Service Oriented Computing: Restful Services

Dr. Cristian Mateos Diaz
(http://users.exa.unicen.edu.ar/~cmateos/cos)
ISISTAN - CONICET

Representational state transfer (REST)

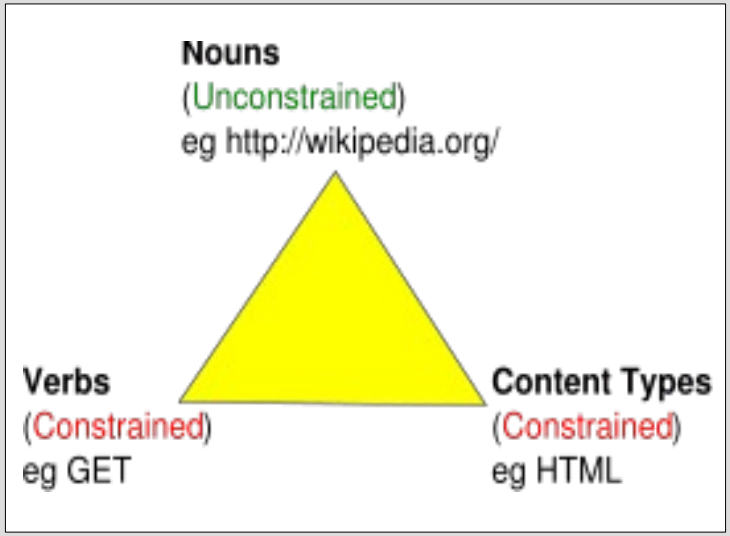
- REST is a set of network architecture principles which outline how resources of a distributed hypermedia system are defined and addressed:
  - *Resources* (sources of specific information), each of which are referred through URIs
  - *Components* of the network, which interchange *representations* of resources (e.g. the circle example)
  - *Layering*: Interactions are stateless, but caching is allowed
  - Applications interact with a resource by knowing its URI and specifying an *action*; the application must understand the returned representation
- The WWW is a *restful* system

REST: Unix pipes for the Web

- Mostly, Unix-style programming has the following characteristics:
  - Do one thing, and do it well
  - Everything is a file
  - Comprise complex systems by connecting smaller, simpler programs (e.g., Unix pipes)
- The *everything-is-a-file* abstraction worked for Unix because there was a small set of common operations that applied to files (open, close, read, write)...
  - Doing REST is just exploit HTTP and its standard HEAD, GET, PUT, POST, DELETE verbs.

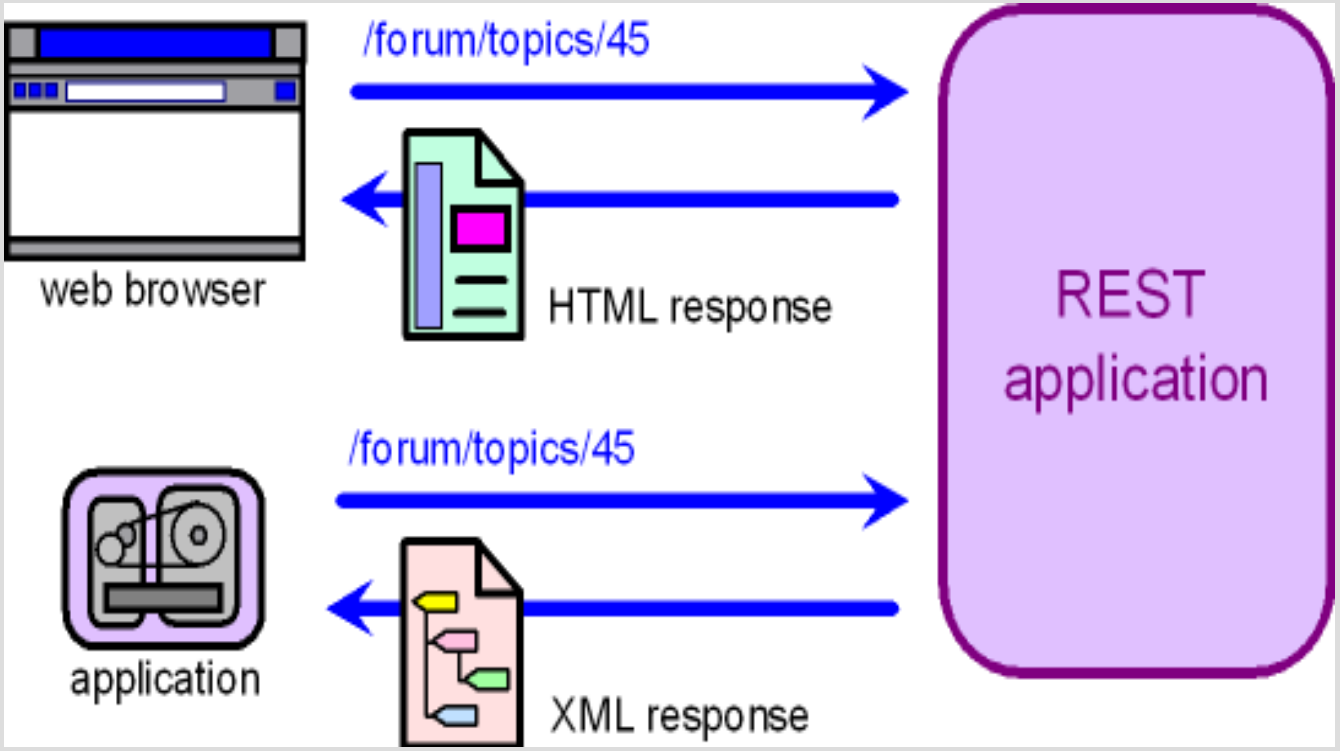
REST vs RPC

- First off, let us please left philosophy aside!
- REST (resources >> actions)
  - Resource-oriented; actions are simple, standard and constrained
  - Noun-oriented; only resource representations are exchanged



- RPC (commands > services)
  - Command-oriented; commands are defined in terms of varying complexity
  - Verb-oriented; only method calls are issued

REST vs RPC (cont.)



REST vs RPC (cont.)

- An RPC Web Service might define the following operations:
  - getUser()/addUser()/removeUser()/updateUser()
  - getLocation()/addLocation()/removeLocation()/updateLocation()
  - listUsers()/listLocations()
  - findLocation()/findUser()

- Client code to access these services might look like:

```
serviceEndpointProxy = new InfoService(“nowhere.org:8180”);
serviceEndpoint.addUser(“001”, “John Doe”);
```

- How could we model the same application (both client and “service”) with REST principles?

REST vs RPC (cont.)

- The server might define the following resources:
  - http://example.com/users/
  - http://example.com/users/{user} (one for each user)
  - http://example.com/findUserForm
  - http://example.com/locations/
  - http://example.com/locations/{location}
  - http://example.com/findLocationForm
  - *http://example.com/getUser?id=001 is in principle disallowed*

- Client code to access these resources might look like:

```
userRes = new Resource('http://example.com/users/001”);
representation = userRes.get();
userRes.delete();
```

RESTful Web Services: Brief state of the art

- Axis 2/CXF/WSDL 2.0/JAX-RS support restful Web Services
- REST-specific service description languages: WADL, Swagger
- Many sites are currently using REST: Google, Flickr, Amazon, Twitter, ...
- For example, Twitter allows to:
  - Resource **followers/ids** (GET)
  - Resource **followers/list** (GET)
  - Resource **friendships/destroy** (POST)

RESTful Web Services: The Restlet framework

- Reslet (http://www.restlet.org) is a Java library for RESTful SOC

- Example: Creating a service (a lot of necessary server-side configuration has been omitted – e.g., where the root points to)

```
public class FirstServerResource extends ServerResource {
   public static void main(String[] args) throws Exception {
      new Server(Protocol.HTTP, 8182, FirstServerResource.class).start();
   }
   @Get
   @Produces("text/plain")
   public String toString() { return "hello, world"; }
}
```

- Example: Creating a client

```
ClientResource resource = new ClientResource("http://localhost:8182");
// Customize the referrer property
resource.setReferrerRef("http://www.mysite.org");
// Write the response entity on the console
resource.get().write(System.out);
```

RESTful Web Services: Alternative frameworks

- Jersey (https://jersey.dev.java.net)
  - **Reference implementation** of **JAX-RS**
  - Shipped with Glassfish
  - Spring and Guice integration

- NetKernel (http://www.1060research.com)
  - An implementation of the REST paradigm and architecture for building **any** system beyond Web applications
  - E.g. local file repositories, data storage, etc.

RESTful Web Services: Alternative frameworks (cont.)

- Spring (www.springframework.org)

```
@Controller
@RequestMapping("/people")
public class PersonController {
  private PersonDao personDao;

  @Autowired
  public void setPersonDao(PersonDao personDao) {
    this.personDao = personDao;
  }

  @RequestMapping(method = RequestMethod.GET)
  public People getAll() {
    return new People(personDao.getPeople());
  }

  @RequestMapping(value = "/{id}", method = RequestMethod.GET)
  @ResponseBody
  public Person getPerson(@PathVariable("id") Long personId) {
    return personDao.getPerson(personId);
  }

  @RequestMapping(method = RequestMethod.POST)
  public View savePerson(@RequestBody Person person) {
    personDao.savePerson(person);
    return new RedirectView("/people/" + person.getId());
  }
}
```

Content-type can be then set...

Annotation-based SOC: JSR 181 and beyond

- A considerable effort has been done towards standardizing the use of annotations to code/consume Web Services
  - Web Service annotations bundled into Java SE
- JSR 181 (Web Services Metadata) Annotations:
  - javax.jws.WebService, javax.jws.WebMethod, javax.jws.OneWay, javax.jws.WebParam, javax.jws.WebResult, javax.jws.HandlerChain (intercepting WS calls/results), javax.jws.soap.SOAPBinding
- https://jcp.org/en/jsr/detail?id=181

Annotation-based SOC: JSR 181 and beyond (cont.)

```
@WebService(endpointInterface="example.ICreditService", targetNamespace="http://
example.org/creditCards")

public class CreditService implements example.ICreditService {

    public CreditService() {}

    @WebResult(name="validateReturn",
    targetNamespace="http://example.org/creditCards")

    public boolean validate(

        @WebParam(name="number",
        targetNamespace="http://example.org/creditCards") int number,

        @WebParam(name="code",
        targetNamespace="http://example.org/creditCards") int code) {

        if (Math.random() < .9) {return true;}

            return false;

}
```

Annotation-based SOC: JSR 181 and beyond (cont.)

- JSR 224 (JAX-WS) Annotations
  - Standard annotations needed by JAX-WS (the new specification for invoking services in Java) but not defined in JSR 181
  - Developers may not ever use these annotations directly as some of them are generated by JAX-WS tools
  - Examples: javax.xml.ws.RequestWrapper, javax.xml.ws.ResponseWrapper
- JSR 222 (JAXB) Annotations
  - Control the XML-Java mapping
  - javax.xml.bind.annotation.XmlRootElement, javax.xml.bind.annotation.XmlAccessorType, javax.xml.bind.annotation.XmlTransient, javax.xml.bind.annotation.XmlElement, ...

Annotation-based SOC: JAX-WS annotations example

```
...
public interface AddNumbersImpl implements AddNumbersIF {

    @WebMethod
    @WebResult(targetNamespace = "")
    @RequestWrapper(localName = "addNumbers", targetNamespace =
            "http://server.fromjava/", className = "fromjava.client.AddNumbers")
    @ResponseWrapper(localName = "addNumbersResponse",
            targetNamespace = "http://server.fromjava/", className =
            "fromjava.client.AddNumbersResponse")
    public int addNumbers(
        @WebParam(name = "arg0", targetNamespace = "")
        int arg0,
        @WebParam(name = "arg1", targetNamespace = "")
        int arg1)
        throws AddNumbersException_Exception {
            return arg0 + arg1;
     }
}
```

Annotation-based SOC: JAXB annotations example

```
@XmlRootElement(name="addNumbers",
    namespace="http://server.fromjava/jaxws")
// FIELD, unlike PROPERTY, avoids calling getters/setters during marshal/unmarshal
@XmlAccessorType(AccessType.FIELD)
public class AddNumbers {
    @XmlElement(namespace="", name="number1")
    public int number1;
    @XmlElement(namespace="", name="number2")
    public int number2;

    public AddNumbers(){}
    // getters/setters
}
```

Annotation-based SOC: JAX-RS

- JAX-RS is to Restful service what JAX-WS is to SOAP-based services

- More than 15 annotations to:

  - Set application and resource paths
  - Exploiting HTTP operations
  - Specifying content-types
  - Reading query and form parameters

- Anyway, framework support for such “standard” annotations is partial!
https://dzone.com/articles/7-reasons-i-do-not-use-jax-rs-in-spring-boot-web-a

Annotation-based SOC: JAX-RS (cont.)

```
// http://localhost:8080/webcontext/api/ where “webcontext” is the entire
// application name
 @ApplicationPath("/api")
 public class RESTConfig extends Application {}


// By convention, the container resource is named in plural
// http://localhost:8080/webcontext/api/books
@Path("/books")
public class BookResource {}
```

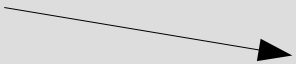
Now, we need to associate HTTP operations with methods...

Annotation-based SOC: JAX-RS (cont.)

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response getAllBooks() {
    List<Book> books = BookRepository.getAllBooks(); // queries database
    GenericEntity<List<Book>> list = new GenericEntity<List<Book>>(books) {};
    return Response.ok(list).build();
}
```

MediaType, GenericEntity and Response are from javax.ws.rs.core

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response saveBook(Book book) {
    book = bookRepository.saveBook(book);
    return Response.ok(book).build();
}
```



@PUT would be associated to an 'updateBook' method

Annotation-based SOC: JAX-RS (cont.)

```
@DELETE
// Matches e.g. http://localhost:8080/api/books/0-201-63361-2
@Path("{isbn}")
@Produces(MediaType.APPLICATION_JSON)

public Response deleteBook(@PathParam("isbn") String isbn) {
    Book book = bookRepository.deleteBookByIsbn(isbn);
    return Response.ok(book).build();
}
```

Annotation-based SOC: JAX-RS (cont.)

```
@DELETE
// Matches e.g. http://localhost:8080/api/books/0-201-63361-2
@Path("{isbn}")
@Produces(MediaType.APPLICATION_JSON)

public Response deleteBook(@PathParam("isbn") String isbn) {
    Book book = bookRepository.deleteBookByIsbn(isbn);
    return Response.ok(book).build();
}
```

But you use **@QueryParameter** for pairs key=value in the query string:

```
@GET
@Produces(MediaType.APPLICATION_JSON)
// e.g. http://localhost:8080/api/books/search?keyword=Java&limit=10
@Path("search")
public Response searchBook(@QueryParam("keyword") String keyword,
@QueryParam("limit") int limit) {
    List<Book> books = bookRepository.searchBook(keyword, limit);
    return Response.ok(new GenericEntity<List<Book>>(books) {}).build();
}
```

Annotation-based SOC: JAX-RS (cont.)

- **@FormParam**: Just like **@QueryParameter**, but for form fields

- **@CookieParam**, to read cookies values (HTTP is stateless!)

- **@HeaderParam**, to read HTTP header info (e.g. “user-agent”)

- **@QueryParameter** versus **@MatrixParameter**
  - http://some.where/thing?paramA=1&paramB=6542 versus http://some.where/thing;paramA=1;paramB=6542
  - URLs with "?" are not cached in many cases; URLs with matrix params are cached
  - However, less frameworks support **@MatrixParameter**

RESTful Web Services: WADL

```
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://research.sun.com/wadl/2006/10 wadl.xsd"
    xmlns:tns="urn:yahoo:yn" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:yn="urn:yahoo:yn" xmlns:ya="urn:yahoo:api"
    xmlns="http://research.sun.com/wadl/2006/10">
    <grammars>
        <include href="NewsSearchResponse.xsd"/>
        <include href="Error.xsd"/>
    </grammars>
    <resources base="http://api.search.yahoo.com/NewsSearchService/V1/">
        <resource path="newsSearch">
            <method name="GET" id="search">
              <!-- Contents shown in the next slide -->
            </method>
        </resource>
    </resources>
</application>
```

RESTful Web Services: WADL (cont.)

```
<method>
  <request>
    <param name="appid" type="xsd:string" style="query" required="true"/>
    <param name="query" type="xsd:string" style="query" required="true"/>
    <param name="type" style="query" default="all">
      <option value="all"/>
      <option value="any"/>
      <option value="phrase"/>
    </param>
    <param name="results" style="query" type="xsd:int" default="10"/>
    <param name="start" style="query" type="xsd:int" default="1"/>
    <param name="sort" style="query" default="rank">
      <option value="rank"/>
      <option value="date"/>
    </param>
    <param name="language" style="query" type="xsd:string"/>
  </request>
  <response>
    <representation mediaType="application/xml" element="yn:ResultSet"/>
    <fault status="400" mediaType="application/xml" element="ya:Error"/>
  </response>
</method>
```

RESTful Web Services: WADL (cont.)

WADL has been however subject of some criticisms:

- Using WADL leads to “RPCized” service specifications
- WS-* must be implemented on top of JSON
- WADL might not be necessary at all (http://bitworking.org/news/193/Do-we-need-WADL)
- This does not mean RESTful should be reconsidered...!

RESTful Web Services: WADL (cont.)

As a response, some **gradual** documentation/specification languages have arisen:

- Swagger (swagger.io, now Open API)
  - Used with JSON/YAML (superset of JSON)
  - Large community/good tooling support
- API blueprint (apiblueprint.org)
  - Based on Markdown
  - Somewhat newer; good tooling
- RAML (raml.org)
  - Based on YAML
  - The most recent one; little tooling; pattern-sharing

- http://apievangelist.com/2014/01/16/api-design-do-you-swagger-blueprint-or-raml/

RESTful Web Services: YAML and API blueprint

```
title: World Music API
baseUri: http://example.api.com/v1
traits:
  - paged:
      queryParameters:
        pages:
          description: # of pages to return
          type: number
```

```
/songs:
  is: [ paged ]
  get:
    queryParameters:
      genre:
        description: filter the songs by genre
  /{songId}:
    get:
      responses:
        200:
          body:
            application/json:
              schema: |
                { "$schema": "http://json-schema.org/schema",
                  "type": "object",
                  "description": "A canonical song",
                  "properties": {
                    "title":  { "type": "string" },
                    "artist": { "type": "string" }
                  },
                  "required": [ "title", "artist" ]
                }
```

YAML in a nutshell:

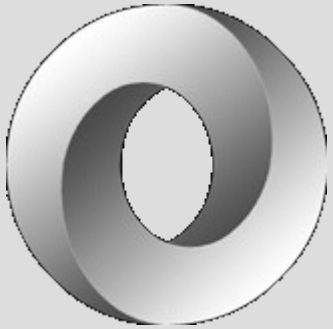
```
one:
  echo
  hello, world!
two: [ echo, "hello, world!" ]
```

RESTful Web Services: JSON (http://www.json.org)

- JSON (JavaScript Object Notation) is a lightweight data-interchange format

  - Similar goals to that of SOAP XML

  - Easy to read (humans), parse and generate

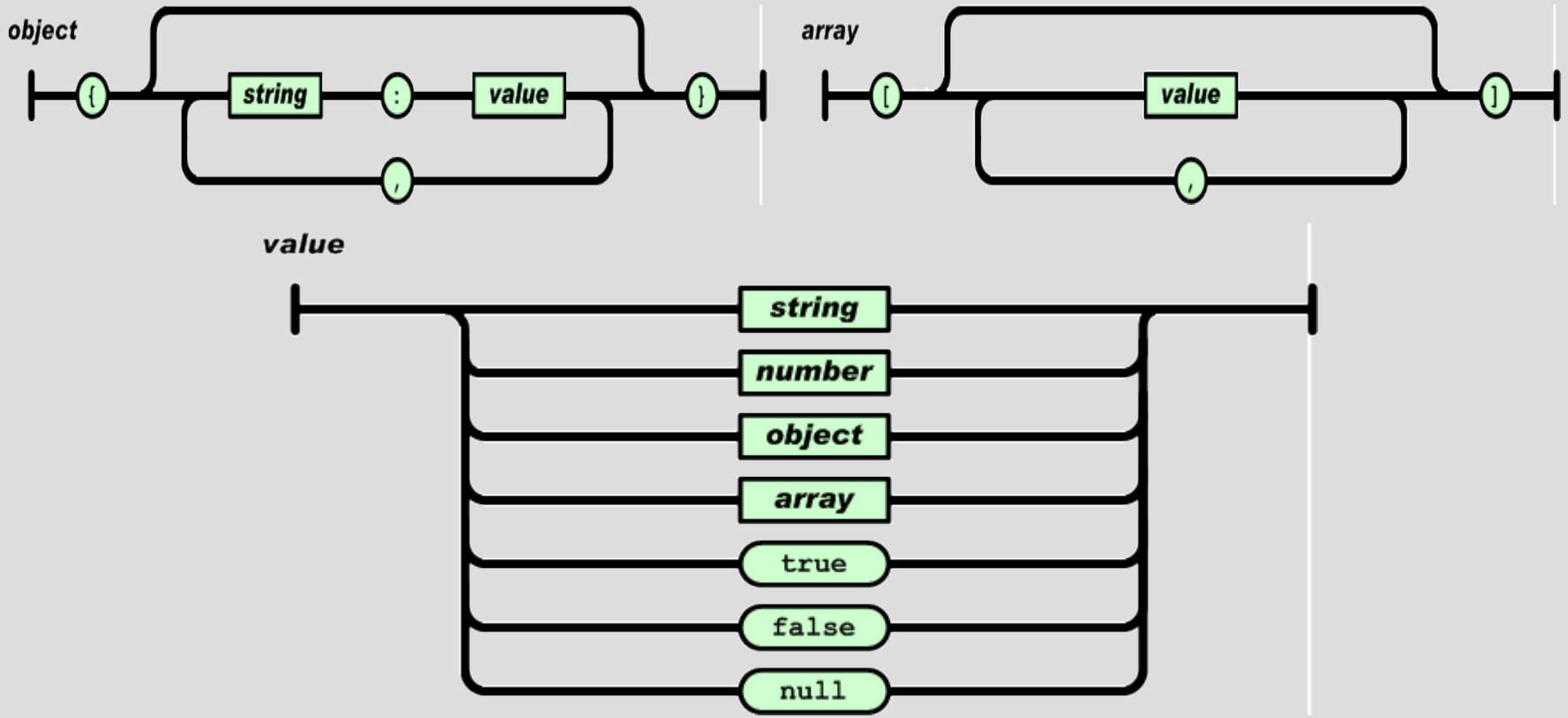


- JSON is built on two structures:

  - A collection of name/value pairs (i.e., object, record, struct, dictionary, hash table, keyed list, or associative array)

  - An ordered list of values (i.e., array, vector, list, or sequence)

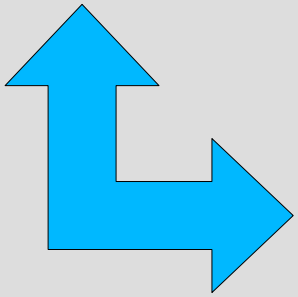
RESTful Web Services: JSON (cont.)



RESTful Web Services: JSON (cont.)

```
<menu id="file" value="File">
  <popup>
    <menuitem value="New" onclick="CreateNewDoc()" />
    <menuitem value="Open" onclick="OpenDoc()" />
    <menuitem value="Close" onclick="CloseDoc()" />
  </popup>
</menu>
```

XML



JSON

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

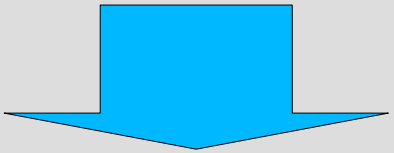
RESTful Web Services: JSON (cont.)

- Some Java-based tools supporting JSON exist: StAX parsers, marshallers, mapping libraries, tags libs, ...
- Example: Json-lib (http://json-lib.sourceforge.net)

```
List list = new ArrayList();
list.add( "JSON" );
list.add( "1" );
list.add( "2.0" );
list.add( "true" );
JSONArray jsonArray = (JSONArray)
JSONSerializer.toJSON( list );
```

RESTful Web Services: JSON versus XML

- ✔ More compact -> less network usage (mobile devices!)
- ✔ Simple, fixed structure -> faster parsing
- ✔ Cleaner semantics since not though to be a metalanguage
- ✘ May not be suitable for semantically-structured data
- ✘ XML is heavily adopted, many tools already exists



Simplicity versus expressivity