JAVA DAVE ENVIRONMENT

Web services, SOAP, Hibernate & More…

Java Boot Camp, August, 2017.

Dr. Kishore Biswas (Forrest/柯修)

PhD. Artificial Intelligence—Natural Language Processing.

CanWay IT Training ®.

CONTENTS

1) SOAP web service (continued..)
2) Restful web service
3) Spring MVC example & practice…
4) Connecting to Oracle database (issues…)

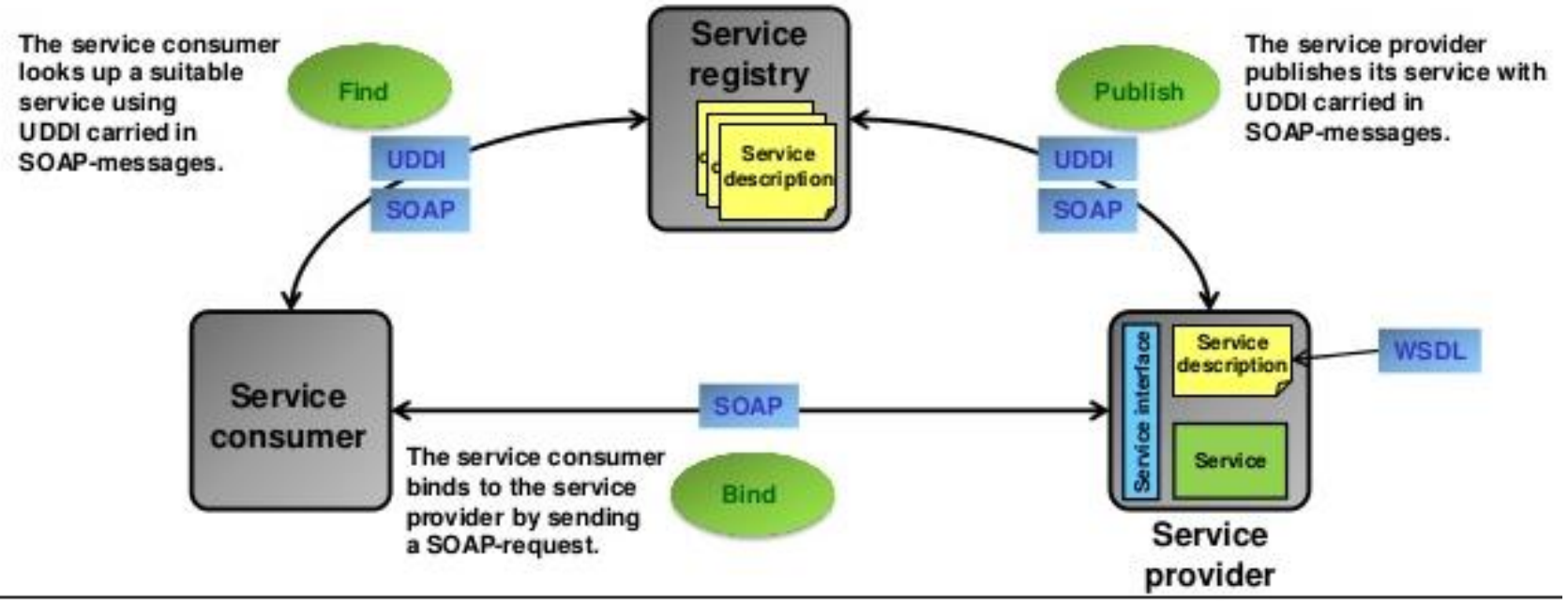
1) SOAP EXAMPLE

SOAP-WSDL-UDDI-SEI

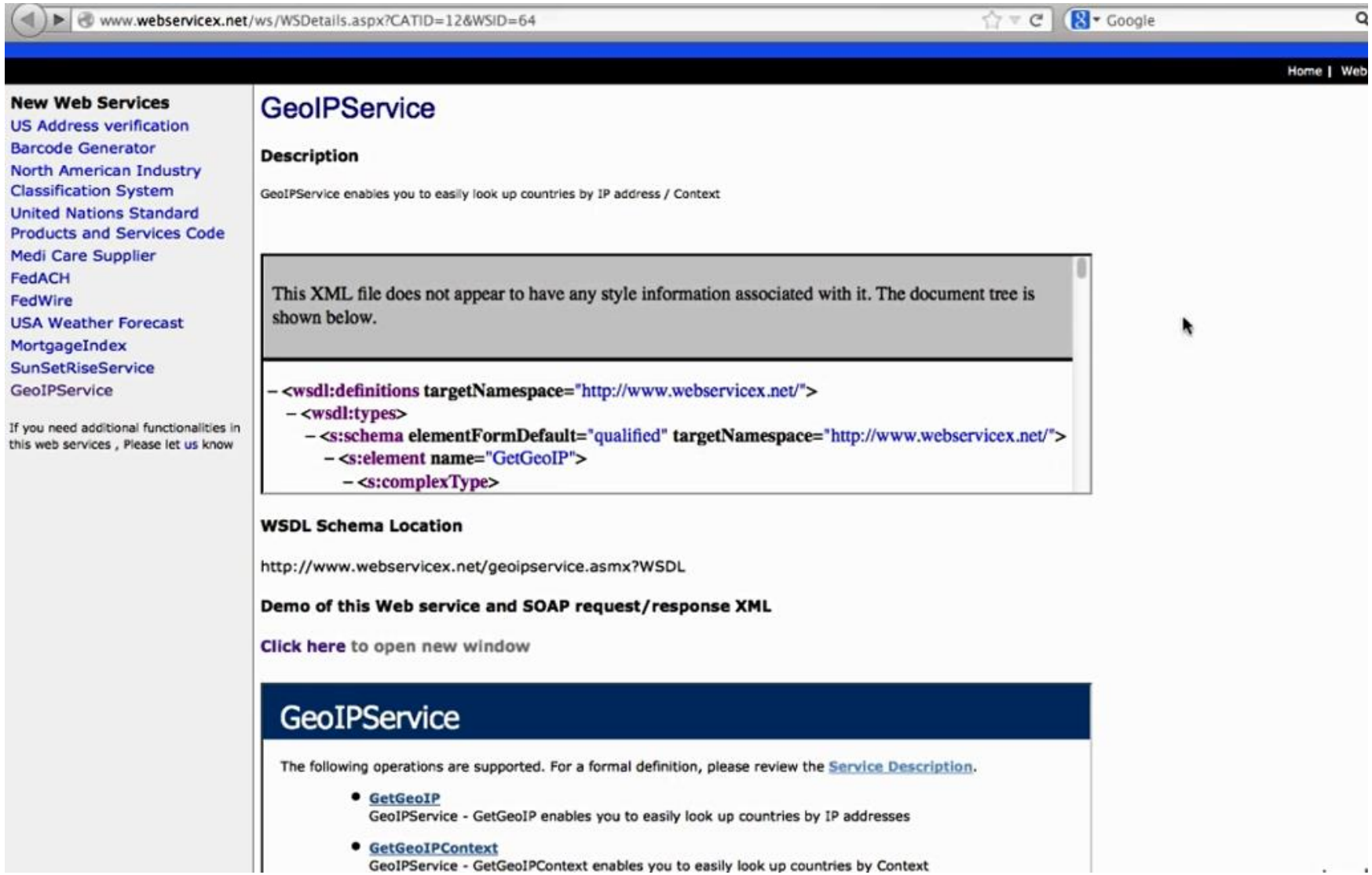
Web service architecture

The combo SOAP+WSDL+UDDI defines a general model for a web service architecture.

SOAP:	Simple Object Access Protocol
WSDL:	Web Service Description Language
UDDI:	Universal Description and Discovery Protocol
Service consumer:	User of a service
Service provider:	Entity that implements a service (=server)
Service registry:	Central place where available services are listed and advertised for lookup



HTTP://WWW.WEBSERVICEX.NET/WS/WSDETAILS.ASPX?CATID=12&WSID=64



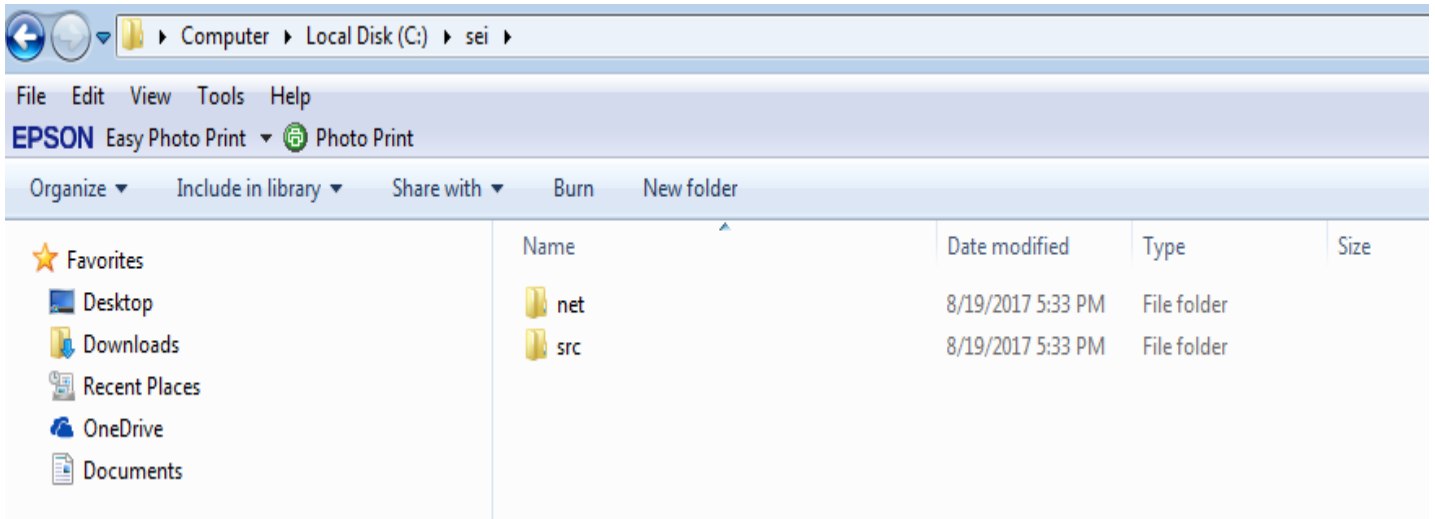
http://www.webservicex.net/

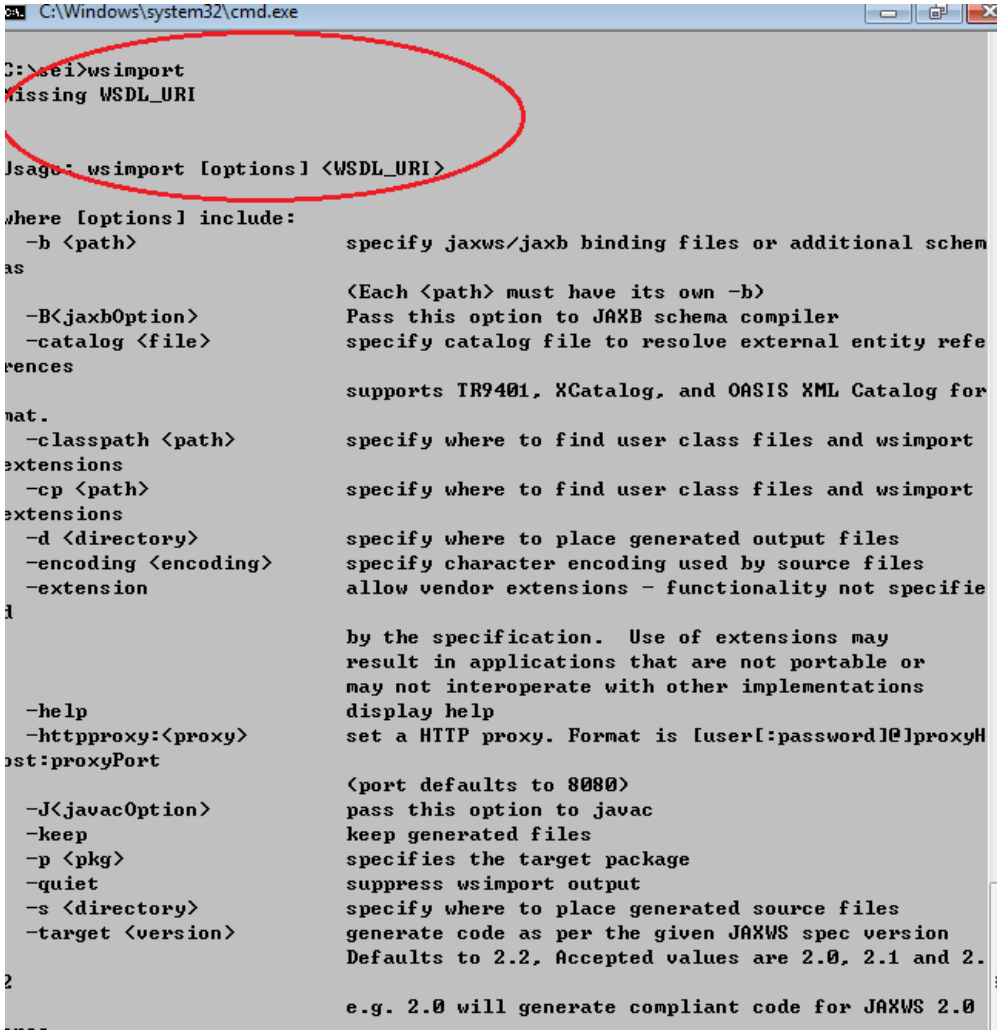
http://www.webservicex.net/geoipservice.asmx?wsdl

```
    </wsdl:operation>
  </wsdl:binding>
- <wsdl:binding name="GeoIPServiceHttpPost" type="tns:GeoIPServiceHttpPost">
    <http:binding verb="POST"/>
  - <wsdl:operation name="GetGeoIP">
      <http:operation location="/GetGeoIP"/>
    - <wsdl:input>
        <mime:content type="application/x-www-form-urlencoded"/>
      </wsdl:input>
    - <wsdl:output>
        <mime:mimeXml part="Body"/>
      </wsdl:output>
    </wsdl:operation>
  - <wsdl:operation name="GetGeoIPContext">
      <http:operation location="/GetGeoIPContext"/>
    - <wsdl:input>
        <mime:content type="application/x-www-form-urlencoded"/>
      </wsdl:input>
    - <wsdl:output>
        <mime:mimeXml part="Body"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
- <wsdl:service name="GeoIPService">
  - <wsdl:port name="GeoIPServiceSoap" binding="tns:GeoIPServiceSoap">
      <soap:address location="http://www.webservicex.net/geoipservice.asmx"/>
    </wsdl:port>
  - <wsdl:port name="GeoIPServiceSoap12" binding="tns:GeoIPServiceSoap12">
      <soap12:address location="http://www.webservicex.net/geoipservice.asmx"/>
    </wsdl:port>
  - <wsdl:port name="GeoIPServiceHttpGet" binding="tns:GeoIPServiceHttpGet">
      <http:address location="http://www.webservicex.net/geoipservice.asmx"/>
    </wsdl:port>
  - <wsdl:port name="GeoIPServiceHttpPost" binding="tns:GeoIPServiceHttpPost">
      <http:address location="http://www.webservicex.net/geoipservice.asmx"/>
    </wsdl:port>
  </wsdl:service>
```

CALLING A WEB SERVICE IN JAVA PROGRAM

1) Code in Java
2) Check UDDI, look for available web services,
3) In Java home mkdir sei
4) Check wsimport
5) sei > mkdir src
6) Run → wsimport –keep – s src
   http://www.webservicex.net/geoipservice.asmx?WSDL





PRACTICE EXERCISE

- Write java client program that consumes a free web service to implement simple translator program.

- http://www.webservicex.net/WS/WSDetails.aspx?CATID=12&WSID=63

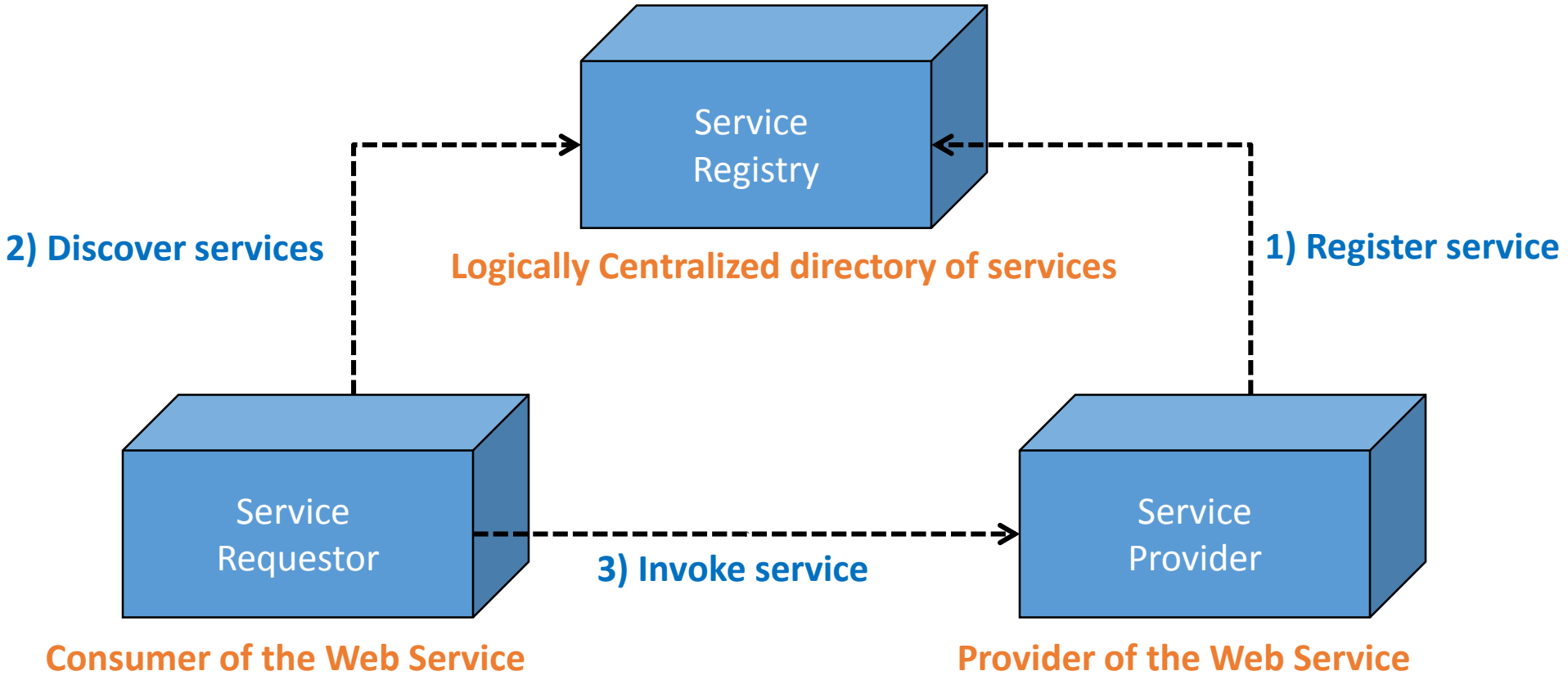
2) RESTful Web Service

What is a Web Service

- A Web Service is any server application that:
- Is available over the web/HTTP
- Is OS and Programming language independent
- Is application to application talk.

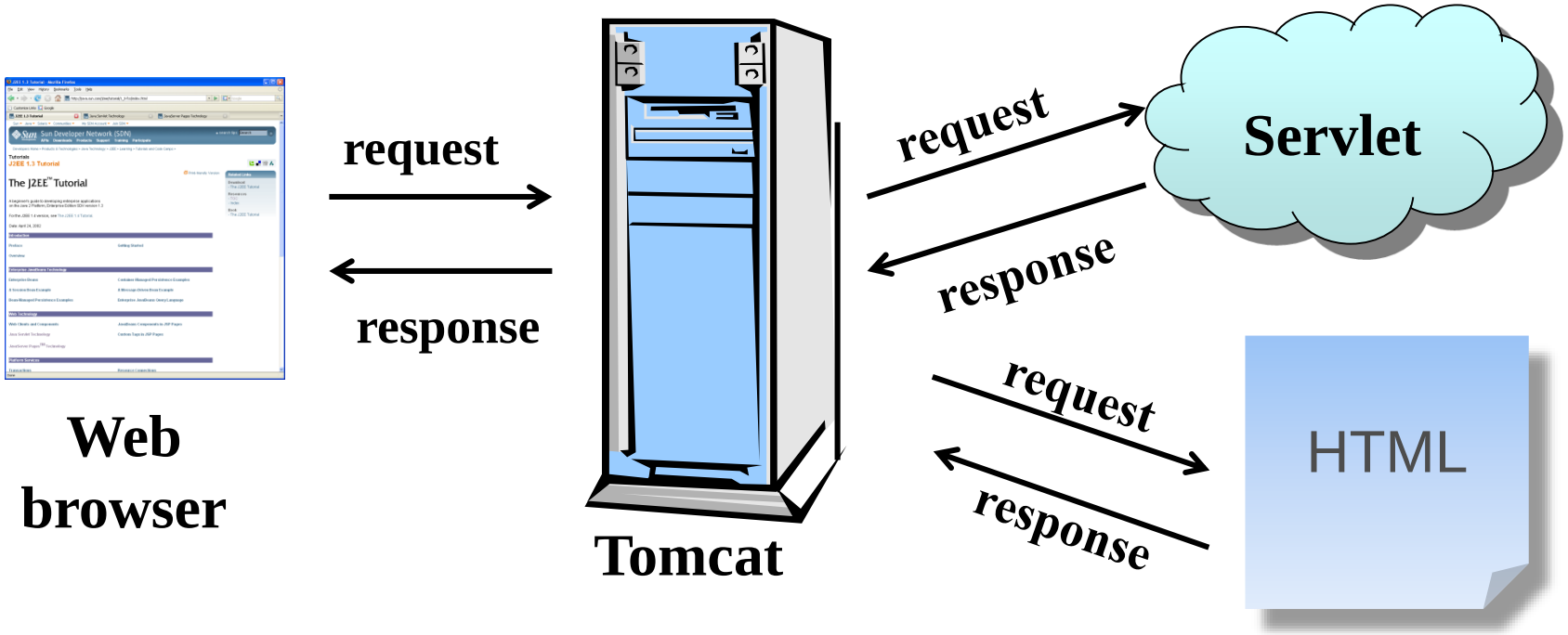
Web Service Roles

There are three major roles



JEE – TOMCAT

- Tomcat is a JEE Servlet Container.
- It can run regular web sites but also Java code.
- Just like any other web server it handles the HTTP protocol (requests and responses).



TYPES OF WEB SERVICES

- Web services can be implemented in various ways. The two most popular methods of implementation are:
- **SOAP** web services
- **RESTful** web services

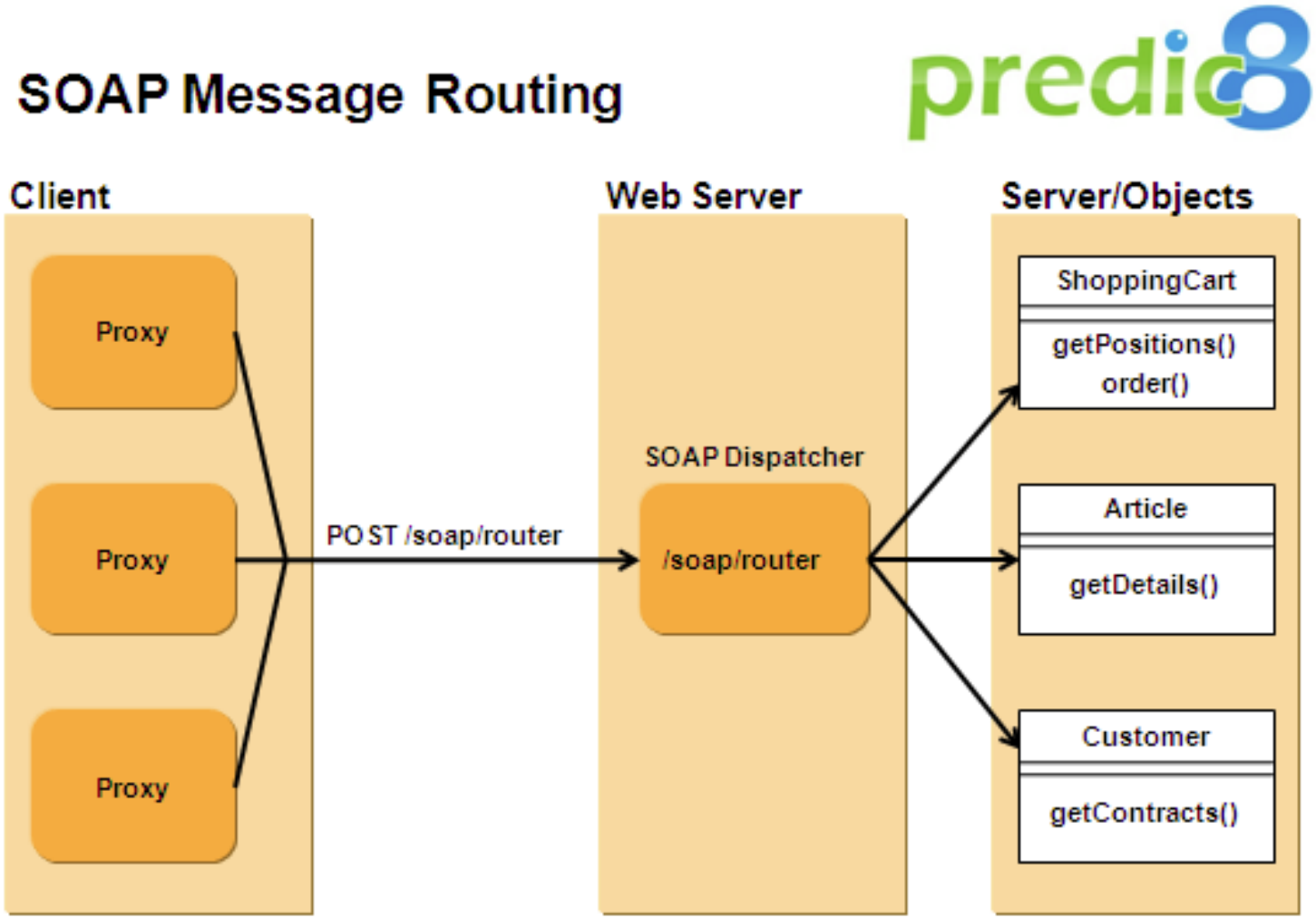
SOAP WEB SERVICE

- What is SOAP?
- **S**imple **O**bject **A**ccess **P**rotocol
- Is a protocol that uses XML messages to perform **RPC** (**R**emote **P**rocedure **C**alls), meaning, call a function on another (usually remote) program
- Requests are encoded in XML and send via HTTP
- Responses are encoded in XML and received via HTTP

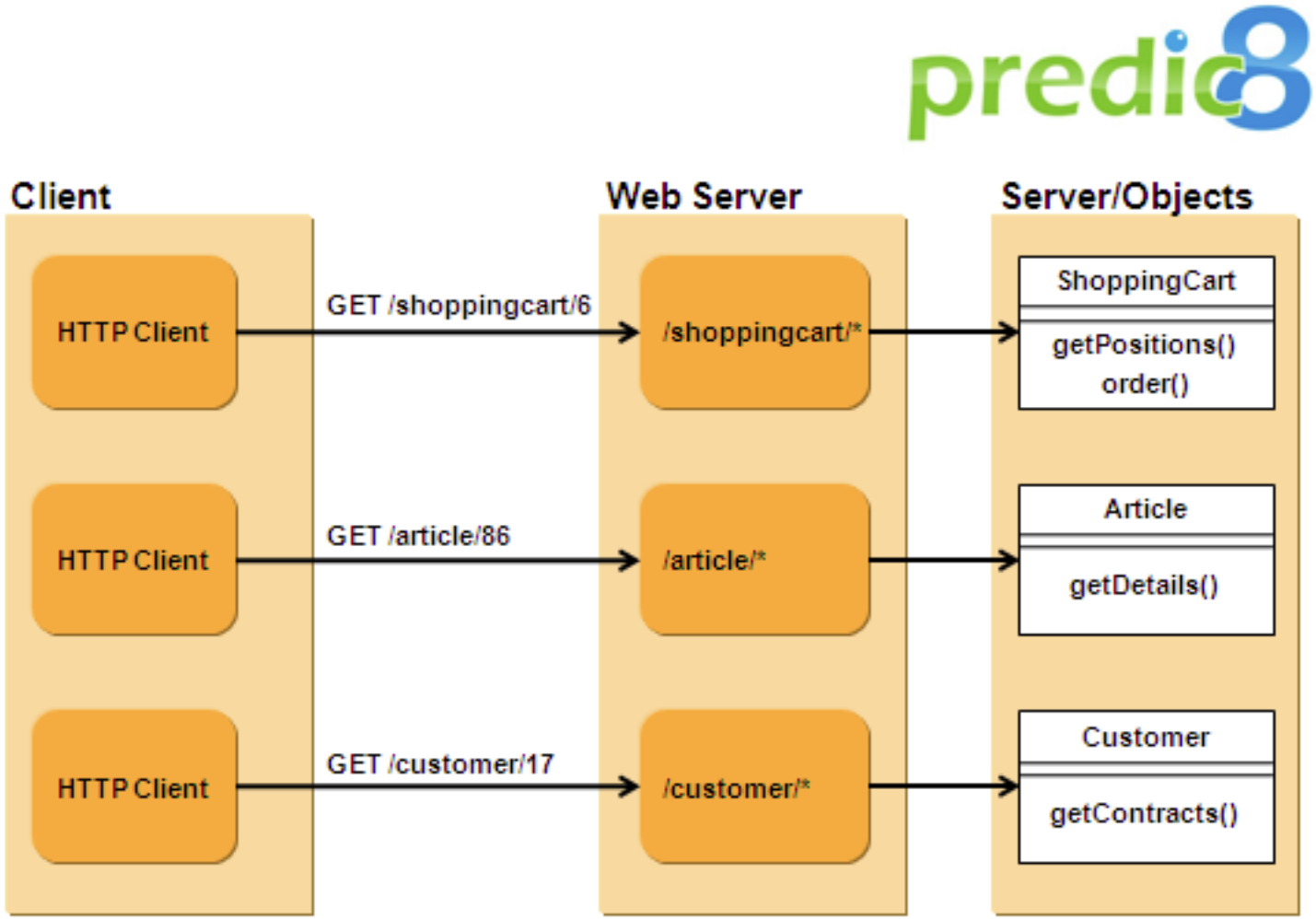
RESTful Web Services

- RESTful web services (**Re**presentational **S**tate **T**ransfer) use existing well-known W3C standards (HTTP, XML, URI, MIME) and have a lightweight infrastructure that allows services to be built with minimal tooling.
- Developing RESTful web services is inexpensive and thus has a very low barrier for adoption.
- REST is well suited for basic scenarios. RESTful web services better integrated with HTTP than SOAP-based services are, do not require XML messages or WSDL service–API definitions.
- In Java EE 6, JAX-RS provides the functionality for RESTful web services.

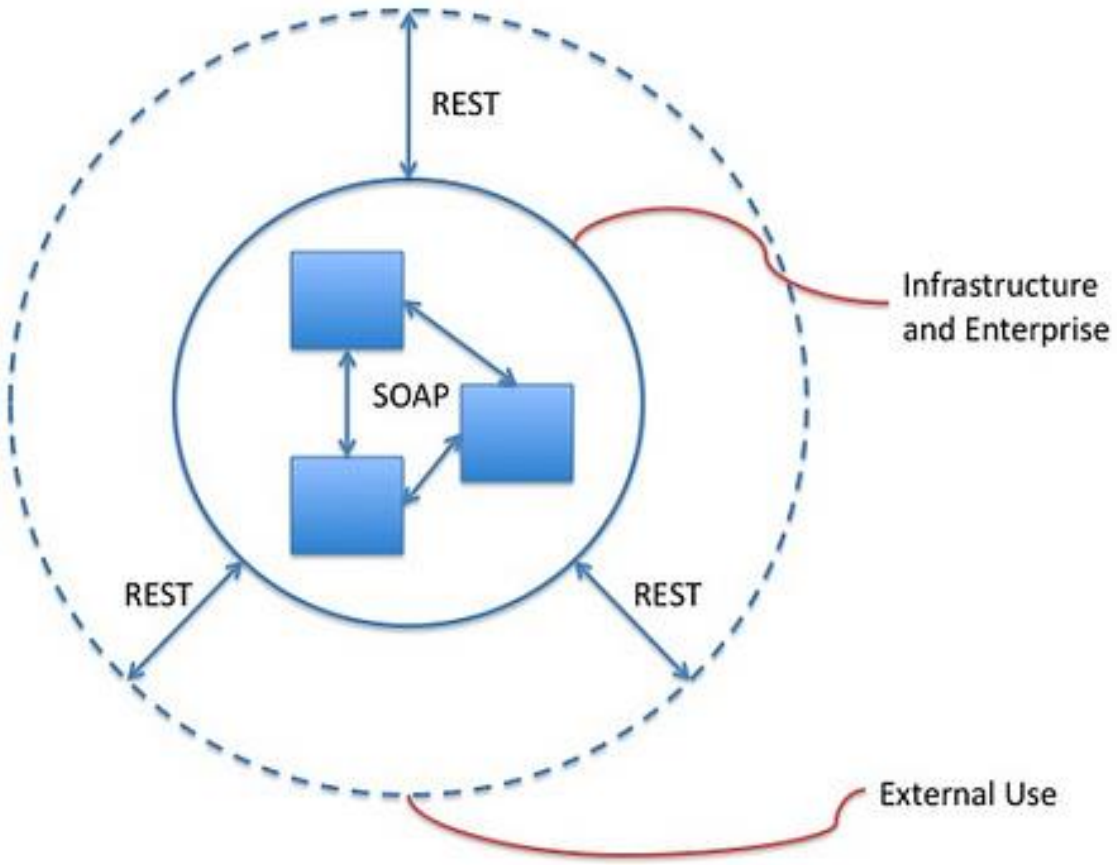
SIMPLE OBJECT ACCESS PROTOCOL (SOAP) WEB SERVICE



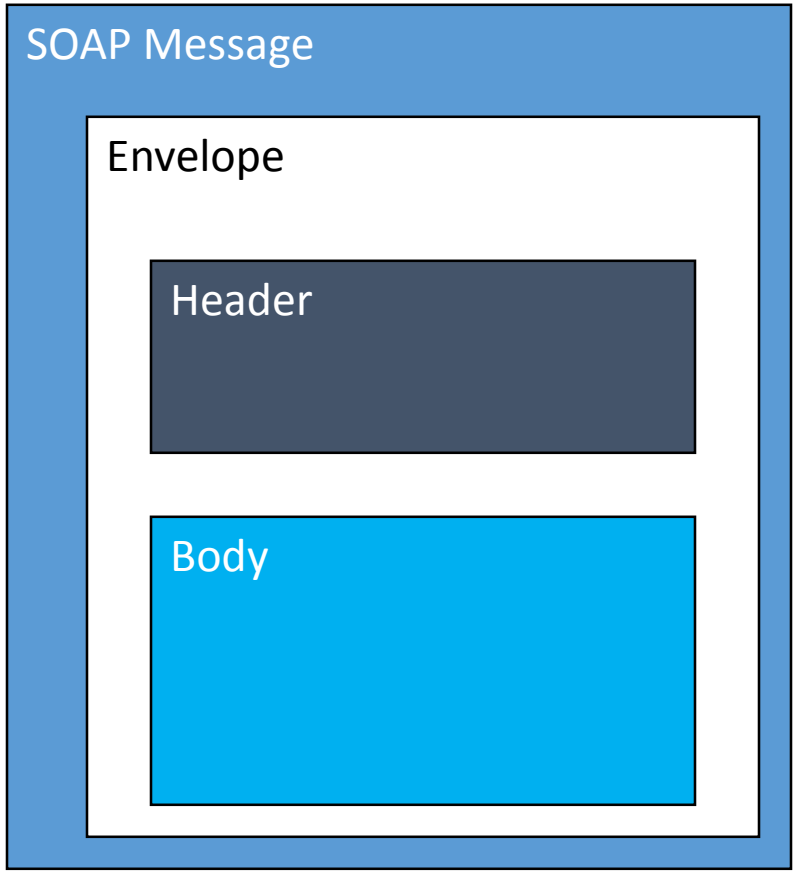
REPRESENTATIONAL STATE TRANSFER (REST) WEB SERVICE



REST Vs. SOAP


REST
Infrastructure
and Enterprise
SOAP
REST
REST
External Use

SOAP MESSAGE



- Envelope is like a wrapper for content
- Header is a optional element that could contain control information
- Body element includes requests and responses
- Body element can also include a Fault element in case of an error

SAMPLE SOAP REQUEST

```
<SOAP-ENV:Envelope
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <SOAP-ENV:Body>
                <ns1:sayHello
                 xmlns:ns1="http://agram.com/">
                    <name xsi:type="xsd:string">Java</name>
                </ns1:sayHello>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SAMPLE SOAP RESPONSE

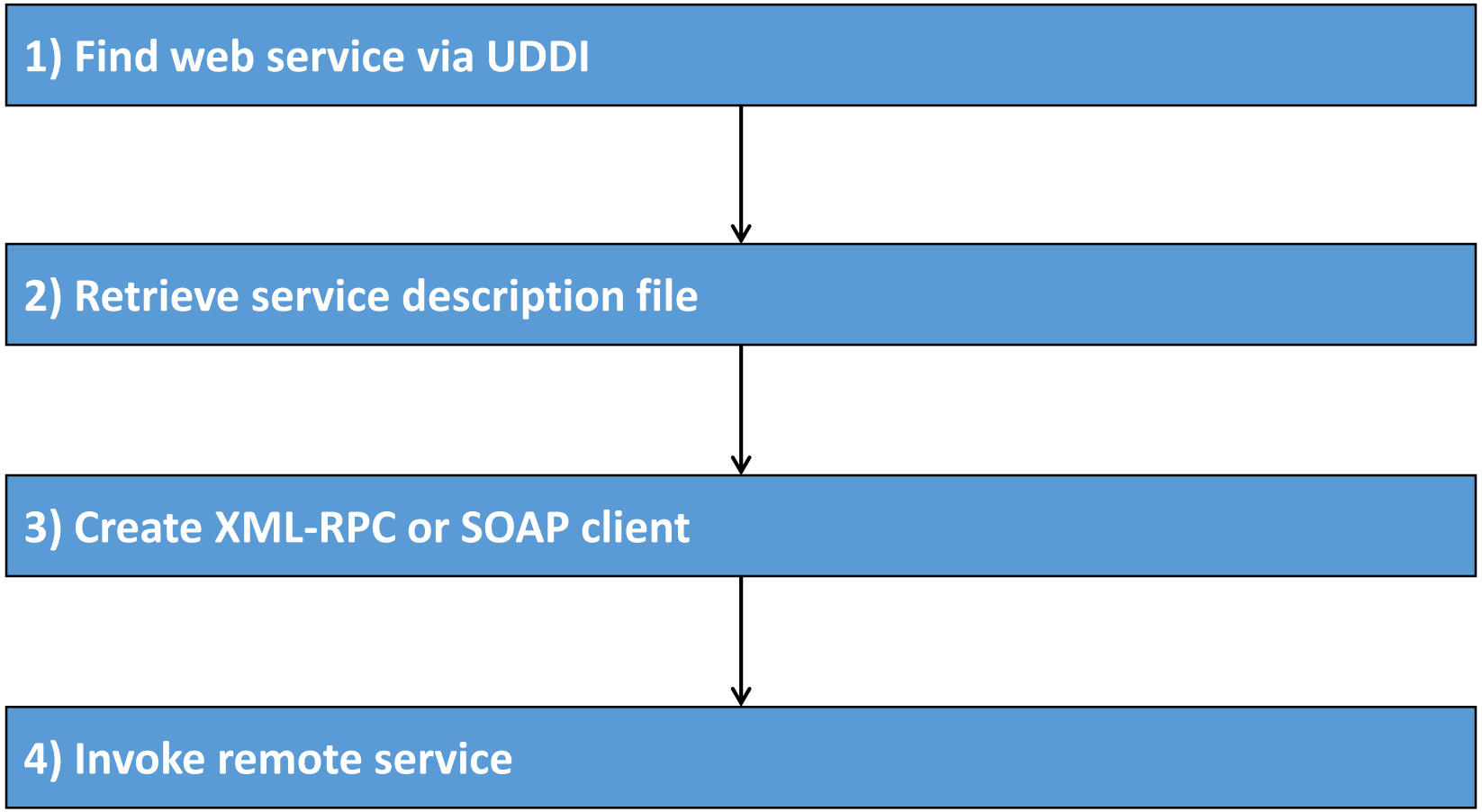
```
<SOAP-ENV:Envelope
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <SOAP-ENV:Body>
                <ns1:sayHelloReponse
                 xmlns:ns1="http://agram.com/">
                    <result xsi:type="xsd:string">Hello Java</result>
                </ns1:sayHelloResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP SERVICE - WSDL

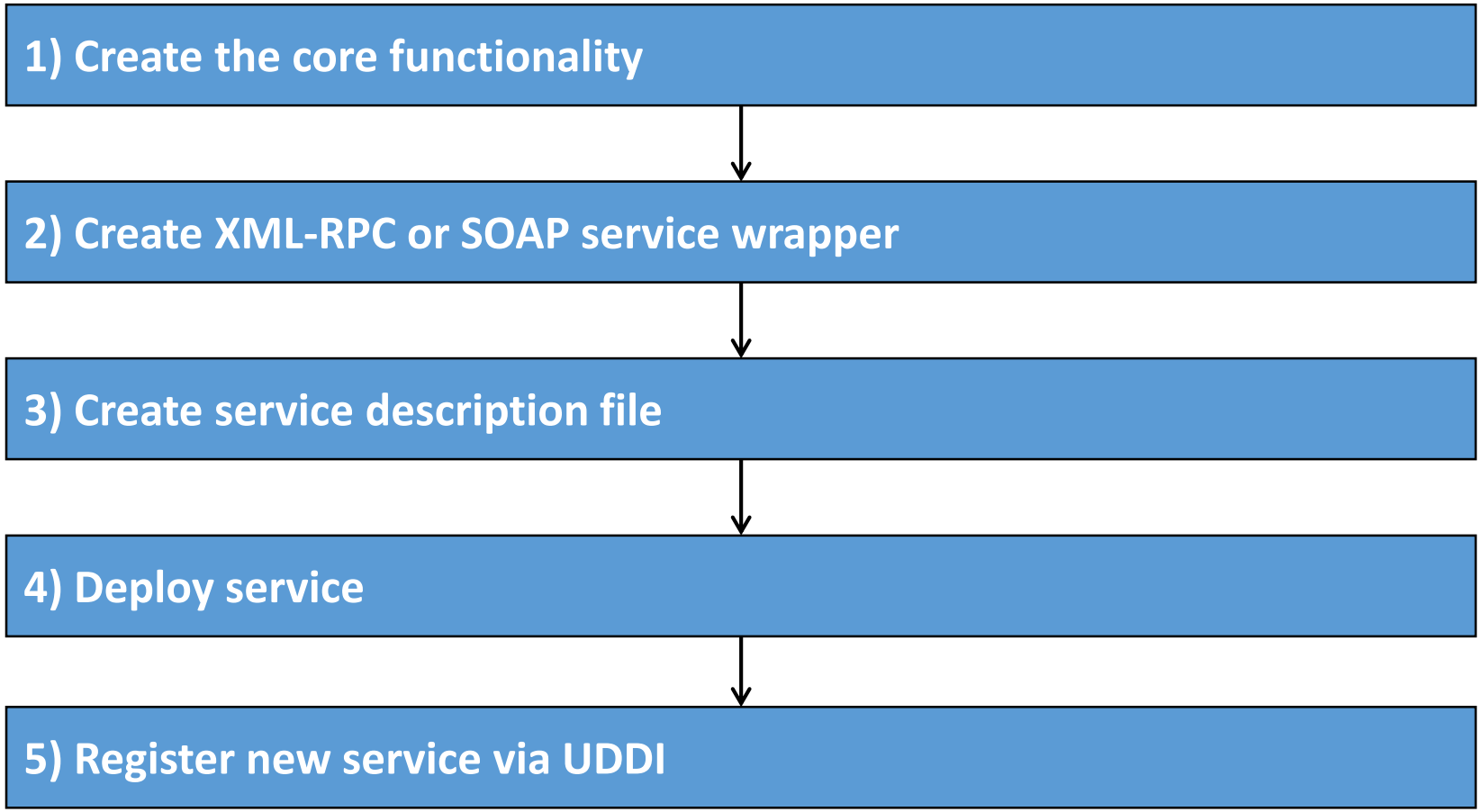
- What is WSDL?
- **W**eb **S**ervices **D**escription **L**anguage
- Has 6 major elements:

1. **Definitions** – defines the name of the web service
2. **Types** – describes all the data types that will be transmitted
3. **Message** – defines the name of the message that will be transmitted
4. **PortType** – defines the operations
5. **Binding** – defines how the message will be transmitted
6. **Service** – defines where the service is located

DEVELOPMENT PLAN FOR SERVICE REQUESTOR



DEVELOPMENT PLAN FOR SERVICE PROVIDER



NOW LETS GO FOR RESTFUL WEB SERVICE

- A RESTful Web Service is a Web Service that binds the HTTP methods (GET, POST, DELETE & PUT) to server methods.
- In this manner, there is no need to send XML messages since the basic methods are already defined in the HTTP protocol.
- It is usually used to expose a CRUD (**C**reate, **R**ead, **U**pdate and **D**elete) functionality of the server.
- The parameters of an HTTP request are used as parameters for the server methods.

Introduction to REST…

An architectural style (extends client-server) introduced by Roy Fielding

REST stands for Representational State Transfer

- It is an architectural ***pattern*** for developing web services as opposed to a ***specification***.
- REST web services communicate over the HTTP specification, using HTTP vocabulary:
  - Methods (GET, POST, etc.)
  - HTTP URI syntax (paths, parameters, etc.)
  - Media types (xml, json, html, plain text, etc)
  - HTTP Response codes.

INTRODUCTION TO REST

- Representational
  - Clients possess the information necessary to identify, modify, and/or delete a web resource. Supports all data representations, eg: xml, JSON.
- State
  - All resource state information is stored on the client.
- Transfer
  - Client state is passed from the client to the service through HTTP.

- Therefore, REST is a web development architecture where Multiple representation of data are supported and by which the state of data is being transferred.

IN A NUTSHELL

- REST is about resources and how to represent resources in different ways.
- REST is about client-server communication.
- REST is about how to manipulate resources.
- REST offers a simple, interoperable and flexible way of writing web services that can be very different from other techniques.
- Comes from Roy Fielding’s Thesis study.

REST IS NOT!

- Unlike SOAP (Simple Object Access Protocol) REST is NOT A protocol.
- A standard.
- A replacement for SOAP.

REST IS…

- **Re**presentational **S**tate **T**ransfer
- Architectural style (technically not a standard)
- Idea: a network of web pages where the client progresses through an application by selecting links
- When client traverses link, accesses new resource (i.e., transfers state)
- Uses existing standards, e.g., HTTP
- REST is an architecture all about the Client-Server communication.

HTTP-REST REQUEST BASICS

- The **HTTP request** is sent *from the client*.
  - Identifies the location of a **resource**.
  - Specifies the **verb**, or HTTP **method** to use when accessing the resource.
  - Supplies optional **request headers** (name-value pairs) that provide additional information the server may need when processing the request.
  - Supplies an optional **request body** that identifies additional data to be uploaded to the server (e.g. form parameters, attachments, etc.)

HTTP-REST REQUEST BASICS

Sample Client Requests:

- A typical client GET request:

```
GET /view?id=1 HTTP/1.1
User-Agent: Chrome
Accept: application/json
[CRLF]
```

Requested Resource (path and query string)

Request Headers

(no request body)

```
POST /save HTTP/1.1
User-Agent: IE
Content-Type: application/x-www-form-urlencoded
[CRLF]
name=x&id=2
```

Requested Resource (typically no query string)

Request Headers

Request Body (e.g. form parameters)

- A typical client POST request:

HTTP-REST RESPONSE BASICS

- The **HTTP response** is sent *from the server*.
  - Gives the **status** of the processed request.
  - Supplies **response headers** (name-value pairs) that provide additional information about the response.
  - Supplies an optional **response body** that identifies additional data to be downloaded to the client (html, xml, binary data, etc.)

HTTP-REST RESPONSE BASICS

Sample Server Responses:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1337
[CRLF]
<html>
  <!-- Some HTML Content. -->
</html>
```

Response Status

Response Headers

Response Body (content)

```
HTTP/1.1 500 Internal Server Error
```

Response Status

```
HTTP/1.1 201 Created
Location: /view/7
[CRLF]
Some message goes here.
```

Response Status

Response Header

Response Body

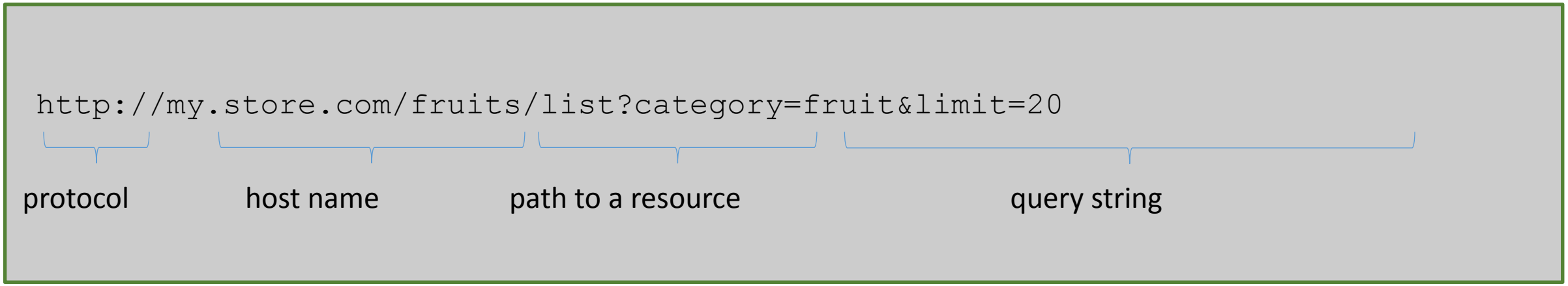
HTTP-REST Vocabulary

HTTP Methods supported by REST:

- GET – Requests a resource at the request URL
  - Should not contain a request body, as it will be discarded.
  - May be cached locally or on the server.
  - May produce a resource, but should not modify on it.
- POST – Submits information to the service for processing
  - Should typically return the new or modified resource.
- PUT – Add a new resource at the request URL
- DELETE – Removes the resource at the request URL
- OPTIONS – Indicates which methods are supported
- HEAD – Returns meta information about the request URL

HTTP-REST VOCABULARY

A typical HTTP REST URL:



- The **protocol** identifies the transport scheme that will be used to process and respond to the request.
- The **host name** identifies the server address of the resource.
- The **path** and **query string** can be used to identify and customize the accessed resource.

HTTP AND REST

A REST service framework provides a **controller** for routing HTTP requests to a request handler according to:

- The HTTP method used (e.g. GET, POST)
- Supplied path information (e.g /service/listItems)
- Query, form, and path parameters
- Headers, cookies, etc.

REST CHARACTERISTICS

- Resources: Application state and functionality are abstracted into resources.
  - ❑ *URI:* Every resource is **uniquely addressable** using URIs.
  - ❑ *Uniform Interface*: All resources share a uniform interface for the transfer of state between client and resource, consisting of
    - Methods: Use only HTTP methods such as **GET**, **PUT**, **POST**, **DELETE**, **HEAD**
    - Representation
- Protocol (The constraints and the principles)
  - ❑ *Client-Server*
  - ❑ *Stateless*
  - ❑ *Cacheable*
  - ❑ *Layered*

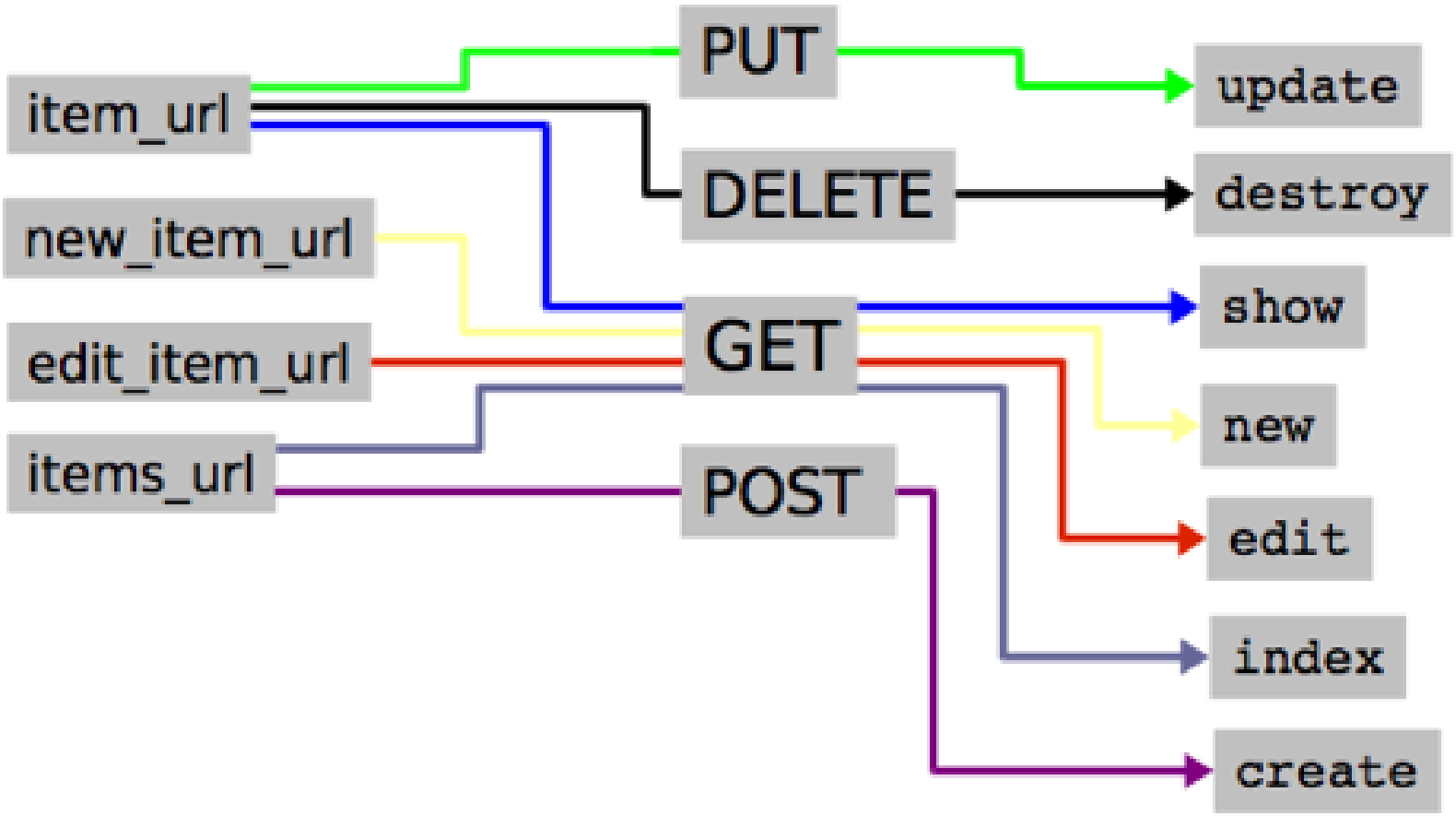
HTTP METHODS

- GET – *safe, idempotent, cacheable*
- PUT - *idempotent*
- POST
- DELETE - *idempotent*
- HEAD
- OPTIONS

CRUD OPERATIONS MAPPED TO HTTP METHODS IN RESTFUL WEB SERVICES

OPERATION	HTTP METHOD
Create	POST
Read	GET
Update	PUT or POST
Delete	DELETE

MAPPING


item_url
new_item_url
edit_item_url
items_url
PUT
DELETE
GET
POST
update
destroy
show
new
edit
index
create

EXAMPLE API MAPPING

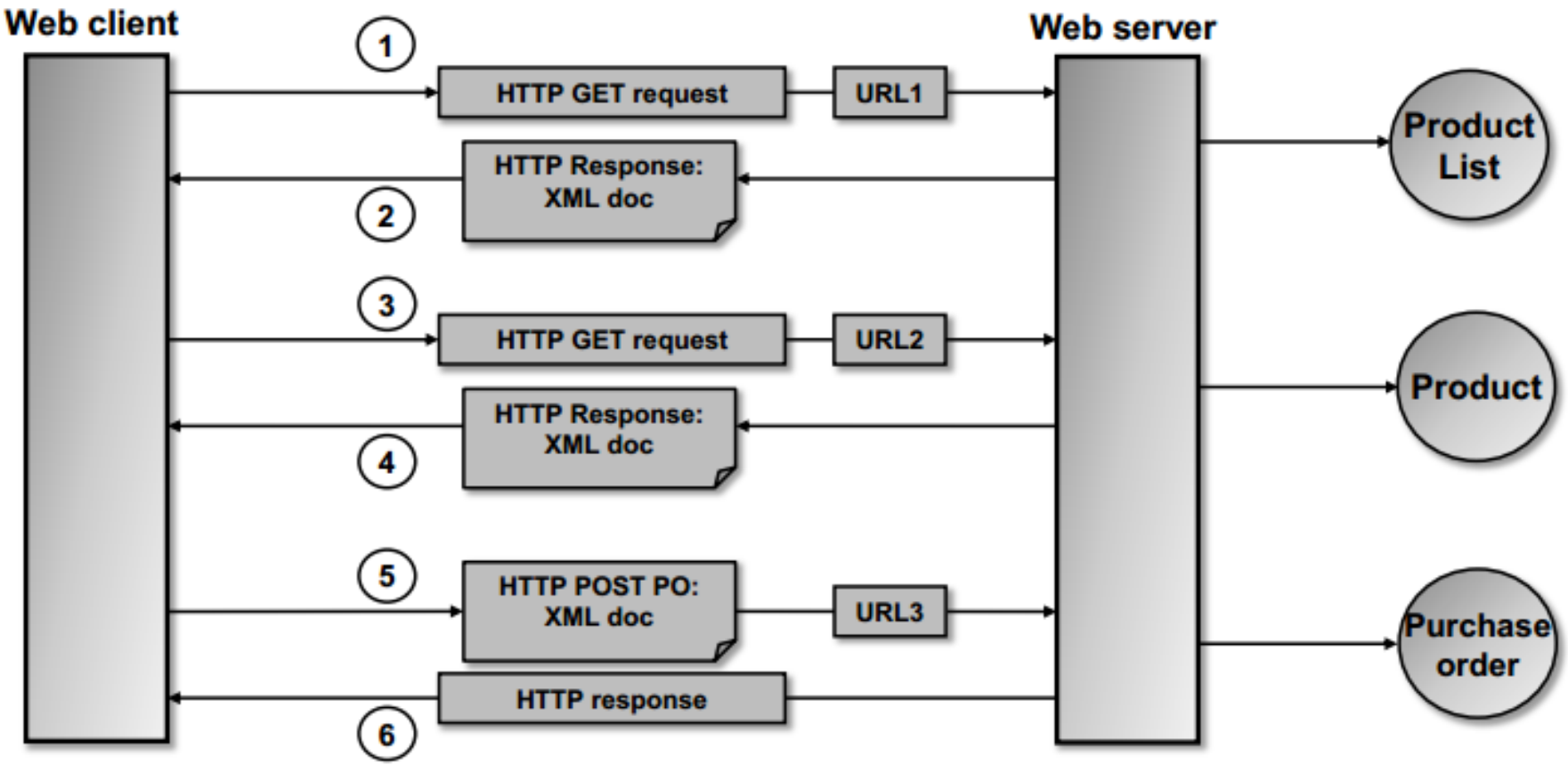
Route	HTTP Verb	Description
/api/user	`GET`	Get all the users.
/api/user	`POST`	Create a new user.
/api/user/{id}	`GET`	Get a single user.
/api/user/{id}	`PUT`	Update a user with new info.
/api/user/{id}	`DELETE`	Delete a user.

Status Codes

HTTP status codes returned in the response header:

- **200 OK** The resource was read, updated, or deleted.
- **201 Created** The resource was created.
- **400 Bad Request** The data sent in the request was bad.
- **403 Not Authorized** The Principal named in the request was not authorized to perform this action.
- **404 Not Found** The resource does not exist.
- **409 Conflict** A duplicate resource could not be created.
- **500 Internal Server Error** A service error occurred.

REST (REPRESENTATIONAL STATE TRANSFER)



REPRESENTATIONS

- Represent to JSON
- Accept: application/json

```
{
    "ID": "1",
    "Name": "M Vaqqas",
    "Email": "m.vaqqas@gmail.com",
    "Country": "India"
}
```

- Represent to XML
- Accept: text/xml

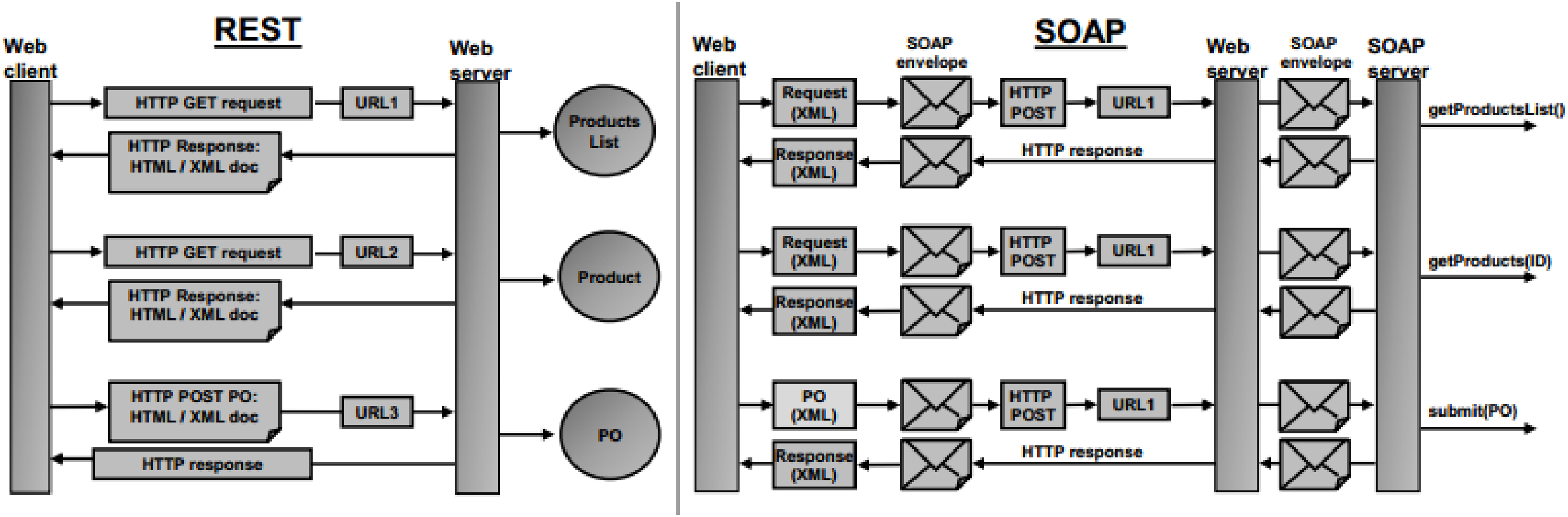
```
<Person>

<ID>1</ID>

<Name>M Vaqqas</Name>

<Email>m.vaqqas@gmail.com</Email>

<Country>India</Country>
</Person>
```

WHY JSON?

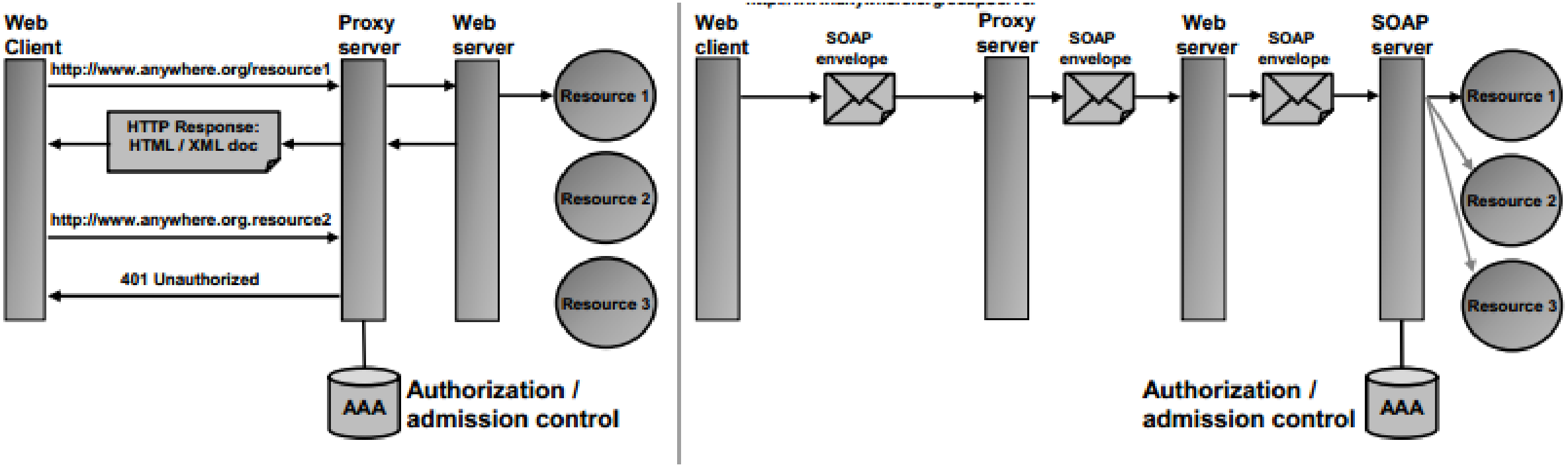
- Ubiquity
- Simplicity
- Readability
- Scalability
- Flexibility

- Twiter’s REST API uses only JASON

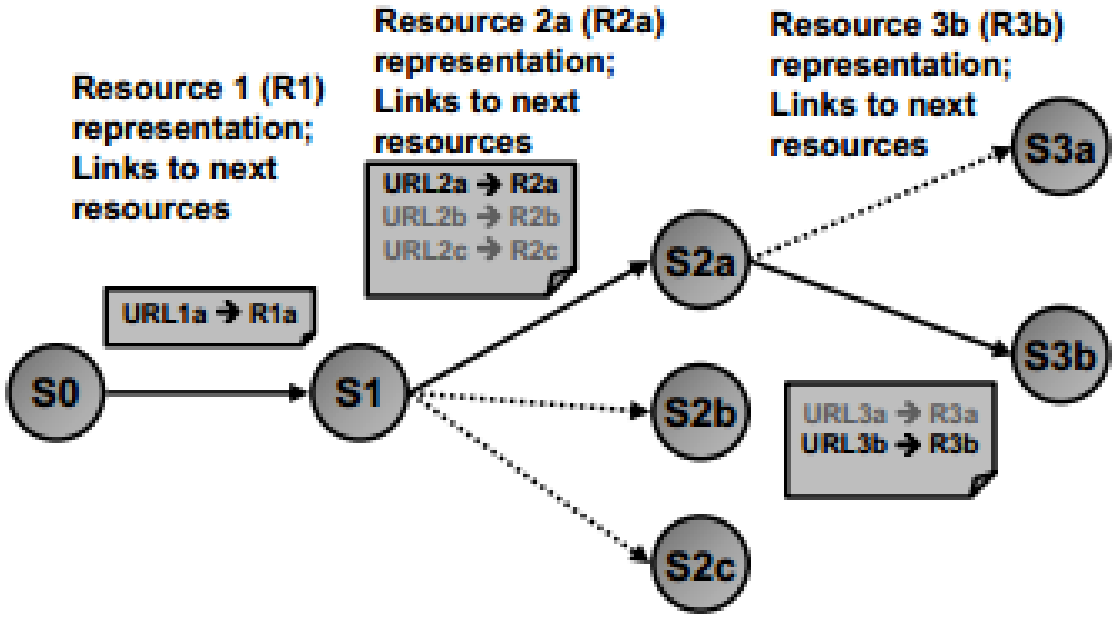
REST VS SOAP (CALL)

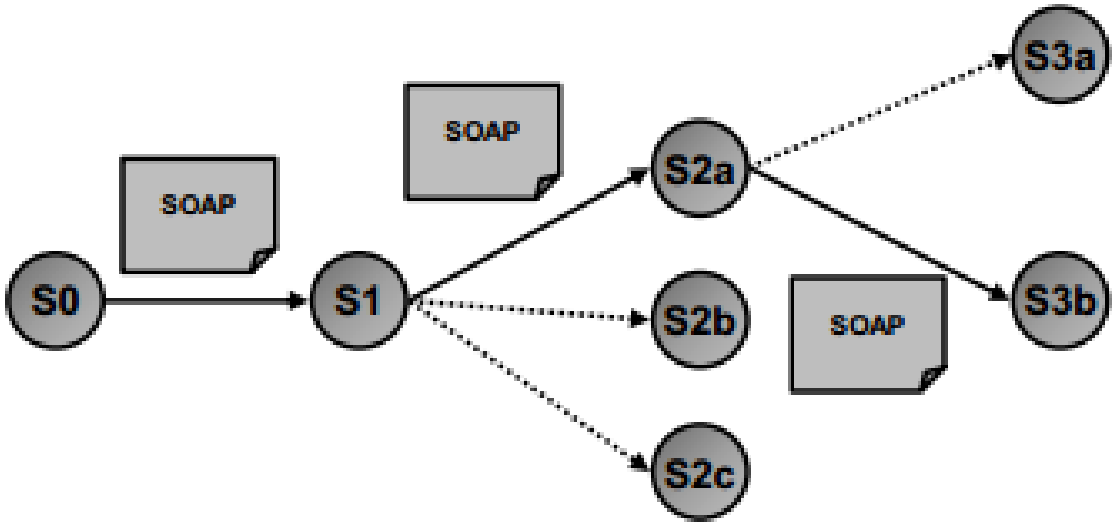


REST VS SOAP (CONTROL)

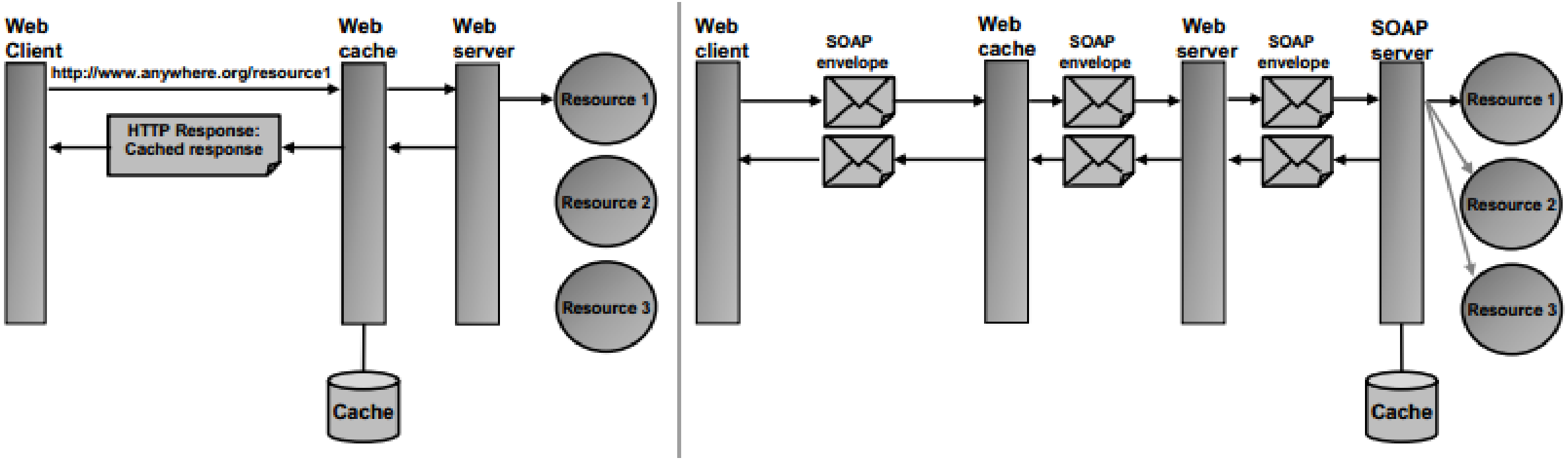


REST VS SOAP (LINK)

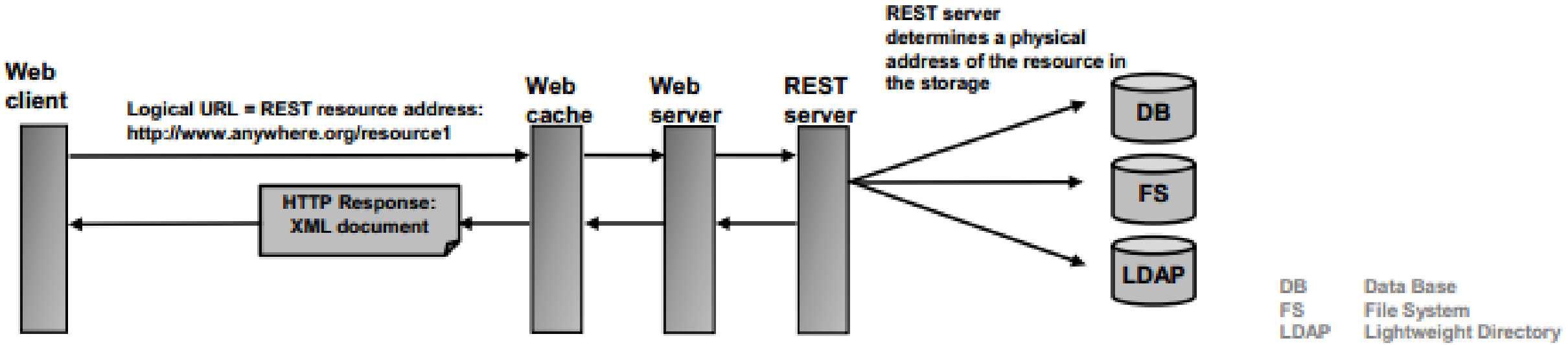




REST VS SOAP (CACHE)



REST VS SOAP (MANAGE)



RESOURCES

Nouns, not Verbs

Coarse Grained, not Fine Grained

Architectural style for use-case scalability

What if?

/getAccount

/createDirectory

/updateGroup

/verifyAccountEmailAddress

What if?

/getAccount
/getAllAccounts
/searchAccounts
/createDirectory
/createLdapDirectory
/updateGroup
/updateGroupName
/findGroupsByDirectory
/searchGroupsByName
/verifyAccountEmailAddress
/verifyAccountEmailAddressByToken
…
Smells like bad RPC. DON’T DO THIS.

LINKS BETWEEN RESOURCES

```
<Club>
        <Name>Authors Club</Name>
        <Persons>
                <Person>
                        <Name>M. Vaqqas</Name>
                        <URI>http://MyService/Persons/1</URI>
                </Person>
                <Person>
                        <Name>S. Allamaraju</Name>
                        <URI>http://MyService/Persons/12</URI>
                </Person>
        </Persons>
</Club>
```

Caching Control With HTTP Header

Header	Application
Date	Date and time when this representation was generated.
Last Modified	Date and time when the server last modified this representation.
Cache-Control	The HTTP 1.1 header used to control caching.
Expires	Expiration date and time for this representation. To support HTTP 1.0 clients.
Age	Duration passed in seconds since this was fetched from the server. Can be inserted by an intermediary component.

Directives in a Cache-Control

Directive	Application
Public	The default. Indicates any component can cache this representation.
Private	Intermediary components cannot cache this representation, only client or server can do so.
no-cache/no-store	Caching turned off.
max-age	Duration in seconds after the date-time marked in the Date header for which this representation is valid.
s-maxage	Similar to max-age but only meant for the intermediary caching.
must-revalidate	Indicates that the representation must be revalidated by the server if max-age has passed.
proxy-validate	Similar to max-validate but only meant for the intermediary caching.

What is API

API

- Application Programming Interface
- Source code interface
  - For library or OS
  - Provides services to a program
- At its base, like a header file
  - But, more complete

Designing an API

- Gather *requirements*
  - Don’t gather *solutions*
  - Extract true requirements
  - Collect specific scenarios where it will be used
- Create short specification
  - Consult with users to see whether it works
  - Flesh it out over time
- Hints:
  - Write plugins/use examples before fully designed and implemented
  - Expect it to evolve

Broad Issues to Consider in Design

- 1. Interface
  - The classes, methods, parameters, names
- 2. Resource Management
  - How is memory, other resources dealt with
- 3. Error Handling
  - What errors are caught and what is done
- Information Hiding
  - How much detail is exposed
  - Impacts all three of the above

INTERFACE PRINCIPLES

- Simple
- General
- Regular
- Predictable
- Robust
- Adaptable

Resource Management

- Determine which side is responsible for
  - Initialization
  - Maintaining state
  - Sharing and copying
  - Cleaning up
- Various resources
  - Memory
  - Files
  - Global variables

Error Handling

- Catch errors, don't ignore them
- "Print message and fail" is not always good
  - Especially in APIs
  - Need to allow programs to recover or save data
- Detect at low level, but handle at high level
  - Generally, error should be handled by *calling* routine
  - The callee can leave things in a "nice" state for recovery, though
    - Keep things usable in case the caller can recover

API DESIGN IS HARD

- What is functionality to expose
- How to expose it
- How best to expose it
- How to adjust and improve

URL Design

Plural nouns for collections	`/dogs`
ID for entity	`/dogs/1234`
Associations	`/owners/5678/dogs`
4 HTTP Methods	`POST GET PUT DELETE`
Bias toward concrete names	`/dogs (not animals)`
Multiple formats in URL	`/dogs.json` `/dogs.xml`
Paginate with limit and offset	`?limit=10&offset=0`
Query params	`?color=red&state=running`
Partial selection	`?fields=name,state`
Use medial capitalization	`"createdAt": 1320296464` `myObject.createdAt;`
Use verbs for non-resource requests	`/convert?from=EUR&to=CNY&amount=100`
Search	`/search?q=happy%2Blabrador`
DNS	`api.foo.com` `developers.foo.com`

Versioning

Include version in URL	`/v1/dogs`
Keep one previous version long enough for developers to migrate	`/v1/dogs` `/v2/dogs`

Errors

8 Status Codes	`200 201 304 400 401 403 404 500`
Verbose messages	`{"msg": "verbose, plain language hints"}`

Client Considerations

Client does not support HTTP status codes	`?suppress_response_codes=true`

Client does not support HTTP methods	`GET /dogs?method=post`
	`GET /dogs`
	`GET /dogs?method=put`
	`GET /dogs?method=delete`

Complement API with SDK and code libraries	1. JavaScript 2. … 3. …

RESOURCE NAMING

Follow KISS principle

Use nouns for resource names

Use verbs actions

RESOURCE NAMING

2 base urls:

- Collection (plural)

  /employees

- Resource

  /employees/itramin

RESOURCE NAMING

Use nouns for non-resource actions:

- convert
- calculate

REPRESENTATION

- Request header: Content-type
  - application/json
  - application/xml
- Extension
  - /employees.json
  - /employees.xml
  - /employees/itramin.json
  - /employees/itramin.xml
- Query param
  - /employees/type=json
  - /employees/type=xml

HTTP METHODS

- **C**reate
- **R**ead
- **U**pdate
- **D**elete

- POST
- GET
- PUT
- DELETE

HTTP METHODS

Resource	POST Create	GET Read	PUT Update	DELETE Delete
/employees	Create new emp	Return list of employees	Bulk update or Error	Delete all employees Error
/employees/itramin	Not used Error	Return employee **itramin**	Update employee **itramin**	Delete emp **itramin**

ERROR HANDLING

- Error handling is very important for reliable and stable API.
- Use appropriate HTTP status codes
- Document success and error cases

HTTP STATUS CODES

Group	Comment
1xx	Informational(100,101)
2xx	Success(200,201)
3xx	Redirect (301, 304)
4xx	Client error(401, 404, 405)
5xx	Server error(500, 503)

http://en.wikipedia.org/wiki/List_of_HTTP_status_codes

Error Messages

Consider the following:

- For user
- For developer
- Unique error code
- Link to documentation

ERROR MESSAGES

{
userMessage:”Show this error to user”,
devMessage:”Detailed error for developer”,
“errorCode”: 12345,
“doc”:”http://api.azercell.com/azimus/docs/errors/12345”
}

Version management

- Consider versioning during design, no later
- Consider support for multiple client

- 3 general practice:
  - Part of url
  - Request param
  - Request header

Version management

- Part of url
  - api.azercell.com/azimus/**v1**/employees
  - api.azercell.com/azimus/**20130501**/employees
- Request header
  - Content-Type: application/json;v=1
- Query param – overrides header
  - ?v=1

PAGING

- It is bad idea to return all resources in database, so use paging

- Query parameter
- Request/response headers

PAGING

Query parameter:

- Facebook - limit, offset
- Twitter – page, rpp(records per page)
- LinkedIn – start, count
- Consider reasonable default values such as limit=20, offset=0

PAGING

- Request header

  Range: items=0-19

- Response header

  Content-Range: items 0-19/152

  Content-Range: items 0-19/*

SEARCH & FILTER & SORT

- Search
- /employees?q=ramin – (default json)
- /employees.xml?q=ramin

- Filter
- /employees?filter="name::ramin|department=ICT"
- /employees?name=ramin&department=ICT

- Sort

/employees?sort=name|surname

/employees?sort=-salary|name

SECURITY

- Authentication
  - HTTP Basic authentication + SSL
  - API token,key
  - Custom authentication mechanism

- Authorization
  - Oauth 1.0
  - Oauth 2.0
  - Custom mechanism
    - Amazon AWS Identity and Access Management
    - PayPal Permissions Service
    - Your own homegrown api

SECURITY

- HTTP Basic authentication + SSL

```
GET /employees/ HTTP/1.1
Host: www.example.org
Authorization: Basic cGhvdG9hcHAuMDAxOmJhc2ljYXV0aA==
```

CACHE & SCALABILITY

- Carefully implement caching

HTTP/1.1 200 OK

Date: Sat, 06 Jul 2013 09:56:14 GMT

Last-Modified: Sat, 06 Jul 2013 09:56:14 GMT

Expires: Sun, 06 Jul 2013 10:56:14 GMT

Cache-Control: max-age=3600,must-revalidate

Content-Type: application/xml; charset=UTF-8

REST API WITH PHP MICROFRAMEWORK

- BulletPHP
- Fat-Free Framework
- Limonade
- Lumen
- Phalcon
- Recess PHP

Silex

Slim

Tonic

The One Framework

Wave Framework

Zaphpa

http://www.gajotres.net/best-available-php-restful-micro-frameworks/

REST API WITH JAVA MICROFRAMEWORK

- Dropwizard
- Jersey
- Ninja Web Framework
- Play Framework
- RestExpress
- Restlet
- Restx
- Spark Framework

DATABASE

- students
- =========
- std_id int pk
- std_name varchar(50)

quiz

=====

quiz_id int auto pk

quiz_name varchar(50)

scores

======

scr_id int auto pk

std_id int fk

quiz_id int fk

scr_score decimal(10,2)

What is Node.js

- As an asynchronous event driven framework, Node.js is designed to build scalable network applications.

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 1337;

http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World\n');
}).listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

HOW TO INSTALL AND USING NODE.JS

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

LTS
Mature and Dependable

Stable
Latest Features

Windows Installer
node-v4.2.6-x86.msi

Macintosh Installer
node-v4.2.6.pkg

Source Code
node-v4.2.6.tar.gz

Windows Installer (.msi)	32-bit	64-bit	
Windows Binary (.exe)	32-bit	64-bit	
Mac OS X Installer (.pkg)	64-bit		
Mac OS X Binaries (.tar.gz)	64-bit		
Linux Binaries (.tar.xz)	32-bit	64-bit	
SunOS Binaries (.tar.xz)	32-bit	64-bit	
ARM Binaries (.tar.xz)	ARMv6	ARMv7	ARMv8
Docker Image	Official Node.js Docker Image		
Source Code	node-v4.2.6.tar.gz		

How to install and using node.js

Create project directory and create package.json

```
{
  "name": "RESTful-API",
  "version": "0.0.1",
  "scripts": {
    "start": "node Server.js"
  },
  "dependencies": {
    "express": "~4.12.2",
    "mysql": "~2.5.5",
    "body-parser": "~1.12.0",
    "MD5": "~1.2.1"
  }
}
```

Run npm install

HOW TO INSTALL AND USING NODE.JS

```
// server.js
 // BASE SETUP
// ======================================================================
// call the packages we need
var express = require('express'); // call express
var app = express(); // define our app using express
var bodyParser = require('body-parser'); // configure app to use bodyParser()
// this will let us get the data from a POST
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
var port = process.env.PORT || 8080; // set our port
// ROUTES FOR OUR API
// ======================================================================
var router = express.Router(); // get an instance of the express Router
// test route to make sure everything is working (accessed at GET http://localhost:8080/api)
router.get('/', function(req, res) {
            res.json({ message: 'hooray! welcome to our api!' });
});
// more routes for our API will happen here
// REGISTER OUR ROUTES -------------------------------
// all of our routes will be prefixed with /api
app.use('/api', router);
// START THE SERVER
// ======================================================================
app.listen(port); console.log('Magic happens on port ' + port);
```

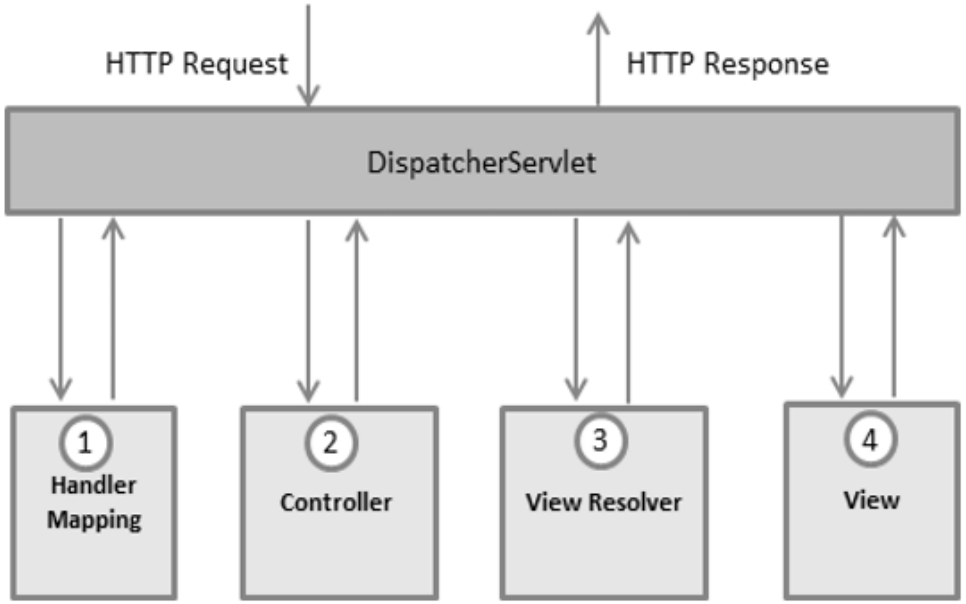
HOW TO INSTALL AND USING NODE.JS

```
// server.js
var express    = require('express');
var app        = express();
var bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
var port = process.env.PORT || 8080;
var router = express.Router();
router.get('/', function(req, res) {
    res.json({ message: 'hooray! welcome to our api!' });
});
app.use('/api', router);
app.listen(port);
console.log('Magic happens on port ' + port);
```

3) Spring MVC Example & Practice...

THE DISPATCHER SERVLET

The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that handles all the HTTP requests and responses.



1) After receiving an HTTP request, DispatcherServlet consults the HandlerMapping to call the appropriate Controller.

2) The Controller takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the DispatcherServlet.

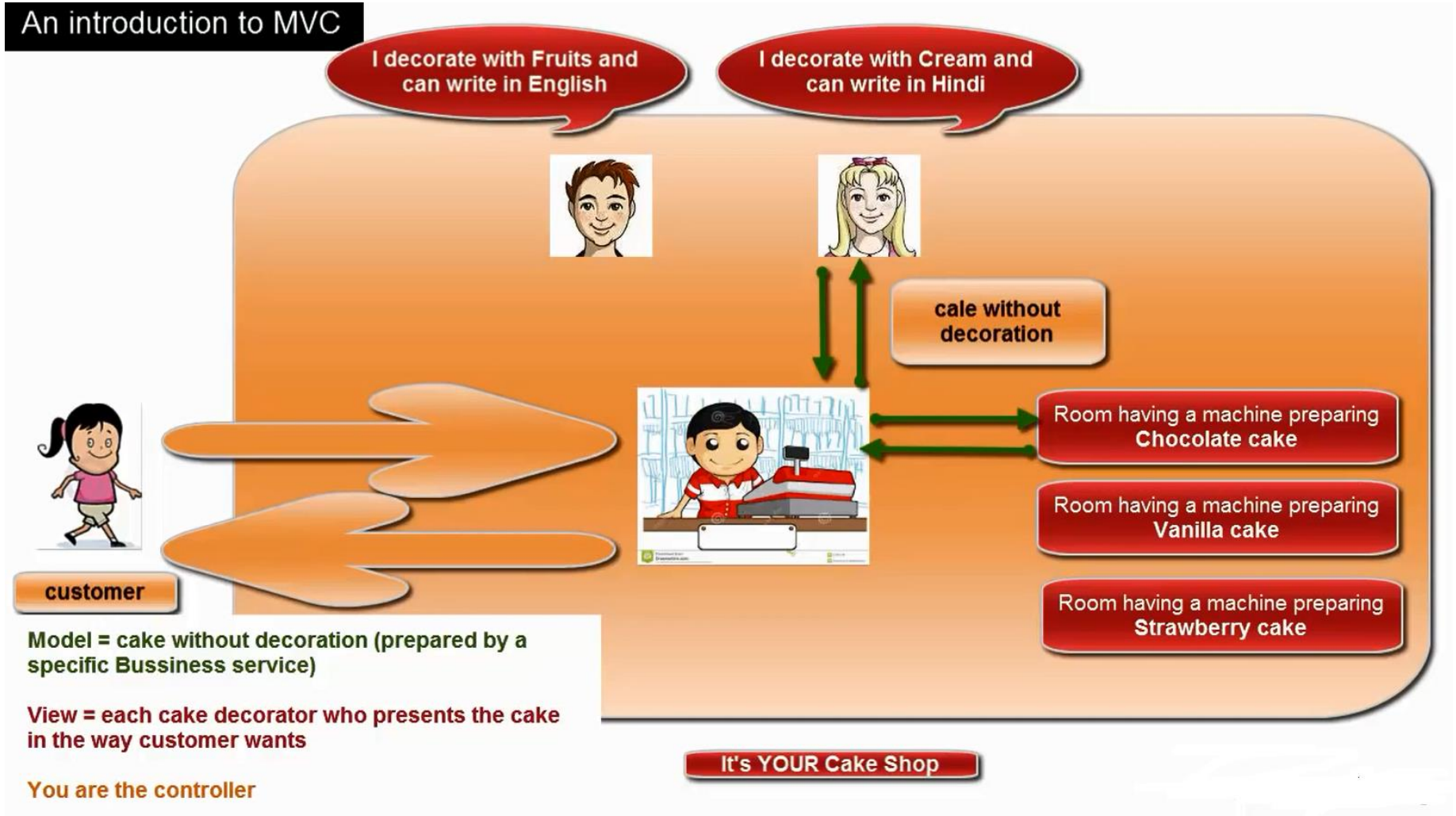
3) The DispatcherServlet will take help from ViewResolver to pickup the defined view for the request.

4) Once view is finalized, The DispatcherServlet passes the model data to the view which is finally rendered on the browser.

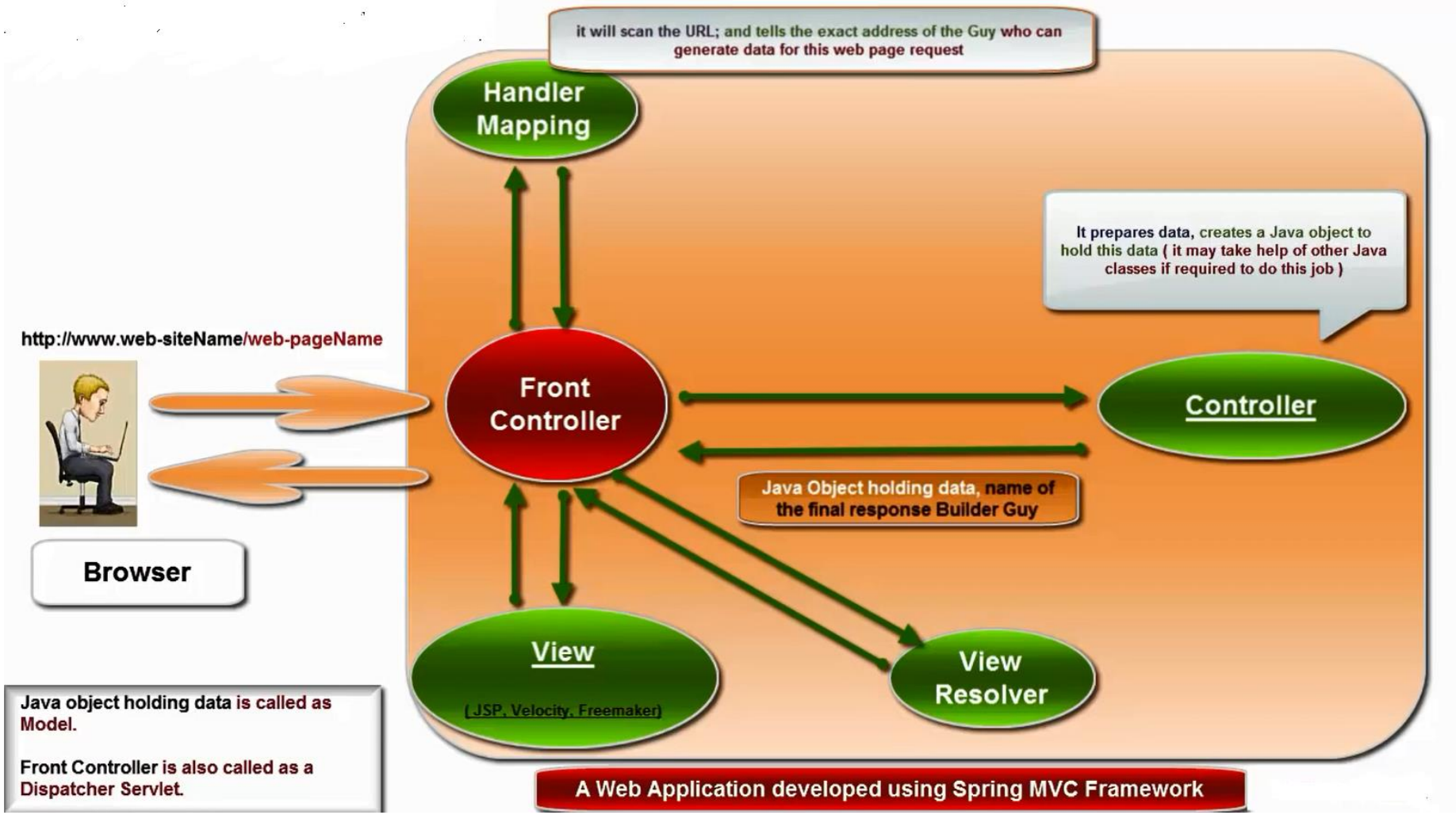
SPRING MVC FORM HANDLING EXAMPLE

- Steps
- 1 Create a Dynamic Web Project with a name HelloWeb and create a package under the src folder in the created project.
- 2 Drag and drop below mentioned Spring and other libraries into the folder WebContent/WEB-INF/lib.
- 3 Create a Java classes Student and StudentController under the com.tutorialspoint package.
- 4 Create Spring configuration files Web.xml and HelloWeb-servlet.xml under the WebContent/WEB-INF folder.
- 5 Create a sub-folder with a name jsp under the WebContent/WEB-INF folder. Create a view files student.jsp and result.jsp under this sub-folder.
- 6 The final step is to create the content of all the source and configuration files and export the application

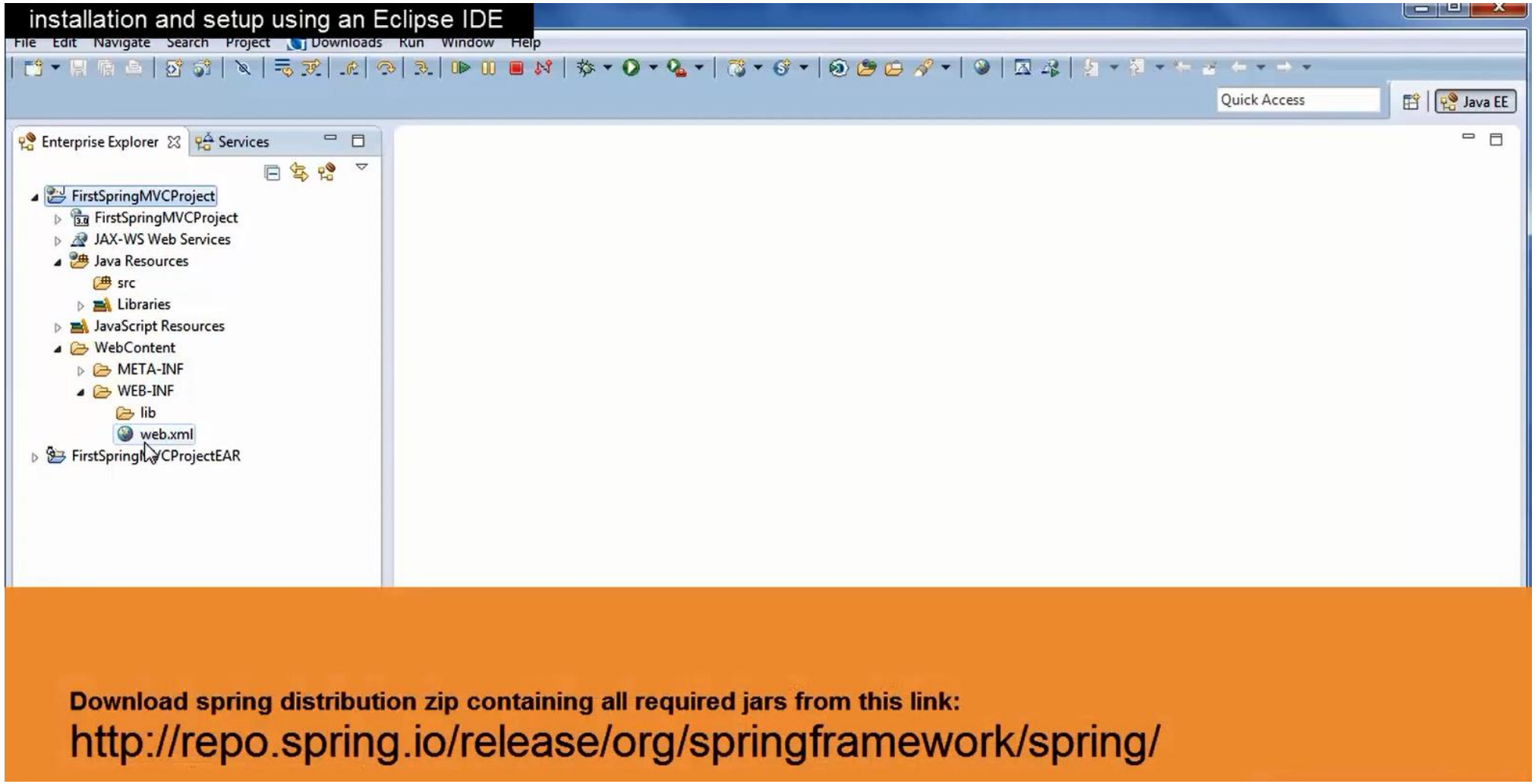
MVC SCENARIO (REAL LIFE EXAMPLE)



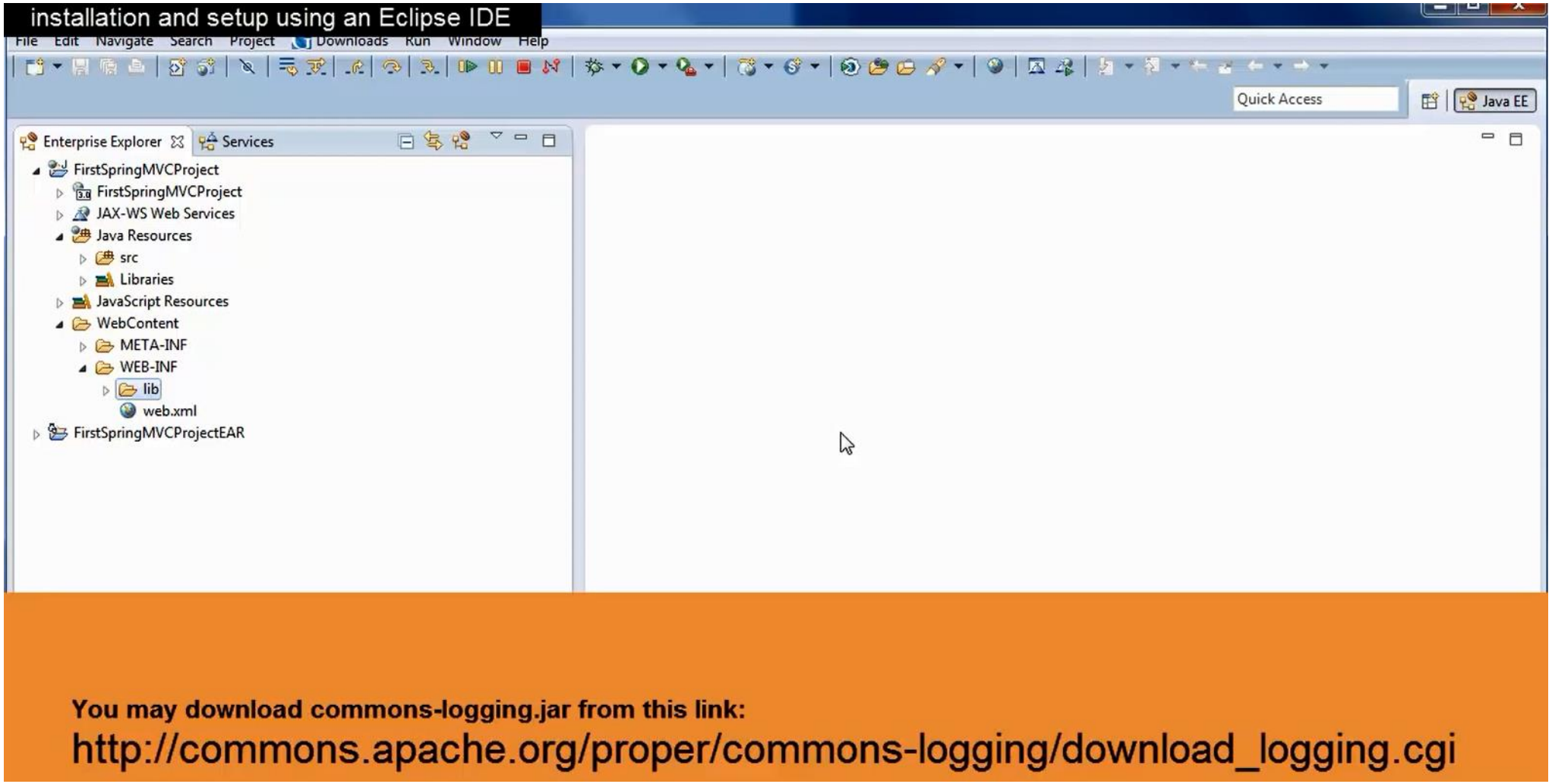
SPRING MVC SCENARIO



MVC SET UP IN ECLIPSE IDE...



MVC SET UP IN ECLIPSE IDE...



4-STEPS APPROACH TO CREATE SPRING MVC

1. Modify web.xml file
2. Create spring-dispatcher-servlet.xml
3. create the HelloController.Java classs (A controller)
4. Create the HelloPage.jsp file (A View)

4) Connecting to Oracle Database

EDITING LISTENER.ORA

- C:\oraclexe\app\oracle\product\11.2.0\server\network\ADMIN

```
SID_LIST_LISTENER =
 (SID_LIST =
  (SID_DESC =
   (SID_NAME = PLSExtProc)
   (ORACLE_HOME = C:\oraclexe\app\oracle\product\11.2.0\server)
   (PROGRAM = extproc)
  )
  (SID_DESC =
   (SID_NAME = CLRExtProc)
   (ORACLE_HOME = C:\oraclexe\app\oracle\product\11.2.0\server)
   (PROGRAM = extproc)
  )
 )
LISTENER =
 (DESCRIPTION_LIST =
  (DESCRIPTION =
   (ADDRESS = (PROTOCOL = IPC)(KEY = EXTPROC1))
   (ADDRESS = (PROTOCOL = TCP)(HOST = kishore-PC)(PORT =
1521))
  )
 )

DEFAULT_SERVICE_LISTENER = (XE)
```

EDITING TNSNAMES.ORA

- C:\oraclexe\app\oracle\product\11.2.0\server\network\ADMIN

```
XE =
 (DESCRIPTION =
  (ADDRESS = (PROTOCOL = TCP)(HOST = kishore-PC)(PORT = 1521))
  (CONNECT_DATA =
   (SERVER = DEDICATED)
   (SERVICE_NAME = XE)
  )
 )

EXTPROC_CONNECTION_DATA =
 (DESCRIPTION =
  (ADDRESS_LIST =
   (ADDRESS = (PROTOCOL = IPC)(KEY = EXTPROC1))
  )
  (CONNECT_DATA =
   (SID = PLSExtProc)
   (PRESENTATION = RO)
  )
 )
```

```
LISTNER_ORCL =
 (ADDRESS = (PROTOCOL = TCP)(HOST = localhost)(PORT = 1522))


ORACLR_CONNECTION_DATA =
 (DESCRIPTION =
  (ADDRESS_LIST =
   (ADDRESS = (PROTOCOL = IPC)(KEY = EXTPROC1522))
  )
  (CONNECT_DATA =
   (SID = CLRExtProc)
   (PRESENTATION = RO)
  )
 )

ORCL =
 (DESCRIPTION =
  (ADDRESS = (PROTOCOL = TCP)(HOST = localhost)(PORT = 1522))
  (CONNECT_DATA =
   (SERVER = DEDICATED)
   (SERVICE_NAME = orcl)
  )
 )
```

MORE TO COME...

- Spring jar source url
- http://repo.spring.io/release/org/springframework/spring/4.3.9.RELEASE/

EXERCISE

- Use Restful web service approach publish any two of your weekly (week 3) homework as web service.