PHP function to convert an array into JSON:  
    `json_encode()`
- echo it



# (client) 8: Process JSON data

```
// this is called whenever a XMLHttpRequest changed its state
function updateDisplay(){
    // 4: Completed
    if (xmlHttp.readyState == 4) {
        // re-create the object from JSON, using eval
        response = eval(xmlHttp.responseText);
        // clear all markers and the "loading" overlay on map
        map.clearOverlays();
        if (response.length > 0) {
            for (i = 0; i < response.length; i++) {
                // for each photo, create a marker and add to the map
                map.addOverlay(createMarker(response[i]));
            }
        }
    }
};
```

**This block executes only when the request is finished**

**Response is now an array of photo objects**

**Clears all markers on the map, including the "loading" overlay**

**For each photo, add a marker to the map. createMarker() creates a GMarker object from an item in the response**

# (client) 9: Update display (map)

```
// create a marker for a photo (item)
function createMarker(item) {
    // create a GIcon using the photo's icon image
    var myicon = new GIcon(G_DEFAULT_ICON, item.icon);
    myicon.iconSize = new GSize(50, 50);

    // Options for the marker: icon and title
    mopt = {
        icon: myicon,
        title: item.title
    };

    // create a marker at the photo's location (item.lat, item.lng)
    var marker = new GMarker(new GLatLng(item.lat, item.lng), mopt);

    // a "click" event handler for the marker
    // to display an InfoWindow containing a bigger photo and its title
    GEvent.addListener(marker, "click", function() {
        var details = "<b><a target='_blank' href='" + item.link + "'>";
        details += item.title + "</a></b><br>";
        details += "<img src='" + item.thumb + "'><br>&nbsp; ";
        marker.openInfoWindowHtml(details);
    });
    return marker;
}
```

**Create a marker and set its position and other properties (icon image, title)**

**Return this marker to updateDisplay**

**Set up a click handler for the marker to display the photo's details**

# Summary

## ■ SOAP and REST compared

### □ SOAP's benefits:

- Higher flexibility in defining the API interface, interaction patterns and better vendor support
- Some consider that SOAP is better for closed systems running within an enterprise

### □ REST's benefits:

- Use of widely-deployed, highly-optimized server/client software; caching is supported, great scalability and interoperability, etc.
- No need to formulate service “contracts”

### □ The importance of limiting the number of “verbs” in protocol designs

## ■ How to share enterprise data through web APIs using Java EE (JAX-WS for SOAP and JAX-RS for RESTful APIs)

## ■ RPC-based protocols as a lightweight form of SOAP

### □ XML-RPC and JSON-RPC

## ■ Web API mashups

### □ Motivations: data combination, visualization and aggregation

### □ Examples of mashup

- Server-side: using Flickr
- Integration (client+server-sides): Flickr + Google Maps

# Post-class readings

- Post-class readings:

- Take a look at the list of mashups at [Programmableweb.com](http://Programmableweb.com)

- Reference materials:

- The Java EE 7 Tutorial (JAX-WS and JAX-RS are covered in Part VI “Web Services”)

- R. Yee. Pro Web 2.0 mashups: remixing data and web services (e-book). Apress. 2008.

- This book is rather old, but the concepts of how web API data is combined to form value-added services is still applicable today.

- Please see Moodle for the links.



Reminder:  
Group project

# Reminder

- Group project:
  - For those who haven't grouped, please form the groups by tomorrow (Feb 20, 2021) and send your names to Steven. If you plan to work on your own, please also let him know or he may help you to form groups later.
  - Please send a draft of the Group Report (i.e., the topic of your website, technologies and web APIs used, job distribution, etc.) to Steven by March 6, 2021 for approval.
- If you have any questions, please feel free to post to Moodle or email Steven or me.



# Optional consultation session

# Optional consultation sessions

## Feb 21 (2pm-5pm) and Mar 6 (2pm-5pm)

- Will be conducted at P6-03, Graduate House. Steven and I will be there.
- **Optional** – for those who have questions on the labs, the assignment/project or any other course materials. You may skip these sessions if you don't have any questions.
- You may come in person, **or** join our online Zoom meeting.
- Please take a look at the labs and the assignment before the consultation sessions so that we can discuss if you have any questions about them.
- **For those who have voted for Mar 6:** please consider also joining us on Feb 21 (this Sunday) so that we can discuss your questions earlier. The session on Mar 6 will be held anyway - so you may join both sessions if you like.
- For the Zoom meeting:
  - You may share your screen with us, or grant us the remote access (through Zoom) if you want us to look into the problem you have encountered.
  - We will discuss with the students one at a time, please be patient while you are waiting at the virtual “waiting room”.
- Please read the announcement (at Moodle) for other detail.