PA165 Enterprise Java
2014-2015

Intro to Service Oriented Architecture (SOA)

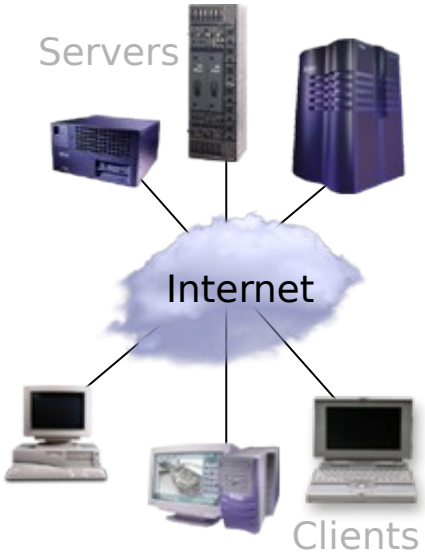
Bruno Rossi & Juha Rikkilä

Objectives and content of this part

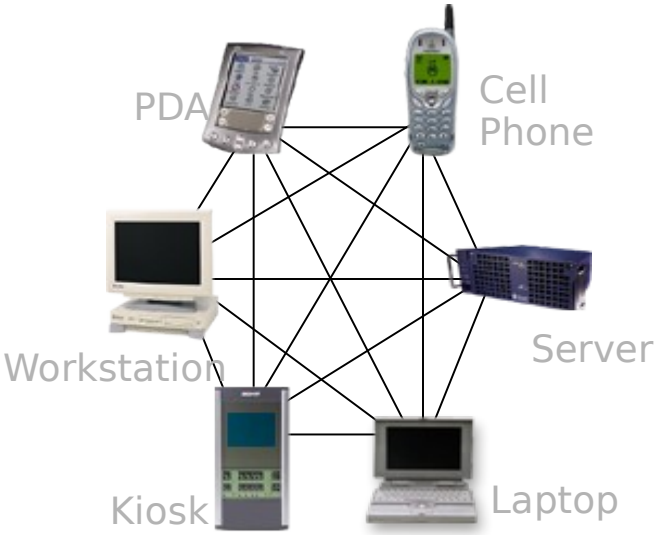
Objectives	Content
Get “the big picture” of SOA and related concepts	▪ Clients and servers ▪ SOA, why and why not ▪ Application development view ▪ Technology stack view ▪ Basic set of concepts

Distributed Computing Evolution

Client-Server(C/S) silos

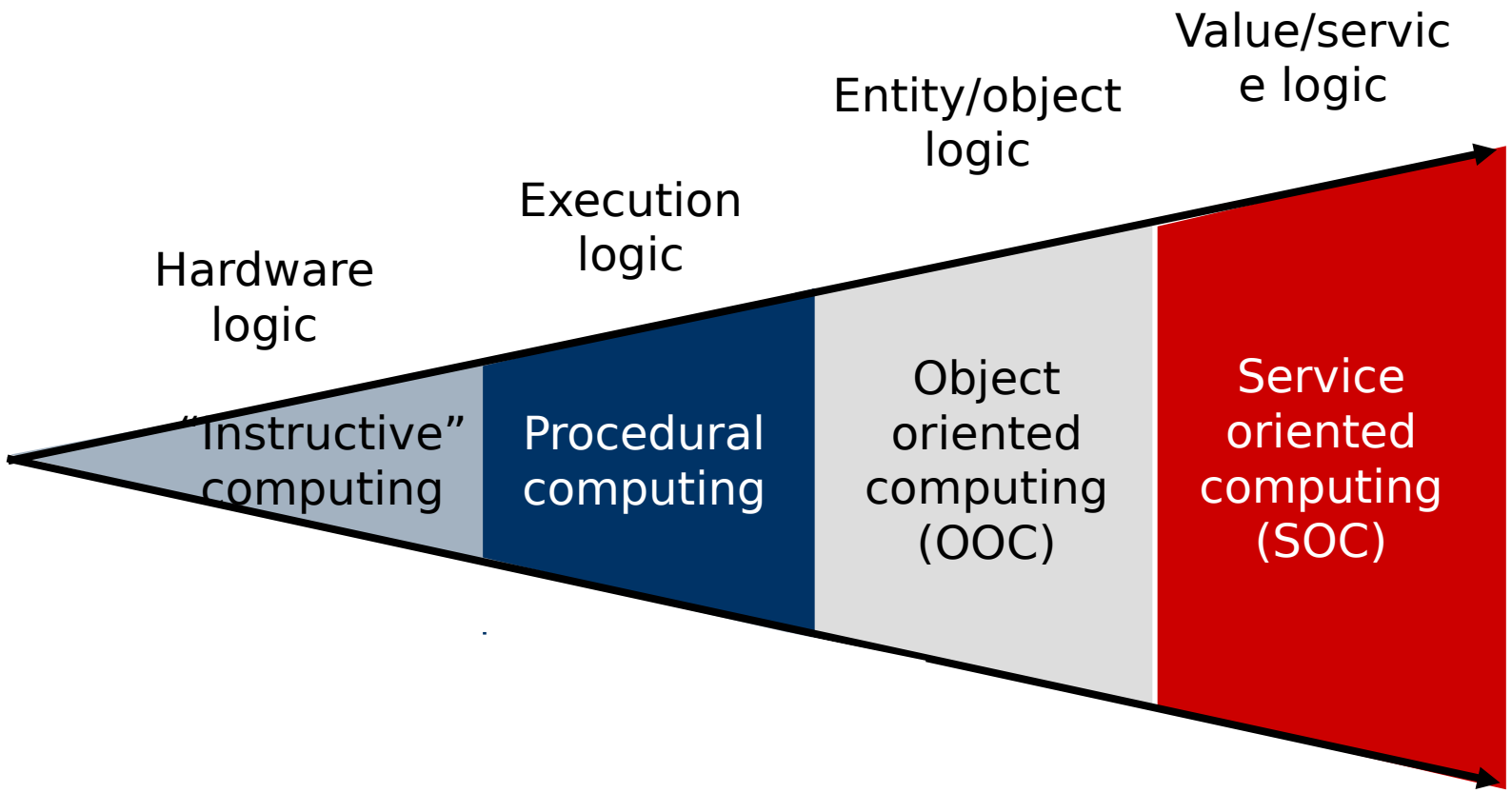


Web-based computing

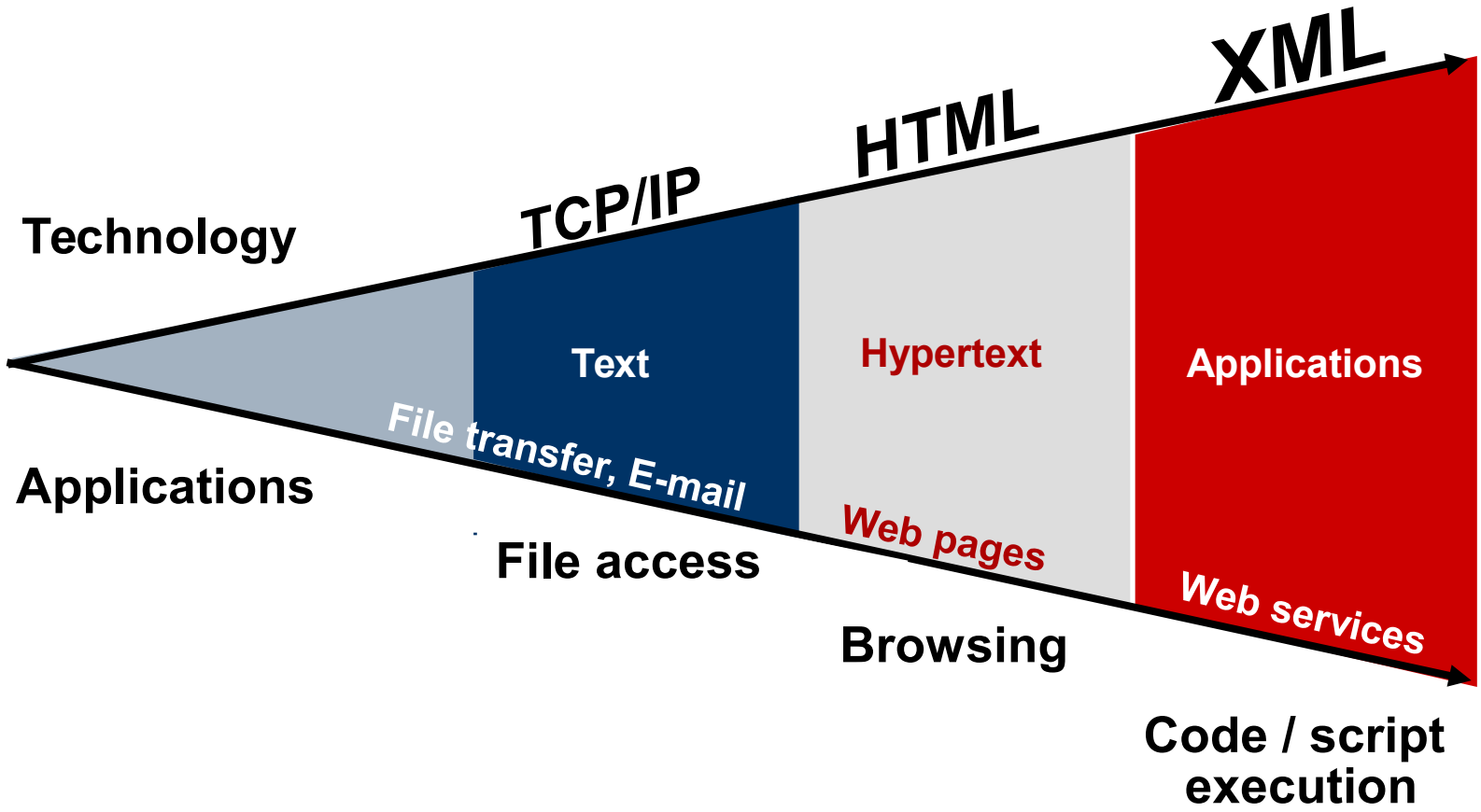


Web Services/Peer-to-Peer

Evolution of software development /programming



Internet evolution



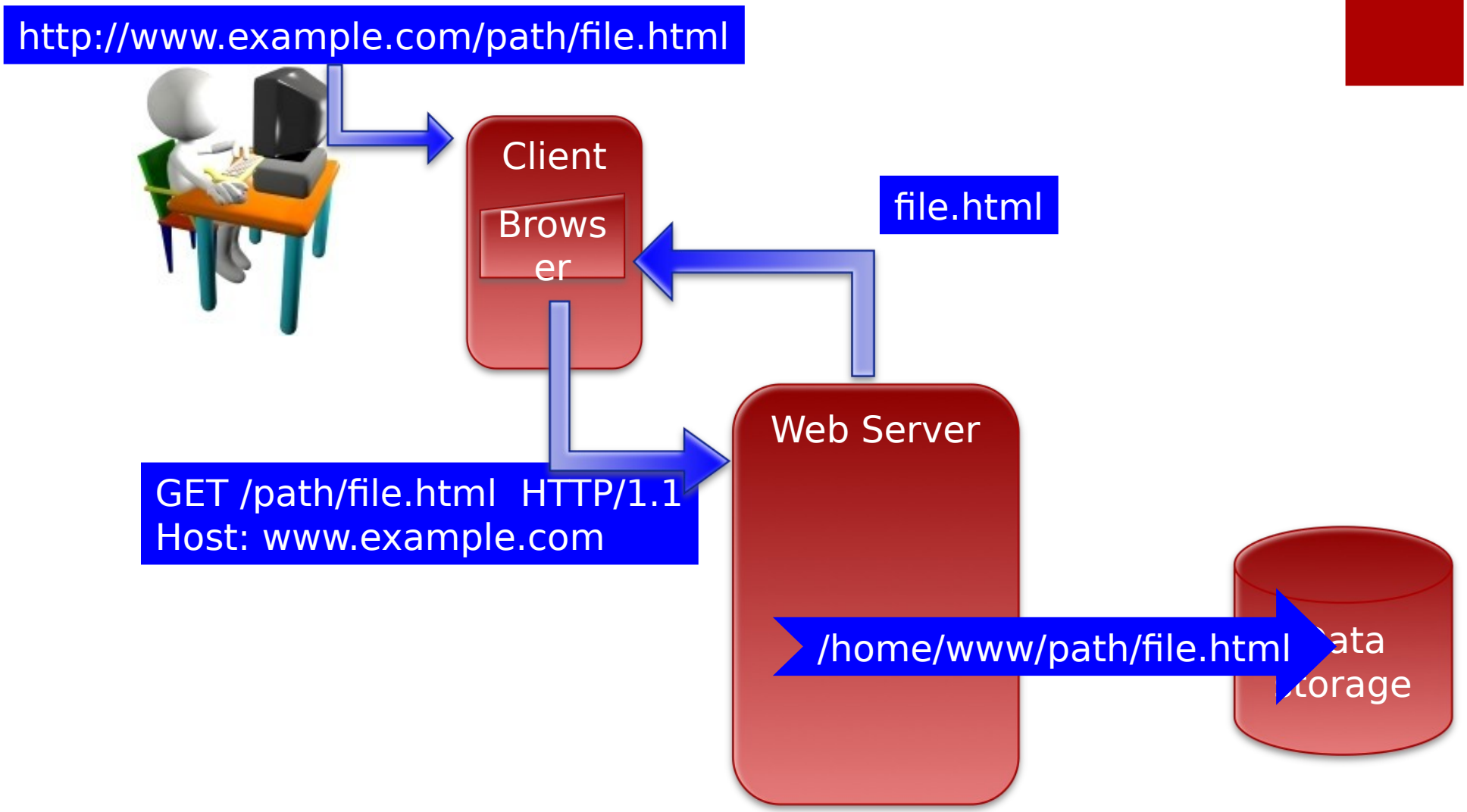
Service Oriented Computing (SOC)

- **SOC is an emerging cross-disciplinary paradigm for distributed computing** that is changing the way software applications are designed, architected, delivered and consumed

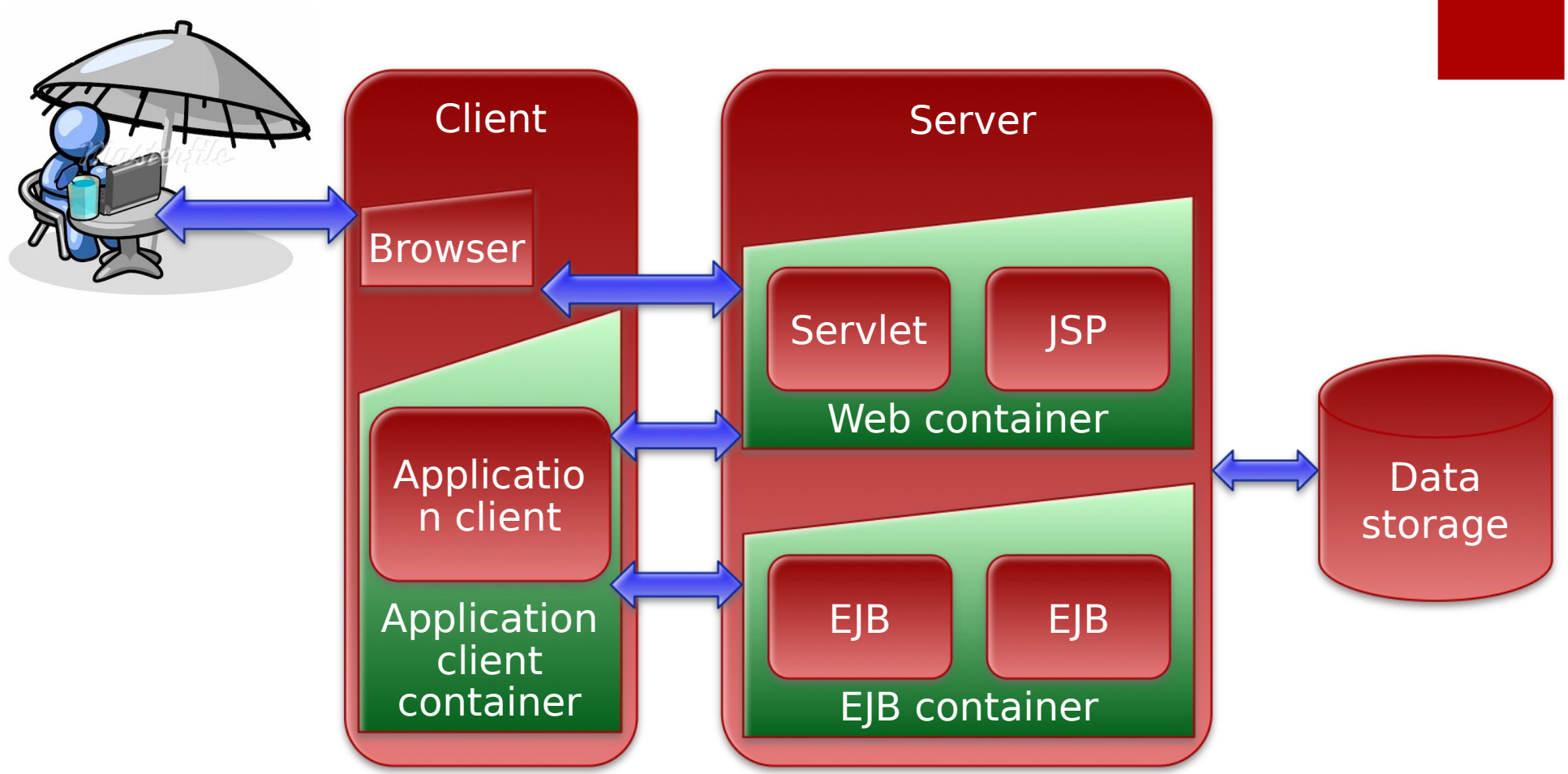
- SOC is a new computing paradigm that utilizes **services** as the **basic constructs** to support the development of rapid, low-cost and easy composition of distributed applications even in heterogeneous environments

S. Dustdar and B. J. Krämer, Eds., “Introduction to Special Issue on Service Oriented Computing (SOC),” ACM Trans. Web, vol. 2, no. 2, pp. 10:1–10:2, May 2008.

Browsing

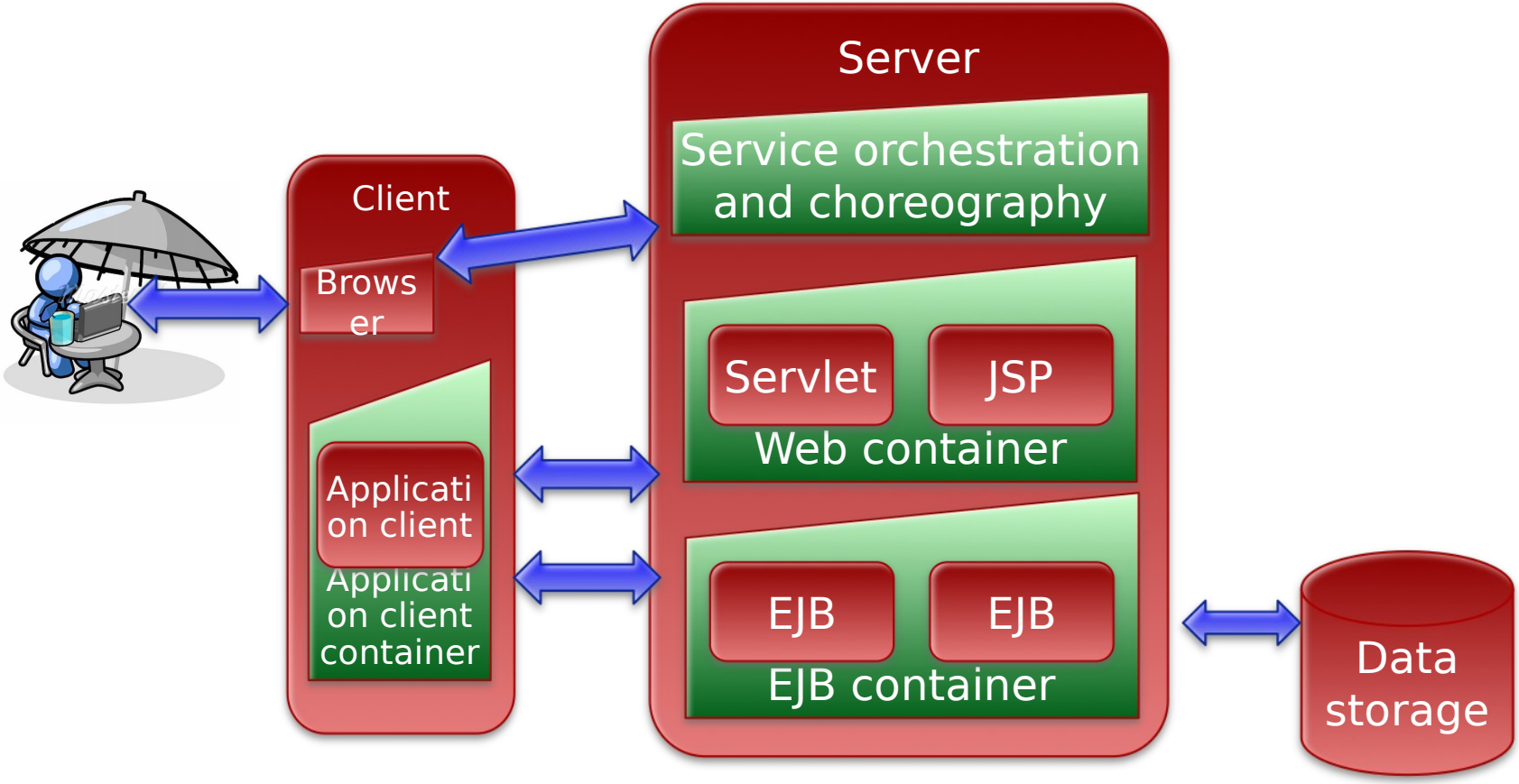

http://www.example.com/path/file.html
Client
Brows
er
file.html
Web Server
GET /path/file.html HTTP/1.1
Host: www.example.com
/home/www/path/file.html
ata
torage

Code / script / application execution

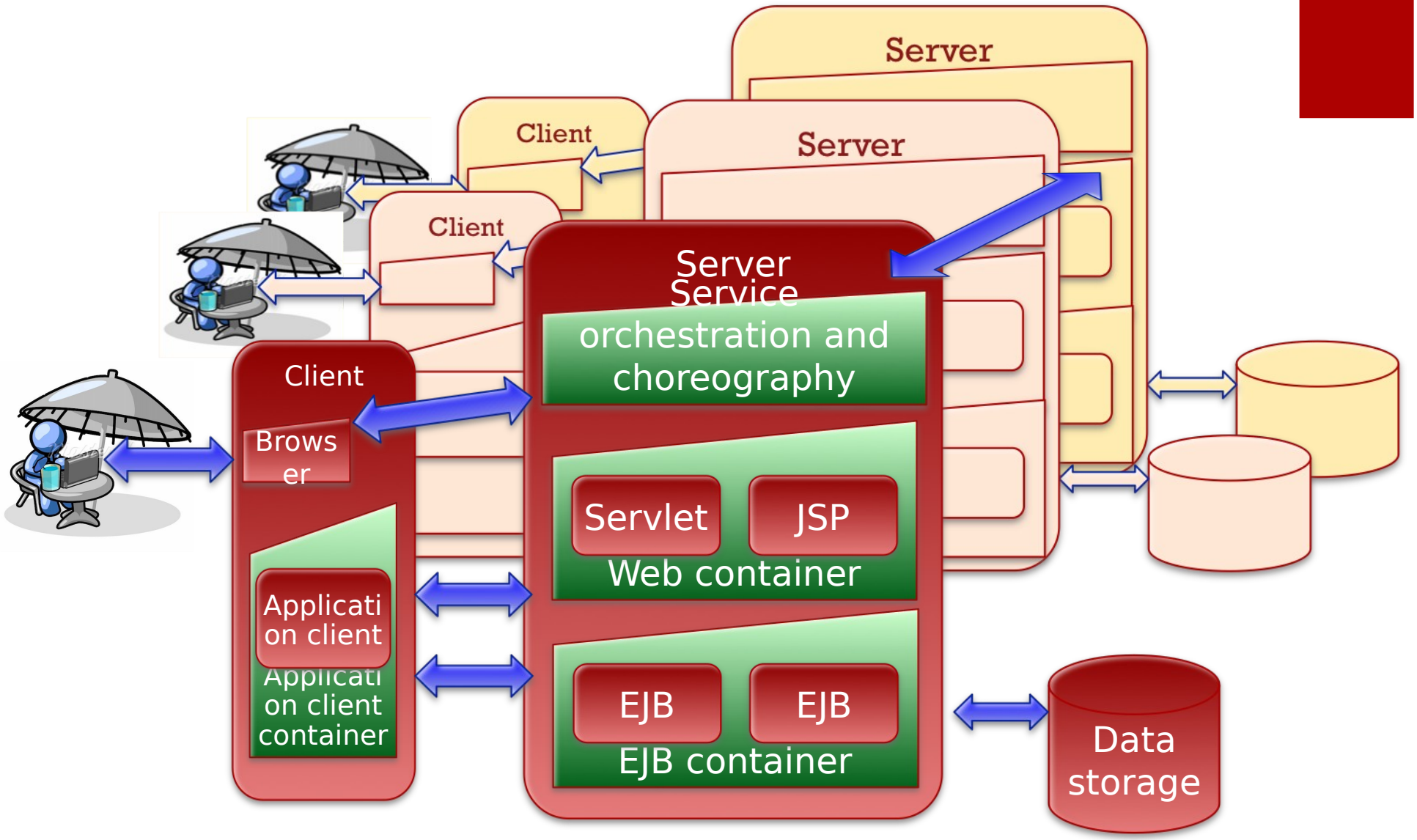


JSP = JavaServer Pages
EJB = Enterprise Java Beans

Service execution (1/2)



Service execution (2/2)



Some SOA definitions (1/2)

A **Service-Oriented Architecture (SOA)** facilitates the creation of flexible, re-usable assets for enabling end-to-end business solutions. *(Open Group Standard: SOA Reference Architecture, 2011)*

Contemporary **SOA** represents an open, agile extensible, federated, composable **architecture** comprised of autonomous, QoS-capable, vendor diverse, interoperable, discoverable, and potentially reusable services, implemented as Web services. *(Erl, T., Service-oriented Architecture: Concepts, Technology and Design, 2005)*

Service-Oriented Architecture is an **IT strategy** that organizes the discrete functions contained in enterprise applications into interoperable, standards-based services that can be combined and reused quickly to meet business needs. (*BEA white paper, 2005 -> 2008 Oracle)*

SOA is a **conceptual business architecture** where business functionality, or application logic, is made available to SOA users, or consumers, as shared, reusable services on an IT network. "Services" in an SOA are modules of business or application functionality with exposed interfaces, and are invoked by messages. *(Marks, E.A., Bell, M., Service Oriented Architecture (SOA): A Planning and Implementation Guide for Business and Technology, 2006)*

Some SOA definitions

Service-oriented architecture (SOA) is a set of **principles and methodologies for designing and developing software** in the form of interoperable services. These services are well-defined business functionalities that are built as software components (discrete pieces of code and/or data structures) that can be reused for different purposes. SOA design principles are used during the phases of systems development and integration. *(Wikipedia)*

SOA is an **architectural style** whose goal is to achieve loose coupling among interacting software agents. A service is a unit of work done by a service provider to achieve desired end results for a service consumer. Both provider and consumer are roles played by software agents on behalf of their owners. *(O'Reilly XML.COM)*

There is no unique definition: some refer to SOA as an architectural style, others as a paradigm, principles and methodologies, IT strategy, etc...

What is SOA

SOA is an **architectural style**,

realized as a collection of **collaborating agents**, each **called a service**,

whose goal is to **manage complexity** and **achieve architectural resilience and robustness** through ideas such as **loose coupling**, **location transparency**, and **protocol independence**.

(IBM definition of SOA)

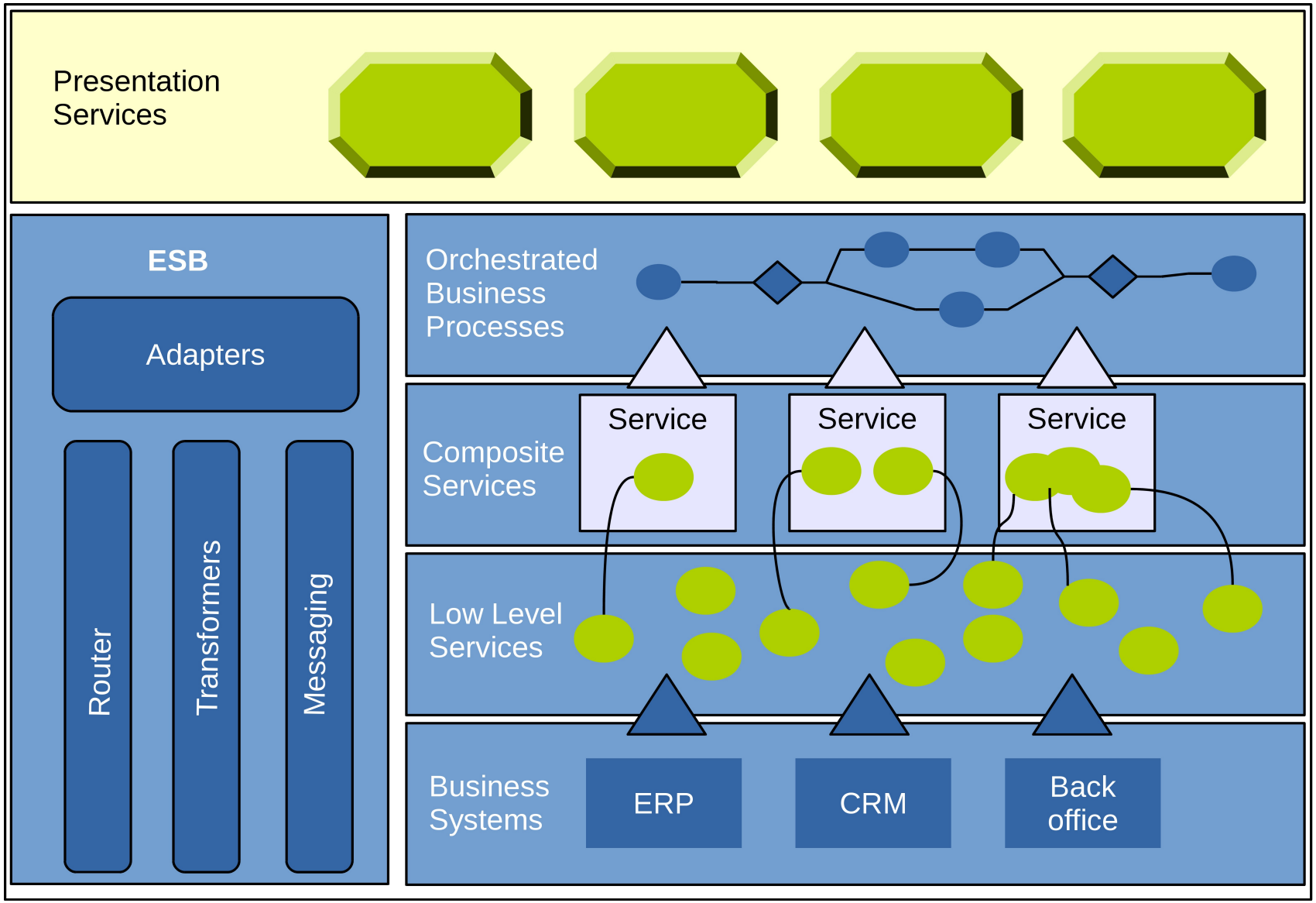
Service

- A **service** is an entity that has a description, and that is made available for use through a published interface that allows it to be invoked by a **service consumer**.
- A **service** in **SOA** is an exposed piece of functionality with three properties:
  - The **interface contract** to the service is platform-independent.
  - The **service** can be dynamically located and invoked.
  - The **service is self-contained**. That is, the service maintains its own state.

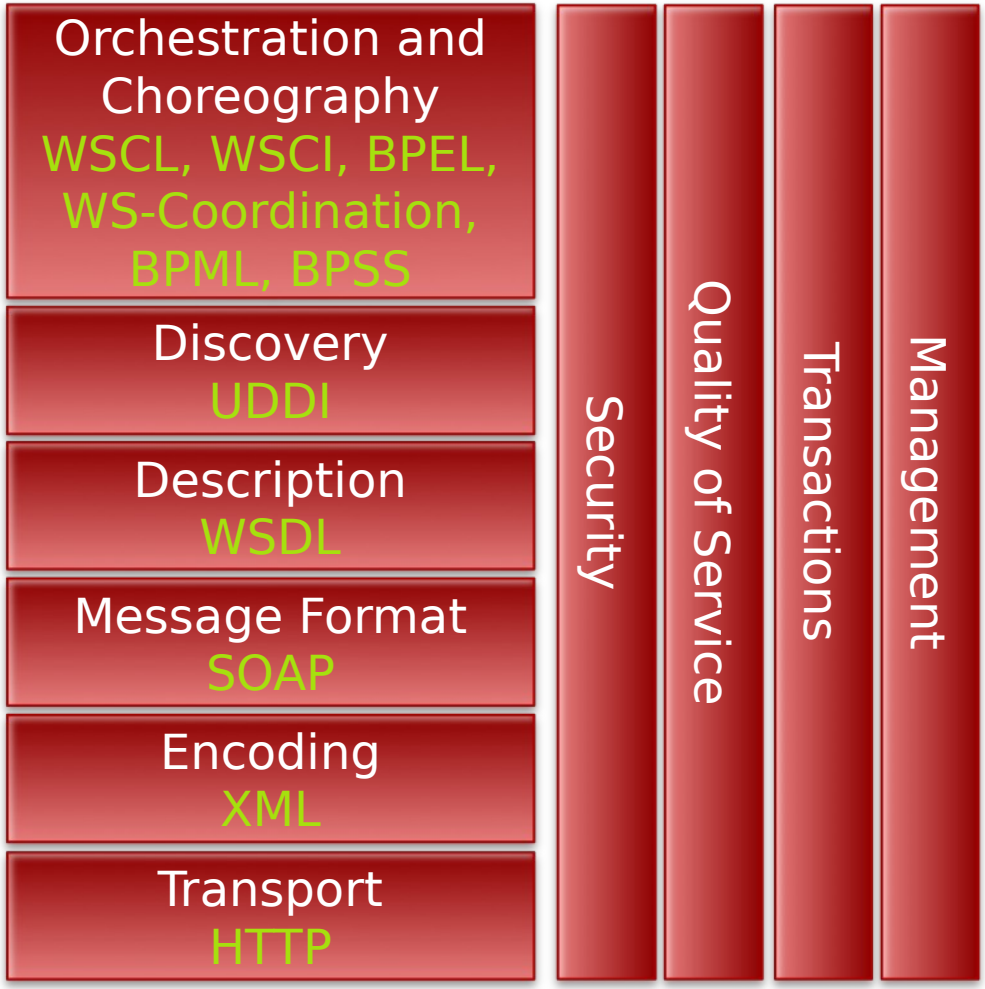
Principles of SOA

- Services
  - Share a **formal contract**
  - Are **loosely coupled**
  - **Abstract underlying logic**
  - Are **composable**
  - Are **reusable**
  - Are **autonomous**
  - Are **stateless**
  - Are **discoverable**

A SOA Characterization



A SOA Technology view: WS* Protocol Stack



WSCL	Web Services Conversation Language
WSCI	Web Service Choreography Interface
BPEL	Business Process Execution Language
WS	Web Services
BPML	Business Process Modeling Language
BPSS	Business Process Specification Schema
UDDI	Universal Description, Discovery and Integration
WSDL	Web Services Description Language
SOAP	Simple Object Access Protocol
XML	eXtensible Markup Language
HTTP	Hypertext Transfer Protocol

Why

- “The quest is **to find a solution that simplifies development and implementation, supports effective reuse of software assets, and leverages the enormous and low-cost computing power now at our fingertips**. While some might claim that service-oriented architecture (SOA) is just the latest fad in this illusive quest, tangible results have been achieved by those able to successfully implement its principles”

- “companies that have embraced SOA have eliminated huge amounts of redundant software, reaped major cost savings from simplifying and automating manual processes, and realized big increases in productivity”

(Open Source SOA, Jeff Davis)

PA165 Enterprise Java
2014-2015

REpresentational State Transfer (REST)

Bruno Rossi & Juha Rikkilä

Objectives and content

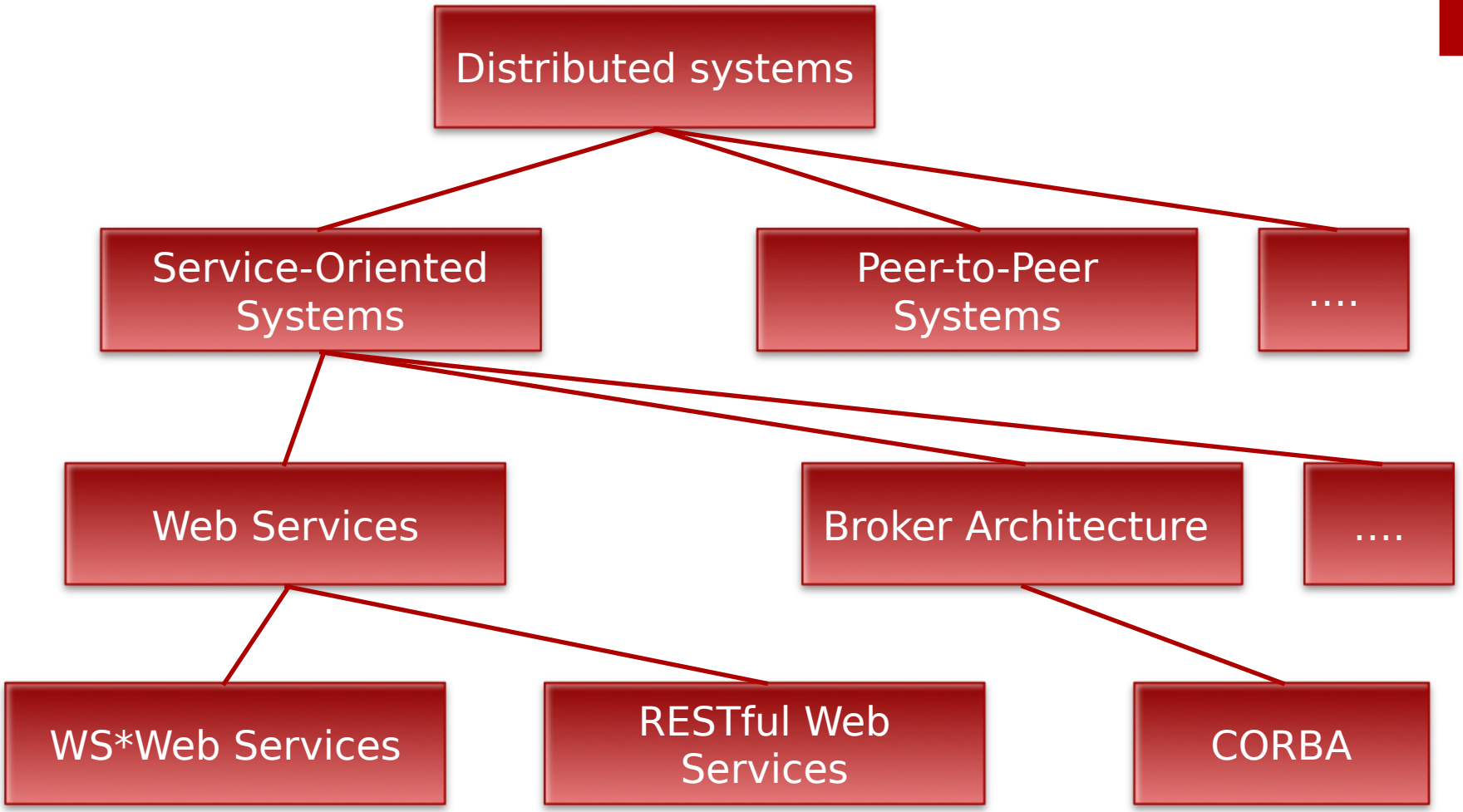
Objectives

Obtain overall understanding of the REST architectural style and its implementation in web.

Content

- Distributed systems
- REST, RESTFUL
- URI
- HTTP, HTTP methods
- Cache, Proxy, Gateway
- Security
- Summary, the six constraints, the principles of the uniform interface

Distributed Systems



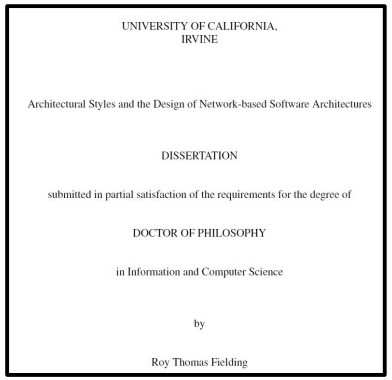
REST=Representational State Transfer

REST

- REpresentational State Transfer
  - Named by Roy Fielding in his Ph.D thesis

"Architectural Styles and the Design of Network-based Software Architectures"

http://ics.uci.edu/~fielding/pubs/dissertation/top.htm


UNIVERSITY OF CALIFORNIA,
IRVINE

Architectural Styles and the Design of Network-based Software Architectures

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

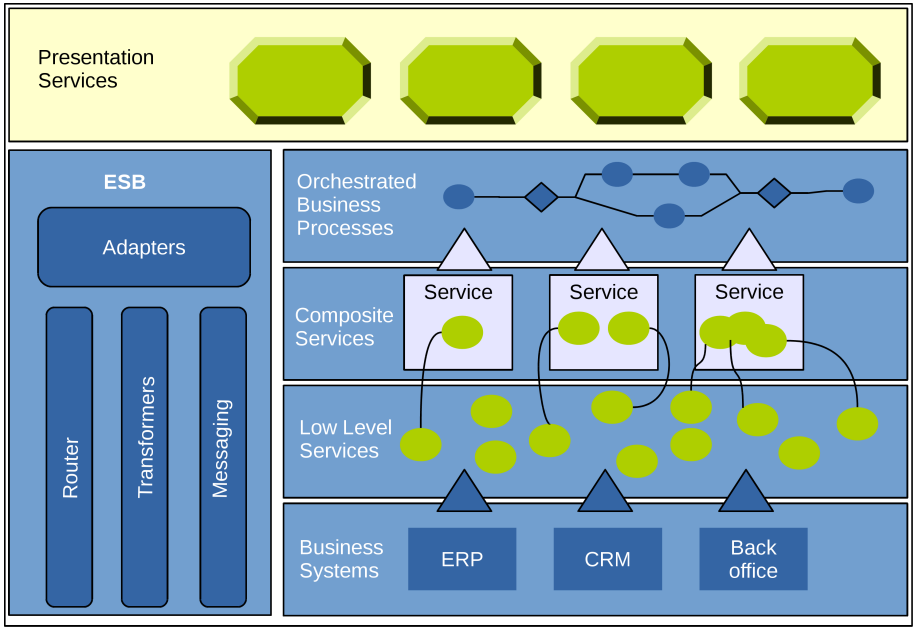
by

Roy Thomas Fielding

- **it is an architectural style**: REST is a sort of reverse-engineering of how the Web works. HTTP and URIs were written with the REST principles in mind before they were formalized
- The original idea behind Representational State Transfer is to mimic the behaviour of Web applications : as a net of Web pages and links, resulting in the next page (state change)
- REST is thoughts in the context of HTTP, but it is not limited to that protocol.

WS* vs. RESTful Web services

REST & SOA



- How does **REST fit in the SOA characterization**?
- What about the **SOA principles**?

Services

Share a **formal contract**

Are **loosely coupled**

Abstract underlying logic

Are **composable**

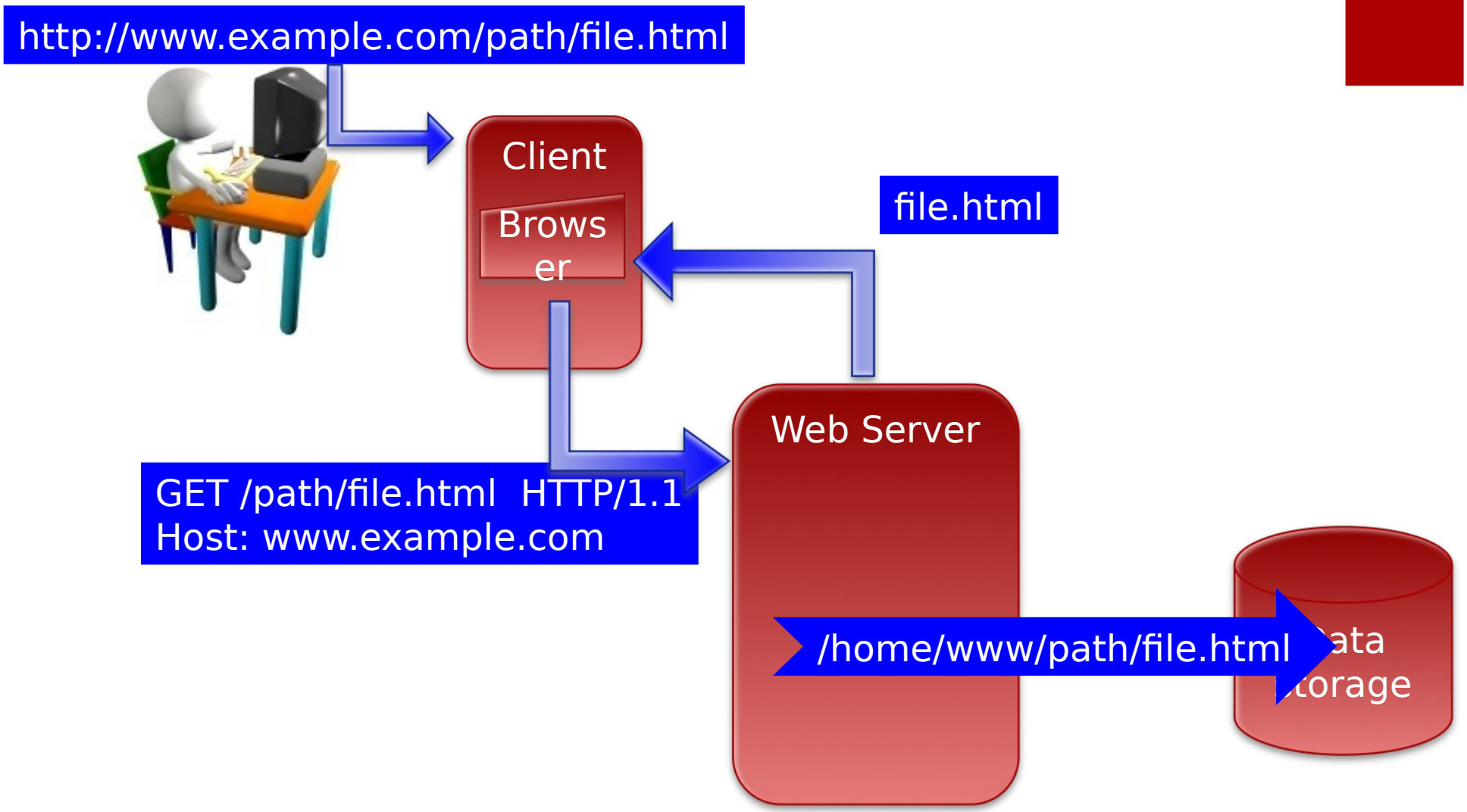
Are **reusable**

Are **autonomous**

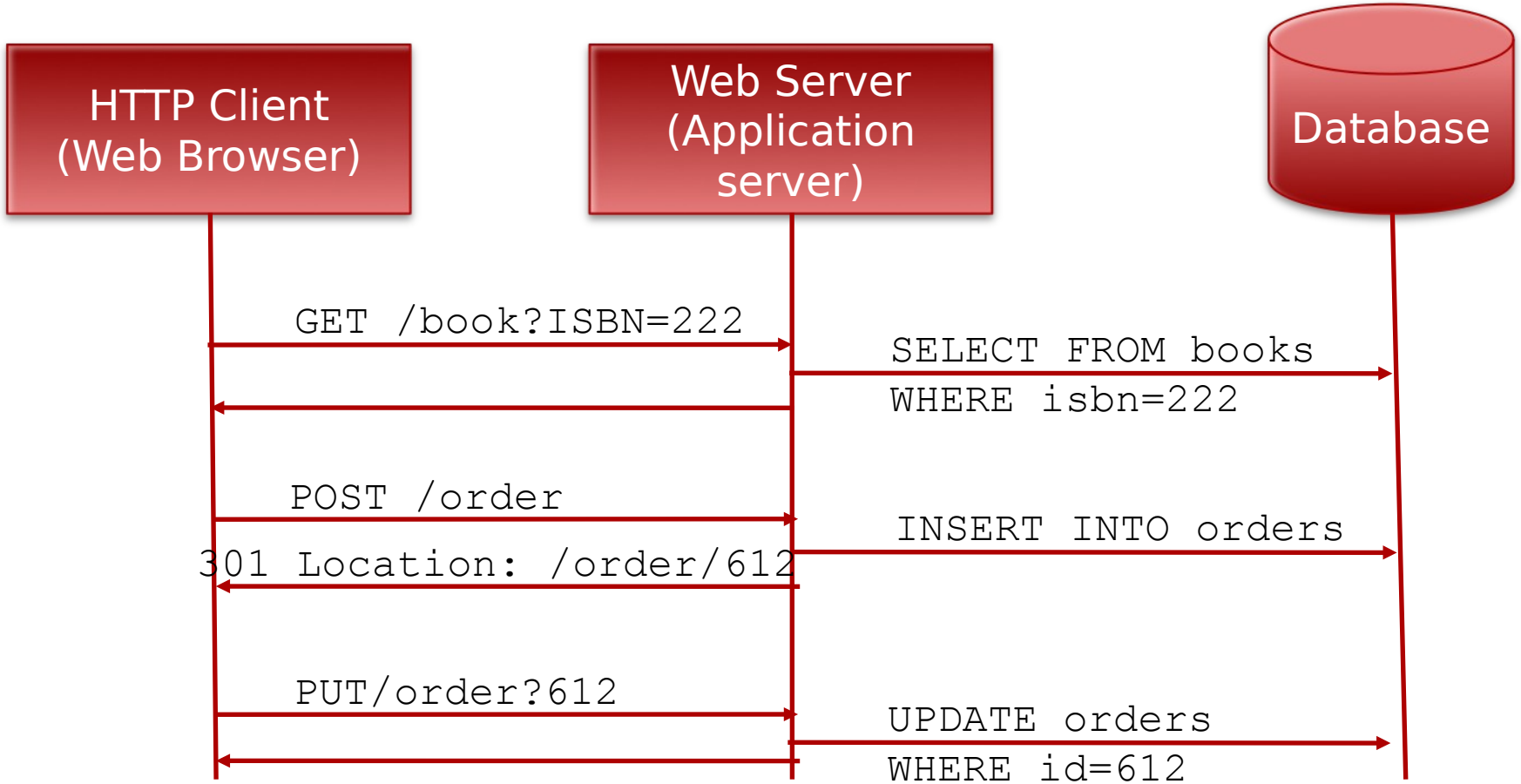
Are **stateless**

Are **discoverable**

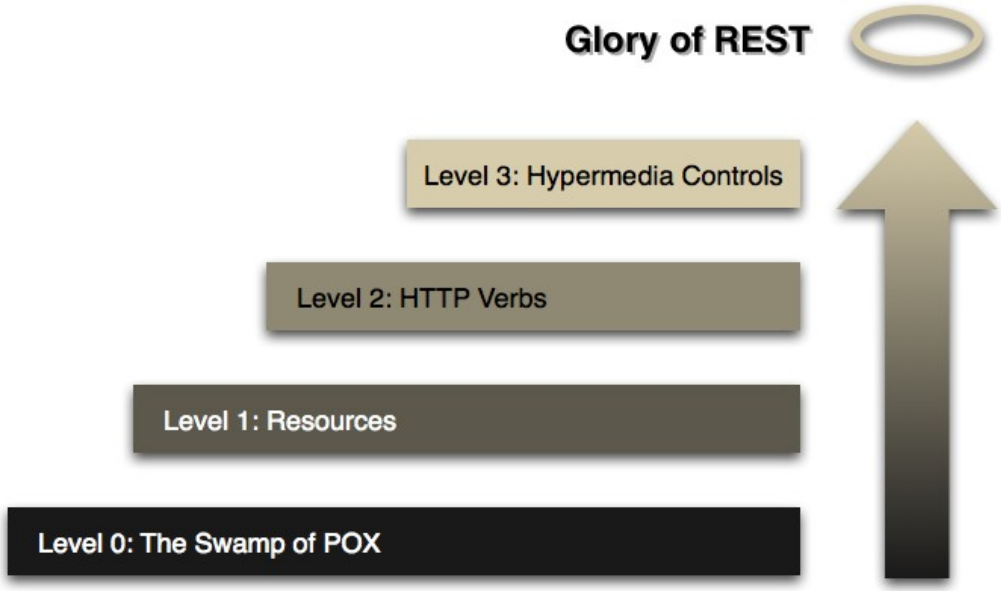
Browsing


http://www.example.com/path/file.html
Client
Brows
er
file.html
Web Server
GET /path/file.html HTTP/1.1
Host: www.example.com
/home/www/path/file.html
ata
torage

An example


HTTP Client
(Web Browser)
Web Server
(Application
server)
Database
GET /book?ISBN=222
SELECT FROM books
WHERE isbn=222
POST /order
INSERT INTO orders
301 Location: /order/612
PUT/order?612
UPDATE orders
WHERE id=612

REST Maturity Models



http://martinfowler.com/articles/richardsonMaturityModel.html

REST Principles (1/4)

- **REST services are stateless**. From Fieldings' thesis: *“each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server”*
- So, server sessions should not be used → all needed to process a request should be available in the request
- **Messages are self-describing**
- No need to start negotiation to understand how to communicate with a service
- Specific to HTTP, URI have semantics

REST Principles (2/4)

- In REST, r**esources are manipulated through the exchange of representations of the resources**
  - The components in the system exchange data (usually XML documents) → this represents a resource.

- REST-based architectures communicate primarily through the transfer of representations of resources
  - Resources have multiple representations (e.g. XML, JSON, XHTML, JPEG img)

REST Principles (3/4)

- **RESTful services have a uniform interface**
  - No WSDL in REST
  - Standard HTTP methods GET, POST, PUT, DELETE, etc…
  - Protocol independence (although by default HTTP is relied on)
- **REST-based architectures are built with resources**

→ Resources are uniquely identified by URIs

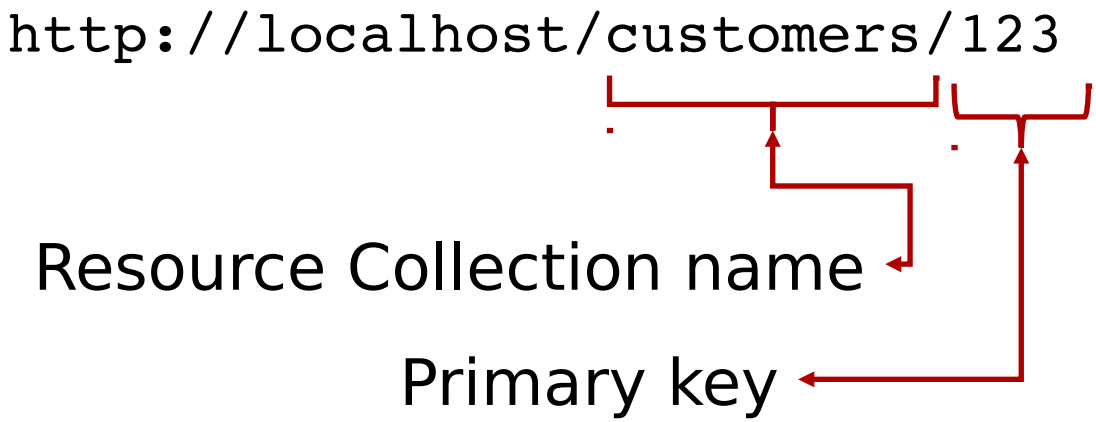
REST Principles (4/4)

- **Hypermedia as the engine of application state (HATEOS)**
- Fielding defines hypertext as: *"the simultaneous presentation of information and controls such that the information becomes the affordance through which the user (or automaton) obtains choices and selects actions"*
- This is important because the implication is that: *every resource returned by a server will allow to follow the URIs to any next step*

See http://spring.io/understanding/HATEOAS

http://spring.io/guides/tutorials/bookmarks/#_building_a_hateoas_rest_service

URI, example


http://localhost/customers/123
Resource Collection name
Primary key

HTTP Methods, for both collection and single item

GET

- to retrieve information
- Retrieves a given URI idempotent, should not initiate a state
- Cacheable

PUT

- to update information
- Full entity create/replace used when you know the “id”

POST

- to add new information
- Add the entity as a subordinate/append to the POSTed resource

DELETE

- to remove (logical) an entity

REST Methods

Method	Collection of resources, e.g. <host:port>/<context>/resources	Single item, e.g. <host:port>/<context>/resources/1
@GET	Get a list of all the resources	Retrieve data for resource with id 1
@PUT	Update the collection with a new one	Update the resource with id 1
@POST	Create a new member resource	Create a sub-resource under resource with id 1
@DELETE	Delete the whole collection	Delete the resource with id 1
@HEAD	Retrieve meta-data information according to HTTP head request	Retrieve data for resource with id 1

Safety and Idempotence

- The term "***safe***" means that if a given method is called, the resource state on the server remains unchanged
- By specifications, GET and HEAD should always be safe – clearly it is up to the developers not to violate this hidden specification
- PUT, DELETE are considered unsafe, while for POST generally depends

Safety and Idempotence

- The word "***idempotent***" means that, independently from how many times a given method is invoked, the end result is the same.
- GET and HEAD are an example of an idempotent operation
- PUT is as well idempotent: if you add several times the same resource, it should be only inserted once

DELETE is as well idempotent: issuing delete several times should yield the same result – the resource is gone (but what about **`DELETE /items/last`** ?)

- POST is generally not considered an idempotent operation

HTTP Request/Response As REST

Request

Method

```
GET /customer/{id}/items HTTP/1.1
Host: localhost
Accept: application/xml
```

Resource

Response

State transfer

```
HTTP/1.1 200 OK
Date: Fri, 22 Jun 2013 17:21:35 GMT
Server: Apache/1.3.6
Content-Type: application/xml; charset=UTF-8

<?xml version="1.0"?>
<items xmlns="…">
  <item>…</item>
  …
</items>
```

Representation

JAX-RS (Jersey) vs Spring

JAX-RS

```
@Path("/customers")
@Singleton
public class CustomersController {

 @GET
 @Path("customers")
 @Produces(MediaType.TEXT_PLAIN)
    public String getPlain() {
        ....
    }
 ....
}
```

Spring

```
@RestController
@RequestMapping("/customers")
public class CustomersController {

 @RequestMapping(value="customers",
      method=RequestMethod.GET,
       headers="Accept=text/plain")
 public String getPlain() {
    ....
 }
  ...
}
```

or produces={MediaType.TEXT_PLAIN}

http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html

Multiple Representations

- Data in a variety of formats
  - XML
  - JSON (JavaScript Object Notation)
  - XHTML

@Produces(MediaType.TEXT_PLAIN [, more-types])	For a method annotated with @GET, specifies the type of data that is returned
@Consumes(type [, more-types])	The type of data that is consumed by the method, for example, "text/plain"

- Content negotiation
  - Accept header

```
GET /customers
Accept: application/json
```

  - URI-based

```
GET /customers.json
```

  - parameter-based

```
http://localhost/customers?type=json
```

Content Negotiation

- Example in JAX-RS

```
@Consumes("text/*")
@Path("/customer")
public class Customer {
  @POST
  public String stringCustomer(String customer)
  {...}

  @Consumes("text/xml")
  @POST
  public String xmlCustomer(Customer customer)
  {...}
}
```

```
POST /customer
content-type: text/xml

<customer name="Roy" surname="Fielding"/>
```

Content Negotiation

- Example in JAX-RS

```
@Produces("text/*")
@Path("/customer")
public class Customer {
  @GET
  public String get()
  {...}

  @Produces("text/xml")
  @GET
  public String getXML()
  {...}
}
```

GET /customer
Accept: text/xml

Content Negotiation

Configuration example in Jersey in web.xml

```
<init-param>

   <param-name>
   jersey.config.server.mediaTypeMappin
   gs
   </param-name>
   <param-value>txt : text/plain, xml :
   application/xml, json :
   application/json
   </param-value>

</init-param>
   ....
```

Configuration example in Spring

```
@Configuration
@EnableWebMvc
public class WebConfig extends
  WebMvcConfigurerAdapter {
  @Override
  public void
  configureContentNegotiation(ContentNegotiation
                Configurer configurer) {
    configurer.favorPathExtension(false).
    favorParameter(true).
  parameterName("mediaType").
  ignoreAcceptHeader(true).
  defaultContentType(MediaType.APPLICATION_JSON)
  .mediaType("txt",MediaType.TEXT_PLAIN).
  mediaType("xml",MediaType.APPLICATION_XML).
  mediaType("json",MediaType.APPLICATION_JSON);
  }
}
```

Managing Exceptions & Return Codes

- It is responsibility of the developer to provide consistent behaviour of their REST API:

- **Successful HTTP response code numbers** go from 200 to 399. The creation will return 200, “OK” if the object returned is not null. 204, “No Content” is returned when a null object was retrieved. As well as if the return is of type void 204, “No Content” is returned.

- **HTTP error response code numbers** go from 400 to 599. A 404 “Not Found” response code will be sent back to the client if the resource requested is not found. A bad request "400" is sent back in case of bad parameters. All the codes in the range 5xx indicate internal errors of the application.

Managing Exceptions in JAX-RS

- In JAX-RS you can use the class *javax.ws.rs.core.Response.ResponseBuilder* to return appropriate HTTP codes, e.g.:

```
....
ResponseBuilder builder = Response.ok(object);
builder.header("header-name", "value");  // set some header value
return builder.build();
....
```

Managing Exceptions in JAX-RS

you can use the enum javax.ws.rs.core.Response.Status (*https://docs.oracle.com/javaee/6/api/javax/ws/rs/core/Response.Status.html*) to return error codes, example:

```
return Response.status(Status.GONE).build();
```

Enum Constant Summary
ACCEPTED 202 Accepted, see HTTP/1.1 documentation.
BAD_REQUEST 400 Bad Request, see HTTP/1.1 documentation.
CONFLICT 409 Conflict, see HTTP/1.1 documentation.
CREATED 201 Created, see HTTP/1.1 documentation.
FORBIDDEN 403 Forbidden, see HTTP/1.1 documentation.
GONE 410 Gone, see HTTP/1.1 documentation.
INTERNAL_SERVER_ERROR 500 Internal Server Error, see HTTP/1.1 documentation.
MOVED_PERMANENTLY 301 Moved Permanently, see HTTP/1.1 documentation.
NO_CONTENT 204 No Content, see HTTP/1.1 documentation.
NOT_ACCEPTABLE 406 Not Acceptable, see HTTP/1.1 documentation.
NOT_FOUND 404 Not Found, see HTTP/1.1 documentation.
NOT_MODIFIED 304 Not Modified, see HTTP/1.1 documentation.
OK 200 OK, see HTTP/1.1 documentation.
PRECONDITION_FAILED 412 Precondition Failed, see HTTP/1.1 documentation.
SEE_OTHER 303 See Other, see HTTP/1.1 documentation.
SERVICE_UNAVAILABLE 503 Service Unavailable, see HTTP/1.1 documentation.
TEMPORARY_REDIRECT 307 Temporary Redirect, see HTTP/1.1 documentation.
UNAUTHORIZED 401 Unauthorized, see HTTP/1.1 documentation.

Managing Exceptions in JAX-RS

- You can also throw exceptions that will be handled by the JAX-RS runtime , you can use javax.ws.rs.WebApplicationException:

```
...
if (object == null) {
  throw new WebApplicationException(Response.Status.NOT_FOUND);
}
...
```

Managing Exceptions in JAX-RS

- … or you can use an exception mapper by implementing and registering instances of javax.ws.rs.ext.ExceptionMapper:

```
@Provider
public class EntityNotFoundMapper implements
                                        ExceptionMapper<EntityNotFoundException> {

     public Response toResponse(EntityNotFoundException e) {
        return Response.status(Response.Status.NOT_FOUND).build();
     }

  }
```

Managing Exceptions in Spring

- Similarly to JAX-RS you can manage exceptions/return codes in different ways. Easiest ways is per single exception:

```
@ResponseStatus(value=HttpStatus.NOT_FOUND, reason="404 Not Found")
    public class CustomerNotFoundException extends RuntimeException {
        // ...
    }
```

```
@RequestMapping(value="customers/{id}", method=RequestMethod.GET,
       headers="Accept=text/plain")
public String getCustomer(@PathVariable("id") long id) {
   ....
        customer = customersService.getCustomerById(id);
        if (customer == null) throw new OrderNotFoundException(id);
   ....
}
```

Managing Exceptions in Spring

- Another way is to manage exceptions thrown in the <u>same</u> controller when managing requests

```
  @RestController
  public class MyController {

  ...
  @ResponseStatus(value=HttpStatus.NOT_FOUND, reason="404 Not Found")
  @ExceptionHandler(CustomerNotFoundException.class)
  public void notFound() {
     ...
  }
 ...
}
```

Managing Exceptions in Spring

- Another way is to have a global advice using @ControllerAdvice that will manage exceptions for all controllers

```
@ControllerAdvice
class GlobalControllerExceptionHandler {
    @ResponseStatus(HttpStatus.NOT_FOUND)
    @ExceptionHandler(CustomerNotFoundException.class)
    public void handleCustomerNotFound() {
        ...
    }
}
```

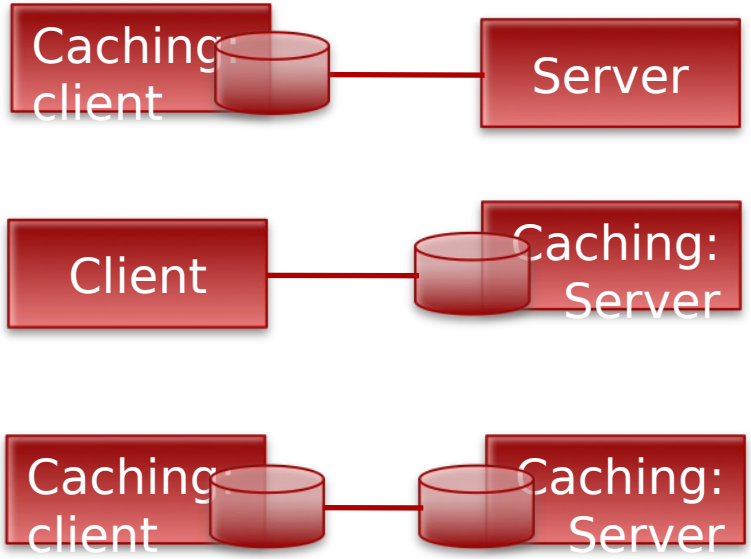
Managing Exceptions in JAX-RS

- See http://www.w3.org/Protocols/rfc2616/rfc2616.html for the expected behaviour of GET, POST, PUT, DELETE, HEAD

Caching

Basic setup

Caching options



Example of Caching in JAX-RS

```
@Path("/items/{id}")
@GET
public Response getItem(@PathParam("id") long id){

        Item item = ItemService.getItem(id);

        CacheControl cc = new CacheControl();
        cc.setMaxAge(86400); // 86400 secs-> one day
        cc.setPrivate(true); // only last in the call line should cache the resource

        ResponseBuilder builder = Response.ok(item);
        builder.cacheControl(cc);
        return builder.build();
}
```

Example of Caching in JAX-RS

```
@Path("/items/cond/{id}")
@GET
public Response getItem(@PathParam("id") long id, @Context Request request){

        Item item = ItemService.getItem(id);
        CacheControl cc = new CacheControl();
        cc.setMaxAge(86400);

        EntityTag etag = new EntityTag(Integer.toString(item.hashCode()));
        ResponseBuilder builder = request.evaluatePreconditions(etag);

        // if builder is null then the cached resource changed
        if(builder == null){
                builder = Response.ok(item);  // this will return HTTP 200 OK
                builder.tag(etag);
        }

        builder.cacheControl(cc);  // if not send HTTP 304 Not Modified
        return builder.build();
}
```

Example of Caching in Spring

From http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/context/request/WebRequest.html#checkNotModified-java.lang.String-

```
public String myHandleMethod(WebRequest request, Model model) {
   String eTag = // application-specific calculation

   if (request.checkNotModified(eTag)) {
     // shortcut exit - no further processing necessary
     return null;
   }

   // further request processing, actually building content
   model.addAttribute(...);
   return "myViewName";
 }
```

Other more advanced ways → using EhCache

Example of Caching in JAX-RS

```
> curl -X GET -i http://localhost:8084/JerseyREST/service/items/cond/1


HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
ETag: "3242771"
Cache-Control: no-transform, max-age=86400
Content-Type: text/plain
Content-Length: 4
Date: Thu, 20 Nov 2014 12:11:35 GMT

> curl -i -X GET http://localhost:8084/JerseyREST/service/items/cond/1
--header 'If-None-Match: "3242771"'

-Match: "3242771"'
HTTP/1.1 304 Not Modified
Server: Apache-Coyote/1.1
ETag: "3242771"
Cache-Control: no-transform, max-age=86400
Date: Thu, 20 Nov 2014 12:16:05 GMT
```

Let's dig into the details: Oracle Tutorials on RESTful Services with JAX-RS

Web Services:

http://docs.oracle.com/javaee/7/tutorial/doc/partwebsvcs.htm

Building RESTful Web Services with JAX-RS:

http://docs.oracle.com/javaee/7/tutorial/doc/jaxrs.htm#GIEPU

Accessing REST Resources with the JAX-RS Client API:

http://docs.oracle.com/javaee/7/tutorial/doc/jaxrs-client.htm#BABEIGIH