System Integration I: Web Services & REpresentational State Transfer (REST)

PA165 Enterprise Java

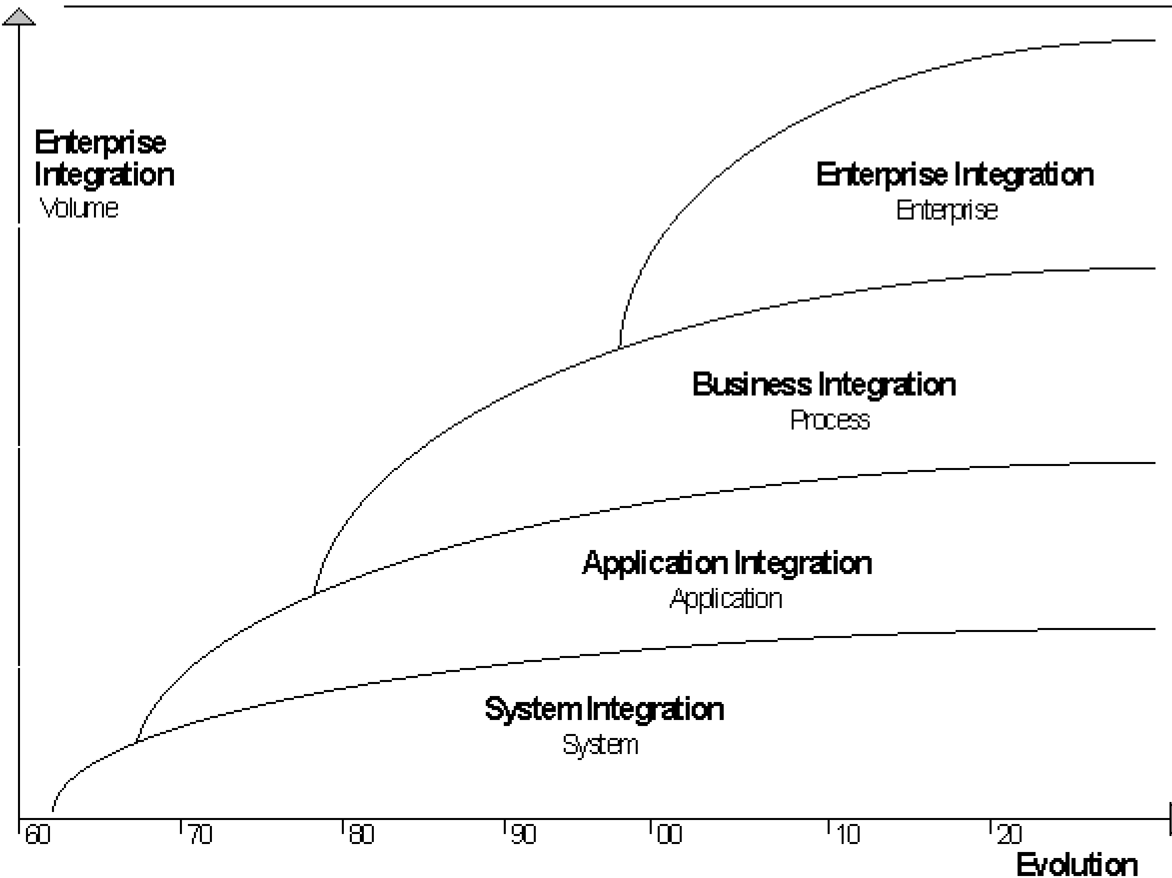
Bruno Rossi

Integration, SOC/SOA & Webservices

System Integration

- **System integration** (SI) is a software engineering process that aims at putting together different subsystems within an overall application.
- SI ensures that each integrated system functions and potentially can also add value by interconnecting different sub-systems/components

Enterprise Integration



Bringing interoperability between the different systems within an enterprise infrastructure

Enterprise Service Bus (ESB)

Integrating different business processes

Business Process Execution Language (BPEL)

Integrating different applications

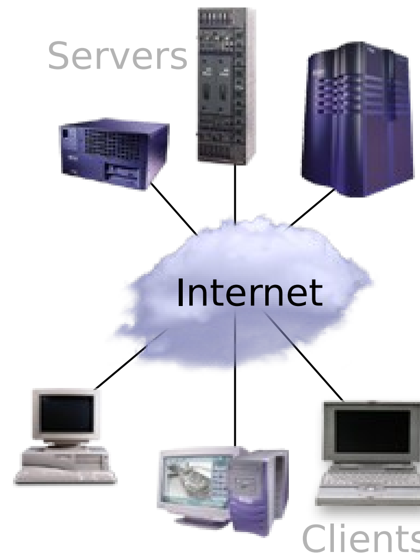
Common Object Request Broker Architecture (CORBA)

Integrating different sub-systems / components

COM (Component Object Model)

Distributed Computing Evolution

Client-Server(C/S) silos

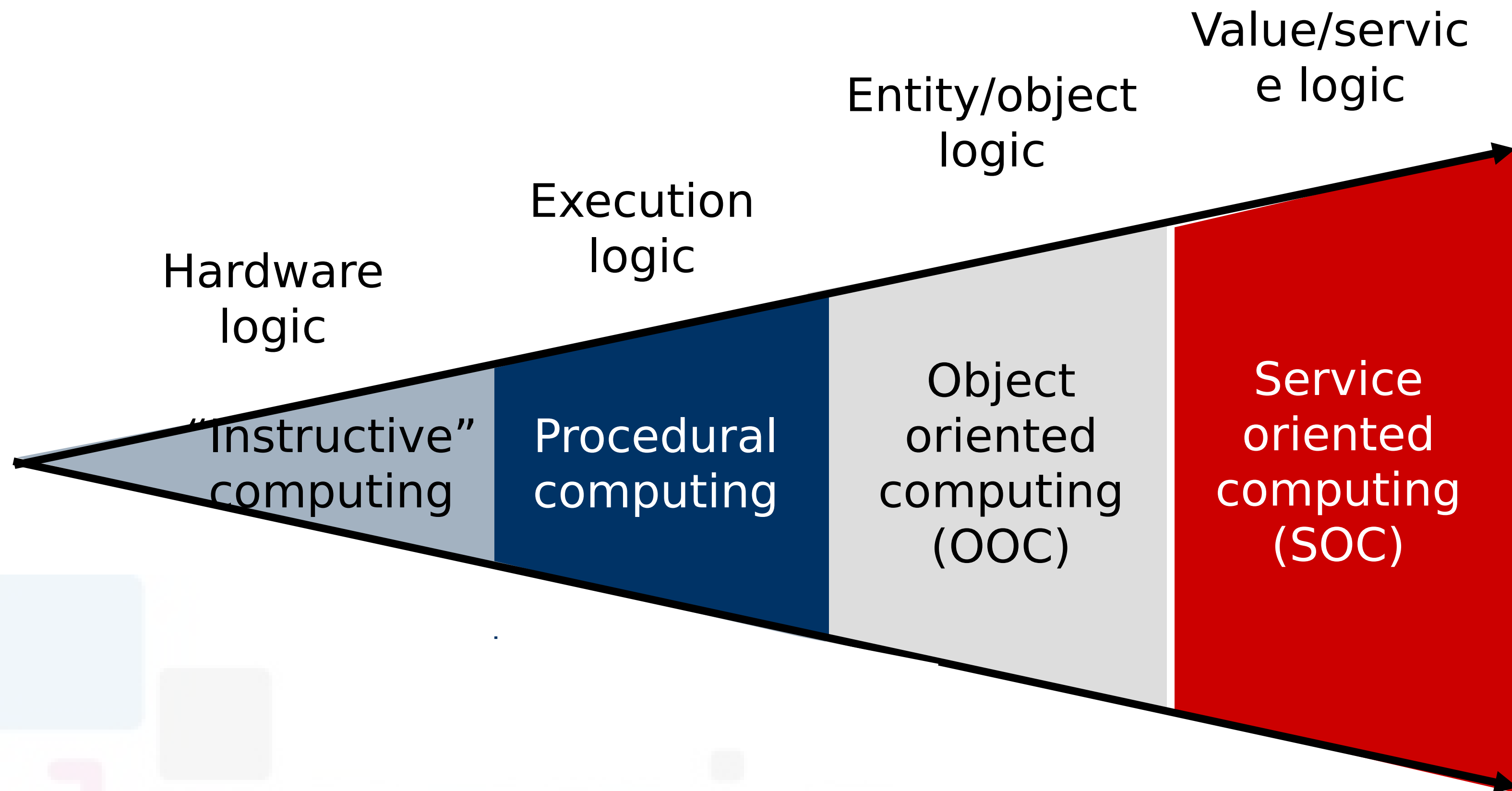


Web-based computing

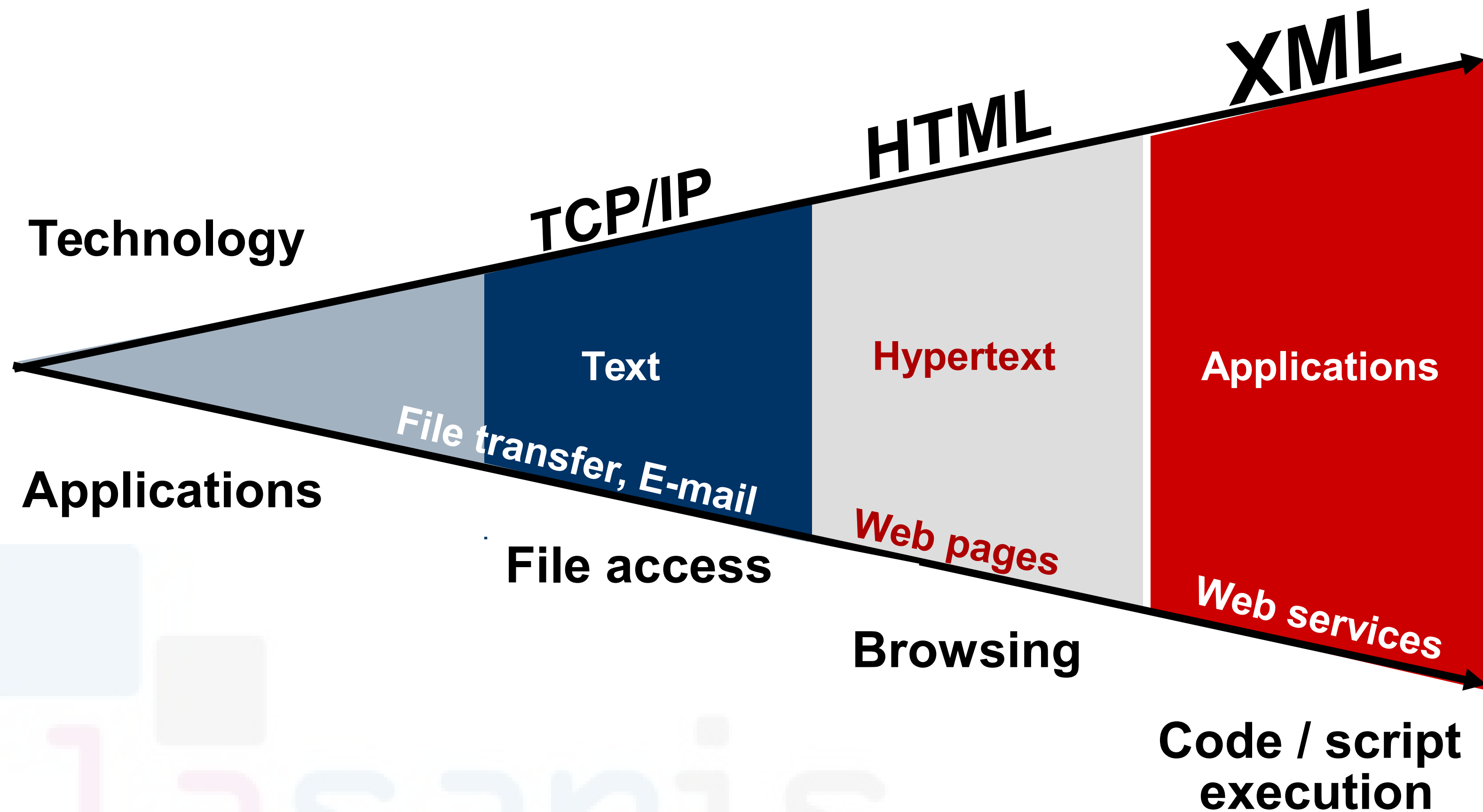
Web Services/Peer-to-Peer

lasaris

Evolution of software development /programming



Internet Evolution



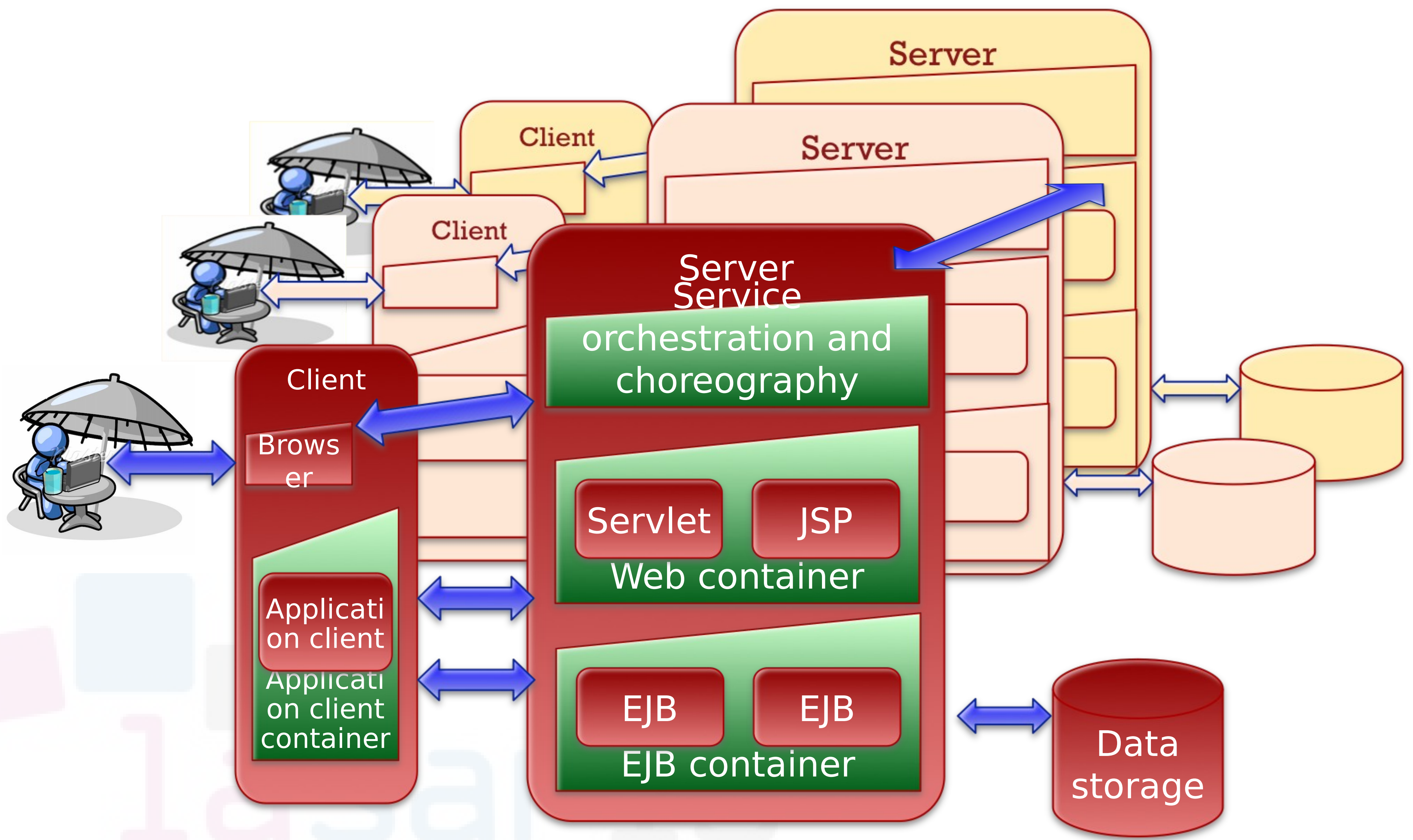
Service Oriented Computing (SOC)

- **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.

lasaris

Services Execution



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)*

lasaris

Some SOA definitions (2/2)

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...

lasaris

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)

lasaris

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.

What is a WebService

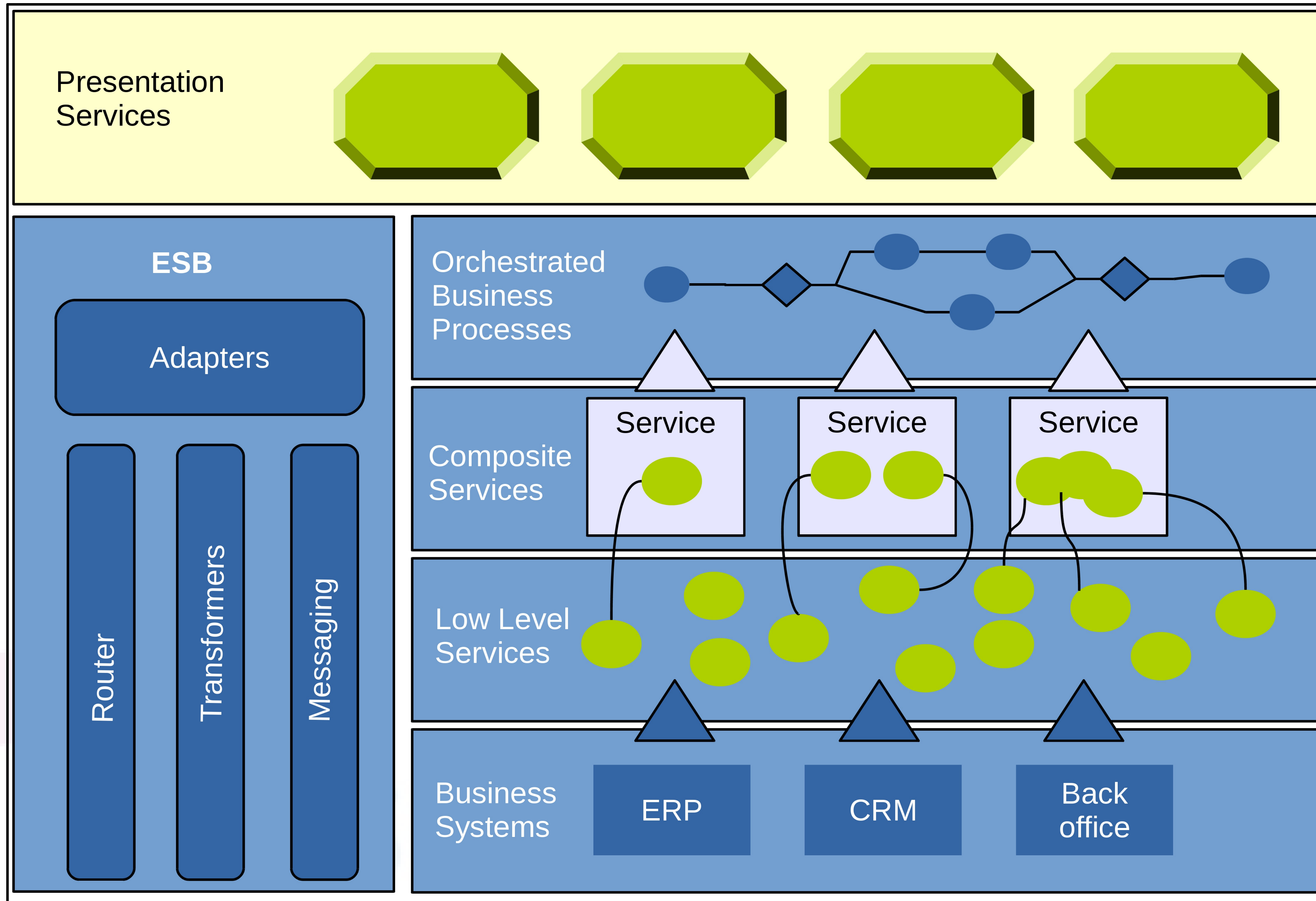
- A Web service is a **software system designed to support interoperable machine-to-machine interaction over a network**. It has an interface described in a machine processable format (usually WSDL).
- Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards

Principles of SOA

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

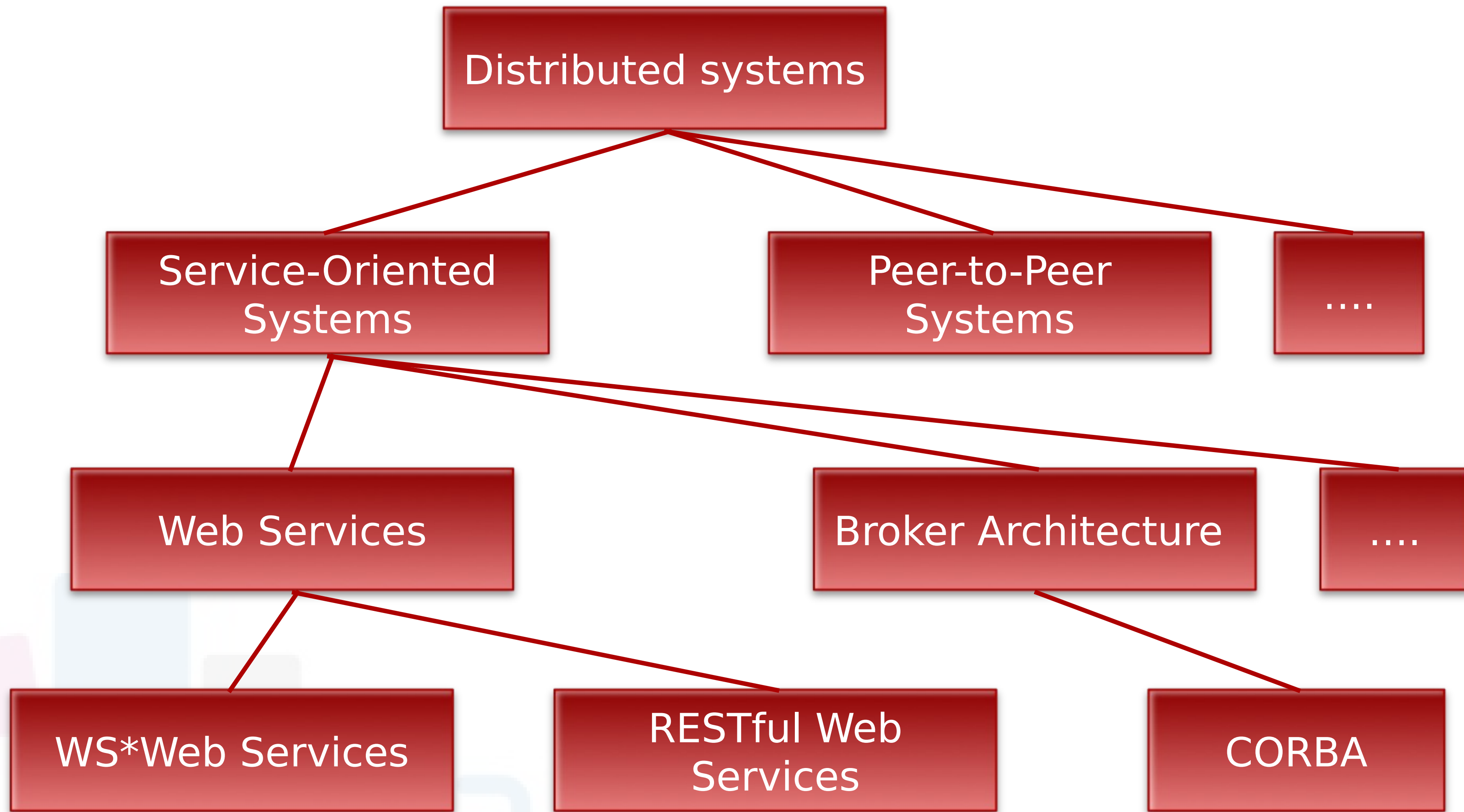
lasaris

A SOA Characterization



REpresentational State Transfer (REST)

Distributed Systems

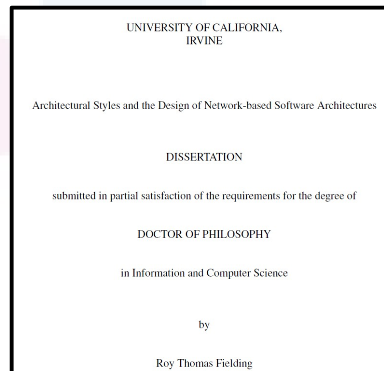


REST

- REpresentational State Transfer
  - Named by Roy Fielding in his Ph.D thesis from 2000

"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

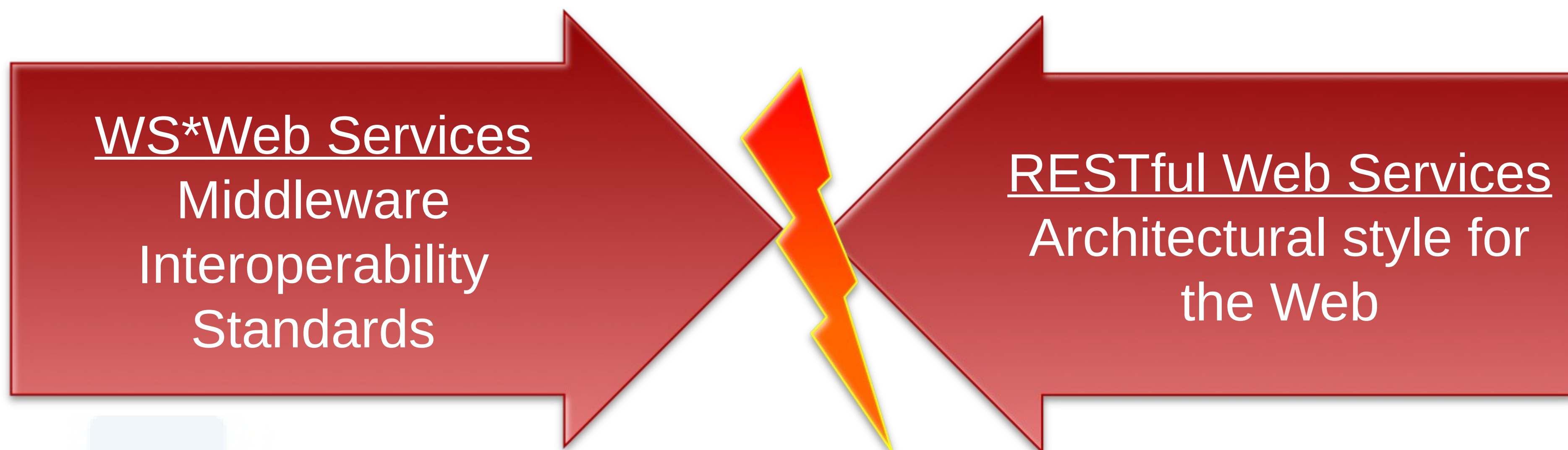
in Information and Computer Science

by

Roy Thomas Fielding

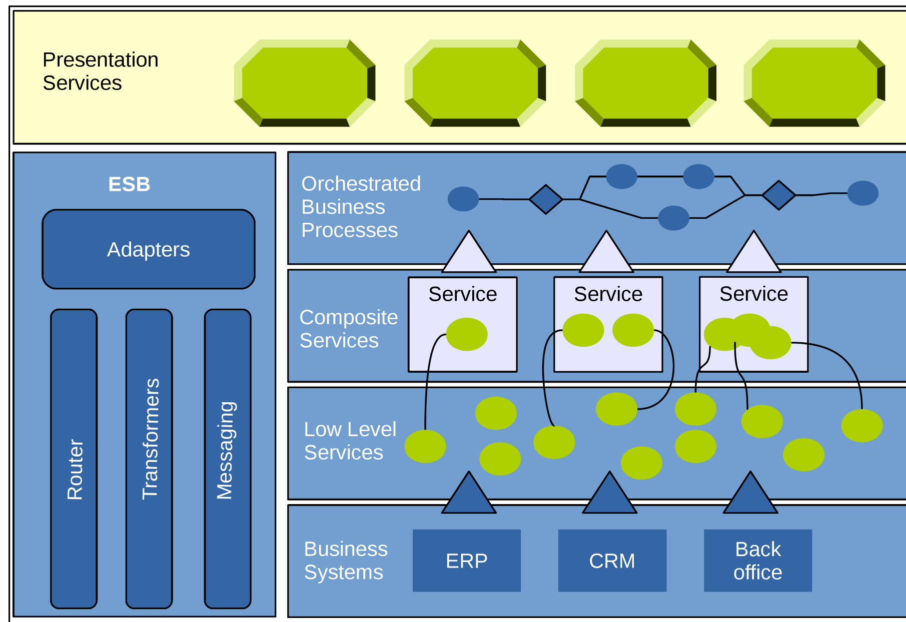
- **it is an architectural style**: REST is a sort of reverse-engineering of how the Web works. HTTP and URIs were written with the REST principles in mind before they were formalized
- The original idea behind Representational State Transfer is to mimic the behaviour of Web applications : as a net of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

WS* vs. RESTful Web services



lasaris

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**

lasaris

HTTP Request/Response as REST

Request

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

Method → GET; Resource → /customer/{id}/items

Response

```
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>
```

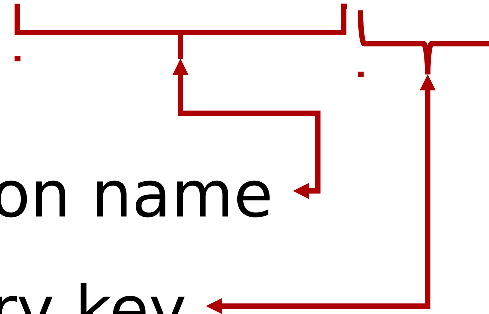
State transfer: the entire response

Representation: the XML body

lasaris

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

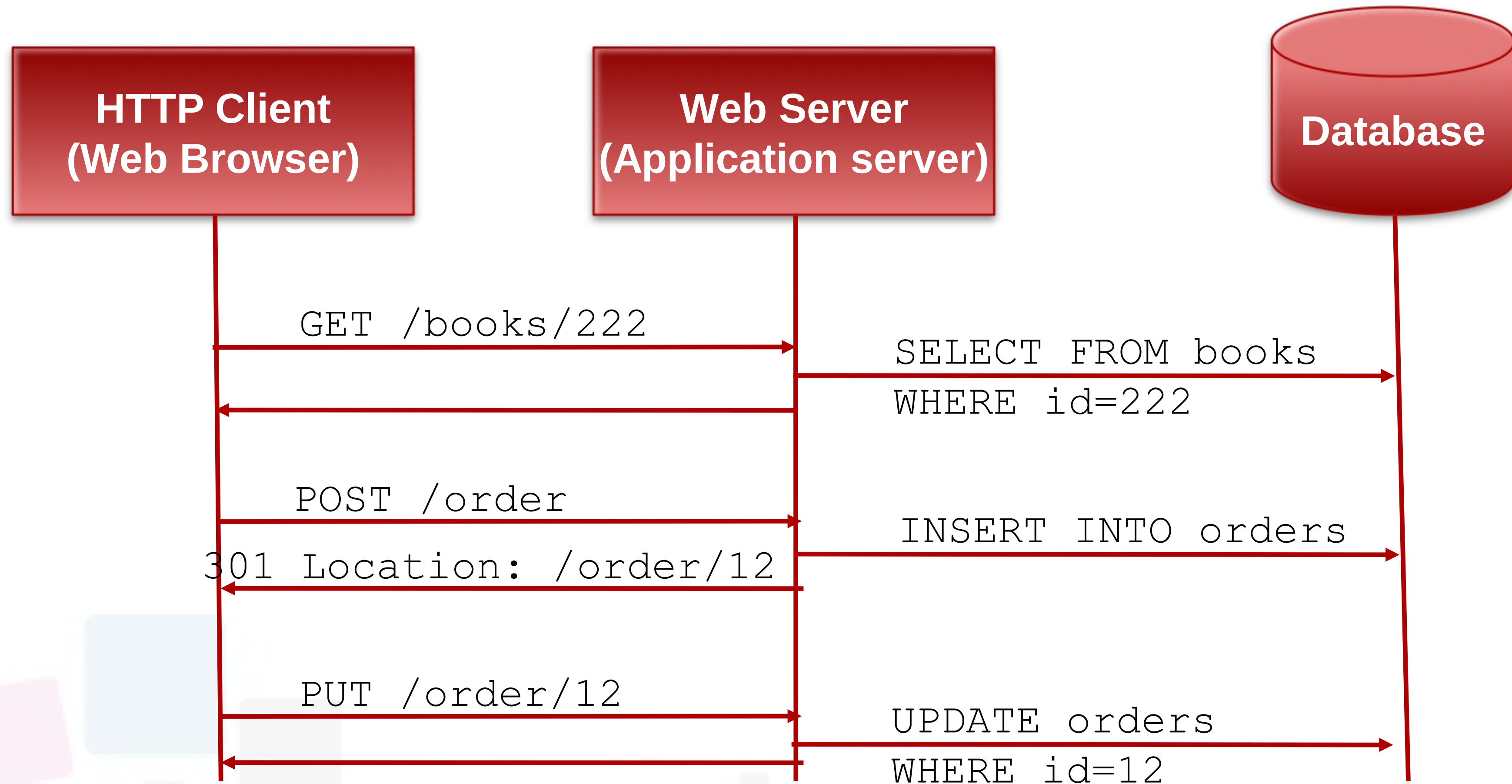
- to add new information
- Add the entity as a subordinate/append to the POSTed resource

DELETE

- to remove (logically) an entity

lasaris

An Example



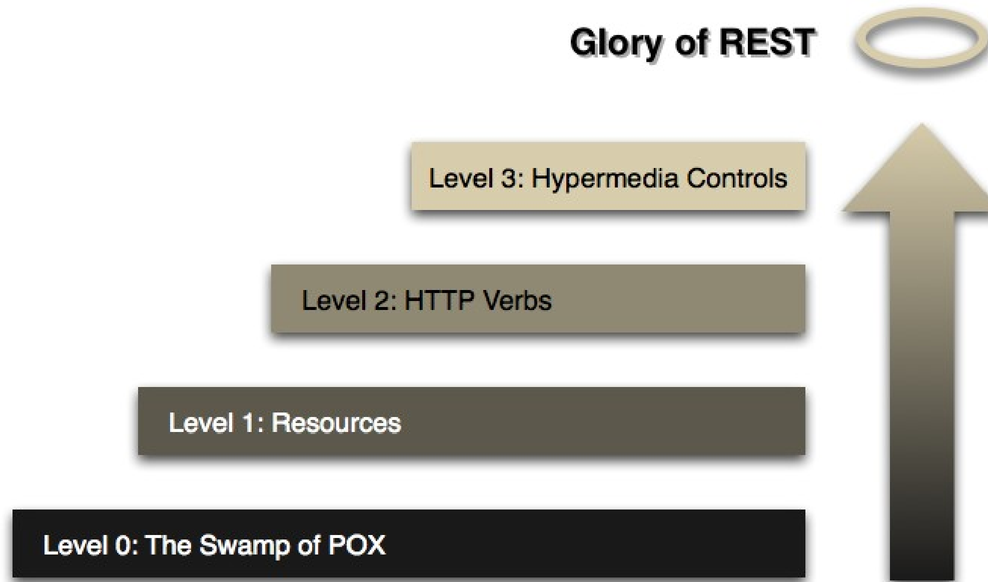
REST Methods

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
@OPTIONS	Retrieved allowed operations, e.g. Allow: GET, OPTIONS	Retrieved allowed operations, e.g. Allow: HEAD,GET,PUT,DELETE,OPTIONS
@PATCH	Partial modification of the collection	Partial modification of some attributes of resource with id 1

lasaris

REST Maturity Models

REST Maturity Model



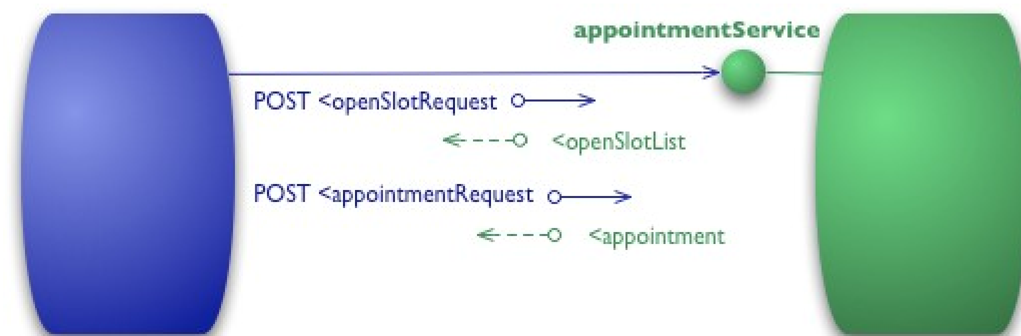
Explains different levels at which REST can be implemented

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

lasaris

Level 0 – The Swamp of POX*

- Looks more as a Remote Procedure Call system
- We post to an endpoint asking for different services
- There is no knowledge about resources, rather messages that are sent to the endpoints (and back responses)




```
POST /appointmentService HTTP/1.1
[various other headers]

<openSlotRequest date = "2010-01-04" doctor = "mjones"/>
```

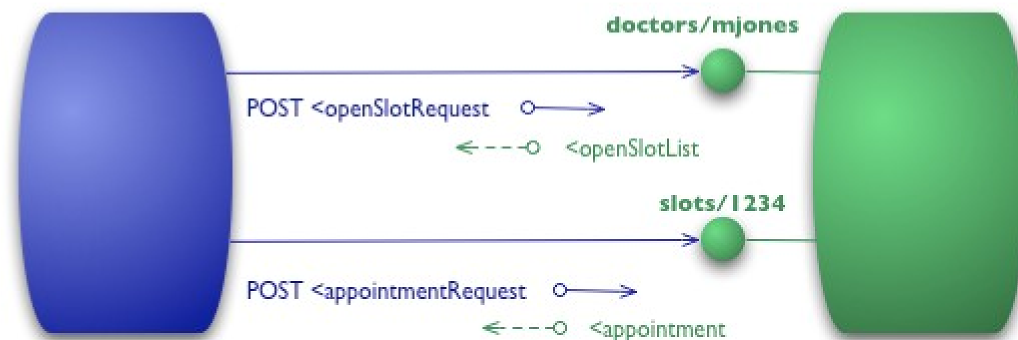
lasaris

* Plain Old XML

Level 1 – Resources

- At this level we introduce Resources
- We contact resources, not endpoints
- Instead of passing parameters, now we contact the specific resource

```
POST /doctors/mjones HTTP/1.1
[various other headers]

<openSlotRequest date = "2010-01-04"/>
```

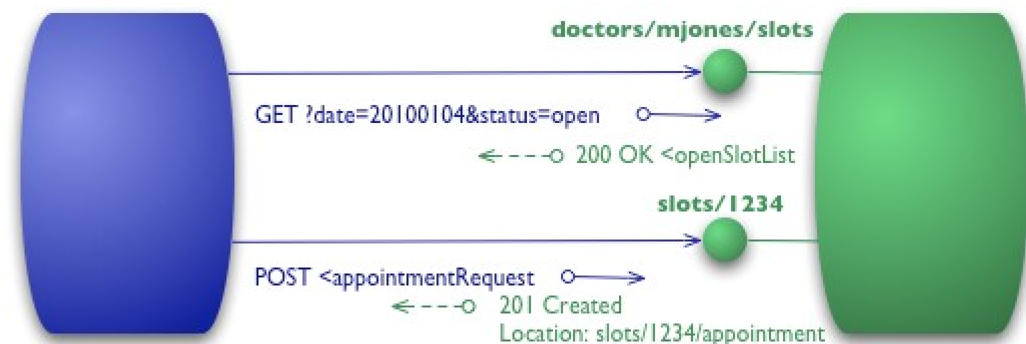


lasaris

Level 2 – HTTP Verbs

- At this level we start using HTTP verbs
- We start differentiating between POST and GET
- We also start using HTTP response codes
- We start differentiating “safe” vs “unsafe” operations

```
GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1
```



lasaris

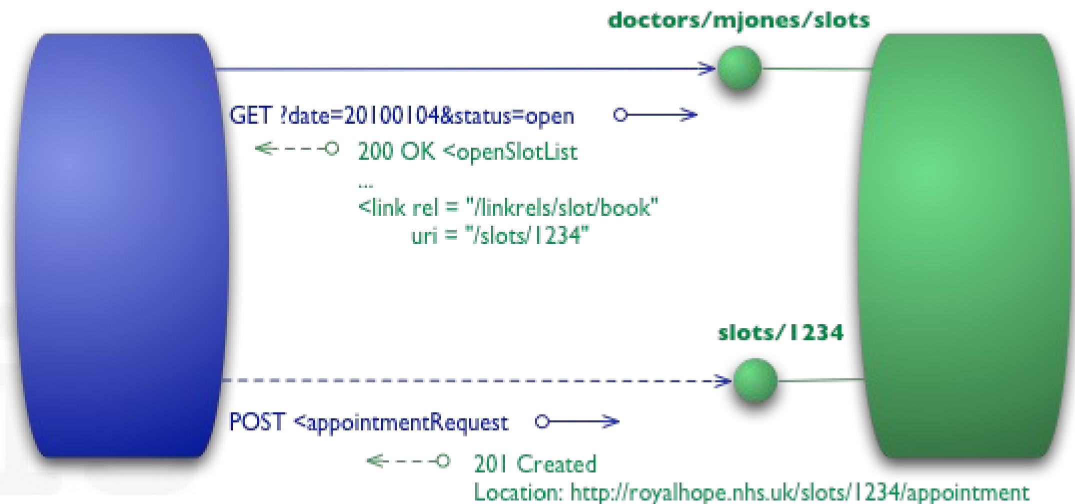
Level 3 – Hypermedia Controls (1/2)

- We introduce HATEOAS (Hypertext As The Engine Of Application State)

```
GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1
```

This time the response contains link to URI:

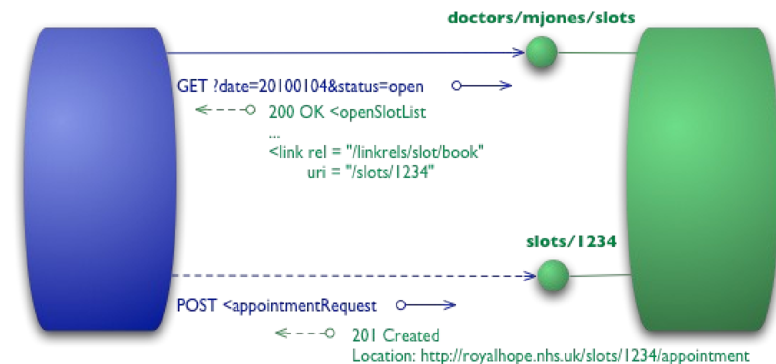
```
<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450">
     <link rel = "/linkrels/slot/book"
           uri = "/slots/1234"/>
  </slot>
...
```



Level 3 – Hypermedia Controls (2/2)

- This allows to create a more fluent flow of resources:

```
...
<link rel = "/linkrels/appointment/cancel"
        uri = "/slots/1234/appointment"/>
  <link rel = "/linkrels/appointment/addTest"
        uri = "/slots/1234/appointment/tests"/>
  <link rel = "self"
        uri = "/slots/1234/appointment"/>
  <link rel = "/linkrels/appointment/changeTime"
        uri = "/doctors/mjones/slots?date=20100104@status=open"/>
  <link rel = "/linkrels/appointment/updateContactInfo"
        uri = "/patients/jsmith/contactInfo"/>
  <link rel = "/linkrels/help"
        uri = "/help/appointment"/>
...
```



REST Principles

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

lasaris

REST Principles (2/4)

- In REST, **resources 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)

lasaris

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

lasaris

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

lasaris

Safety and Idempotence

Method	Safety	Idempotence
@GET	YES	YES
@PUT	NO	YES
@POST	NO	NO
@DELETE	NO	YES
@HEAD	YES	YES
@OPTIONS	YES	YES
@PATCH	NO	NO

lasaris

REST Best Practices

REST Best Practices (1/6)

- Have consistent usage of **resource names**, e.g. plural for resources → **`/users/1`**, **`orders/1`**
- Use URIs to deal with **relationships** → **`GET /users/1/orders`** to get all orders for a user
- Thinking in terms of **CRUD operations**
  - Example: using PUT and DELETE to set flags, rather than **`/users/1/enable /users/1/disable`**
  - If not possible (e.g. retrieval of multiple resources) then also **`/find`** or similar action might be appropriate
- **Filtering and sorting options** should be provided as parameters in the API, e.g. **`GET /users/1/orders?state=active&sorting=by-name`**

lasaris

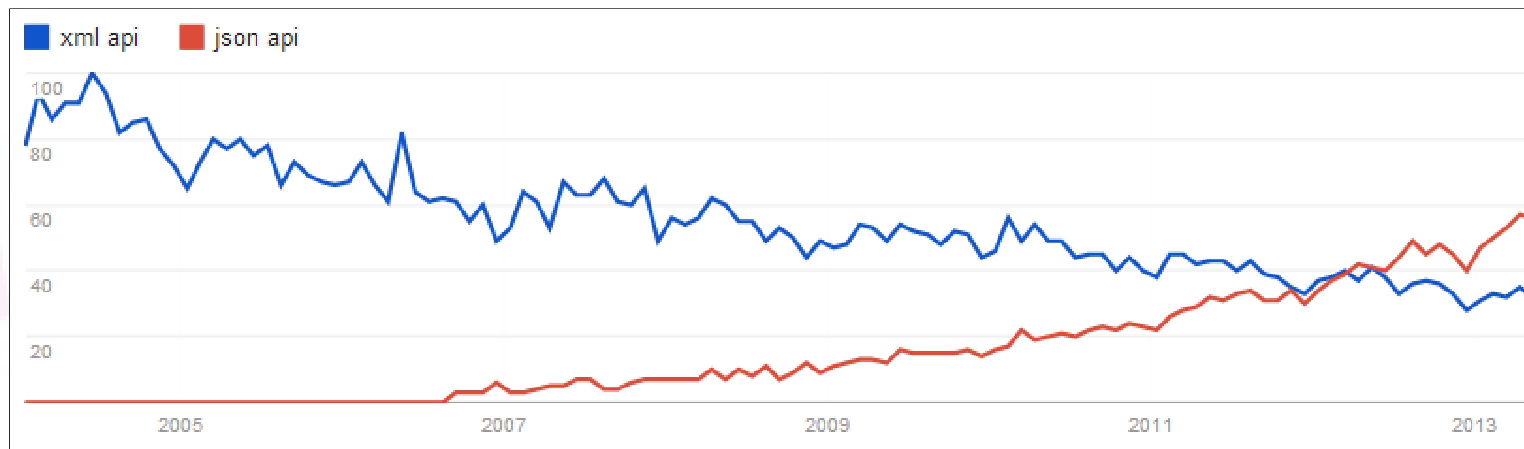
REST Best Practices (2/6)

- Might decide to use some parameter to limit “response-heavy” queries – **`GET /users?fields=id,name,desc`**
  - The Github API takes an interesting approach: collections return only the basic information (id, name, desc,...). If you need more → need to query the specific resource
- Versioning: it is important to **version** the API – the requested version can be given in the header (preferred) or as a parameter.
  - See how the GitHub REST API manages versioning: https://developer.github.com/v3/media/#request-specific-version
- Might use some **aliases** for common queries → **`GET /users/most_popular`**

lasaris

REST Best Practices (3/6)

- **Create** and **Update** methods should return the resource that has been created or modified
- Usage of **HATEOAS** is a design decision, in some cases it might add more overhead to what it is really necessary
- **JSON** is nowadays much more popular than XML in REST APIs*



* Pragmatic Best Practice about REST APIs
http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api

REST Best Practices (4/6)

- Some API prefer to always wrap **all** responses so that there is a standard way of returning data even in case of errors:

```
{
  "data" : {
    "id" : “1”,
    "name" : "Joseph"
  },
   "state" : {
    "name" : “OK”,
    "desc" : "no error"
  }
}
```

This depends on the case, as it introduces **overhead** for every response. Good idea is to return some structured information, e.g. validation errors

```
{
 "code" : 1024,
 "message" : "Validation Failed",
 "errors" : [
   {
    "field" : "first_name",
    "message" : "First name cannot be empty"
   },
   {
     "field" : "price_change",
     "message" : "Price cannot be changed by over 10%"
   }
 ]
}
```

REST Best Practices (5/6)

- When returning **paginated results**, you can use the link header:
  - https://developer.github.com/v3/#pagination
- It can be a good idea to think about limiting access to the API implementing some **limiting counter** returning **`429 Too Many Requests`**
  - https://developer.github.com/v3/#rate-limiting
- Implementing a limiter for access will also “force” clients to use **conditional requests**

lasaris

REST Best Practices (6/6)

- Most used return codes in REST APIs:

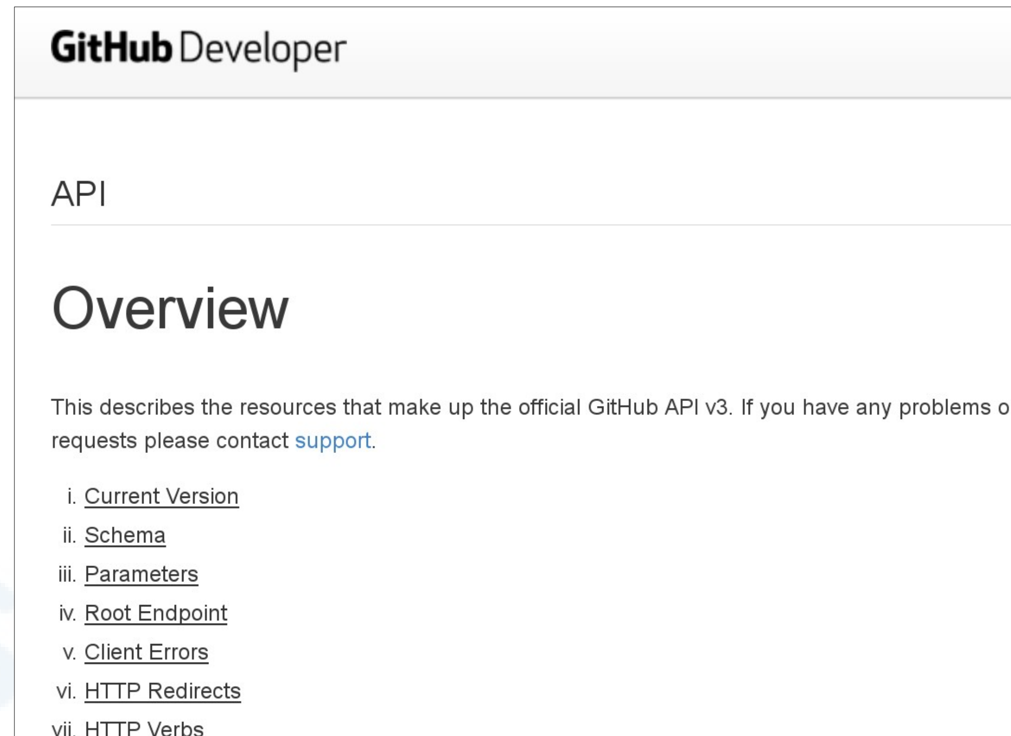
- **200 OK** → successful GET, PUT, PATCH, DELETE, POST (no creation)
- **201 Created** → After a POST (creation). Location header might give location of new resource
- **204 No Content** → Successful but no body (e.g DELETE)
- **304 Not Modified** → HTTP caching
- **400 Bad Request** → error in the request body
- **404 Not Found** → non-existent resource requested
- **409 Conflict** → a resource conflict, e.g. duplicate entities
- **410 Gone** → old API method?
- **415 Unsupported Media Type** → incorrect content type in part of the request
- **422 Unprocessable Entity** → Validation errors
- **429 Too Many Requests** → limiting requests

5xx HTTP codes are used to indicate some server-error – might decide in these cases to always return 500 Internal Server Error

lasanis

Github REST API

- Let's look at GitHub REST API
- https://developer.github.com/v3/
- Also a wrapper of such API: http://github-api.kohsuke.org



REST in Spring

A Spring REST Controller

Spring

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

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

or produces={MediaType.TEXT_PLAIN}

Multiple Representations

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

produces={MediaType.TEXT_PLAIN [, more-types]}	Specifies the type of data that is returned, for example, "text/plain"
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
    ```

Which is the order in which these are considered in Spring?

lasaris

Multiple Representations

- Content negotiation
  - ③ Accept header
    ```
    GET /customers
    Accept: application/json
    ```
  - ① URI-based
    ```
    GET /customers.json
    ```
  - ② parameter-based
    ```
    http://localhost/customers?type=json
    ```

Why is 'accept header' the last option?

Content Negotiation

- Example

```
@RestController
@RequestMapping(value=ApiUris.ROOT_URI_ORDERS,
consumes=MediaType.TEXT_PLAIN_VALUE)
public class CustomerController {

  @RequestMapping(method = RequestMethod.POST, consumes =
MediaType.TEXT_XML_VALUE, produces = MediaType.TEXT_XML_VALUE)
  public CustomerDTO createCustomer(NewCustomerDTO customer)
  {...}

}
```

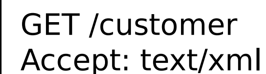
```
POST /customers
content-type: text/xml

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

lasaris

Content Negotiation

- Example

```
@RestController
@RequestMapping(value=ApiUris.ROOT_URI_ORDERS,
produces=MediaType.TEXT_PLAIN_VALUE)
public class CustomersController {

  @RequestMapping(method = RequestMethod.GET)
  public List<CustomerDTO> getCustomersPlain()
  {...}

  @RequestMapping(method = RequestMethod.GET, produces =
                                             MediaType.TEXT_XML_VALUE)
  public List<CustomerDTO> getCustomersXML()
  {...}
}
```



Content Negotiation

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.TE
   XT_PLAIN).mediaType("xml",MediaType.APPLICATION_XML).
   mediaType("json",MediaType.APPLICATION_JSON);
  }
}
```

We are favouring parameter based requests, ignoring accept headers

lasaris

Content Negotiation

Configuring the ObjectMapper

See http://docs.spring.io/spring-boot/docs/current/reference/html/howto-spring-mvc.html

```
@Bean
public MappingJackson2HttpMessageConverter customJackson2HttpMessageConverter() {
    MappingJackson2HttpMessageConverter jsonConverter = new
                                                MappingJackson2HttpMessageConverter();
    ObjectMapper objectMapper = new ObjectMapper();
    objectMapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
    objectMapper.disable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);
    jsonConverter.setObjectMapper(objectMapper);
    return jsonConverter;
}

@Override
public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
    converters.add(customJackson2HttpMessageConverter());
}
```

lasaris

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.

lasaris

Testing the REST API (1/2)

- Very often it is useful also for documentation (see later)
- Using org.springframework.test.web.servlet.MockMvc

```
mockMvc = standaloneSetup(productsController).setMessageConverters(new
MappingJackson2HttpMessageConverter()).build();

[...]

// mocking some facade/service operation
doReturn(Collections.unmodifiableList(this.createProducts())).when(
                 productFacade).getAllProducts();

mockMvc.perform(get(ApiUris.ROOT_URI_PRODUCTS))
                 .andExpect(status().isOk())
                 .andExpect(
                         content().contentType(MediaType.APPLICATION_JSON_VALUE));
```

lasaris

Testing the REST API (2/2)

- Using JSONPath to check the JSON request/response
- https://github.com/jayway/JsonPath

```
mockMvc.perform(get(ApiUris.ROOT_URI_ORDERS).param("status",
"ALL")).andDo(print())
                  .andExpect(status().isOk())
                  .andExpect(
                          content().contentType(MediaType.APPLICATION_JSON_VALUE))
                  .andExpect(jsonPath("$.[?(@.id==1)].state").value("DONE"));
```

Filtering expression, '@' stands for the current node

Managing Exceptions (1/5)

- Any unhandled exception will cause an HTTP 500 response
- However, you can annotate exceptions with @ResponseStatus to return the appropriate HTTP error code & message

```
@ResponseStatus(value=HttpStatus.NOT_FOUND,
                              reason="the resource was not found")
    public class ResourceNotFoundException extends RuntimeException{
        [...]
    }
```

lasaris

Managing Exceptions (2/5)

```
@ResponseStatus(value=HttpStatus.NOT_FOUND, reason="The customer was 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 = customersFacade.getCustomerById(id);
        if (customer == null) throw new CustomerNotFoundException(id);
   ....
}
```

lasaris

Managing Exceptions (3/5)

See org.springframework.http.HttpStatus

http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/http/HttpStatus.html

Enum Constants

Enum Constant and Description
ACCEPTED 202 Accepted.
ALREADY_REPORTED 208 Already Reported.
BAD_GATEWAY 502 Bad Gateway.
BAD_REQUEST 400 Bad Request.
BANDWIDTH_LIMIT_EXCEEDED 509 Bandwidth Limit Exceeded
CHECKPOINT 103 Checkpoint.
CONFLICT 409 Conflict.
CONTINUE 100 Continue.

Managing Exceptions (4/5)

- Methods annotated with @ExceptionHandler are handling exceptions
- You do not need to add @ResponseStatus to the Exceptions
- Gives you more freedom in returning a custom error data structure

```
@RestController
public class MyController {
...
  @ExceptionHandler
  @ResponseStatus(HttpStatus.UNPROCESSABLE_ENTITY)
  @ResponseBody
  ApiError handleException(ResourceAlreadyExistingException ex) {
      ApiError apiError = new ApiError();
      apiError.setErrors(Arrays.asList("the requested resource already
                                       exists"));
      return apiError;
  }
}
```

Managing Exceptions (5/5)

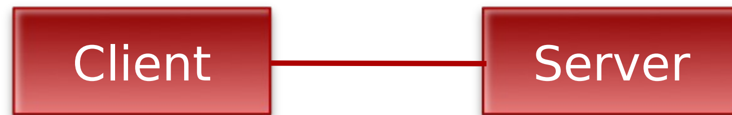
- Another way is to have a global advice using @ControllerAdvice that will manage exceptions for all controllers

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

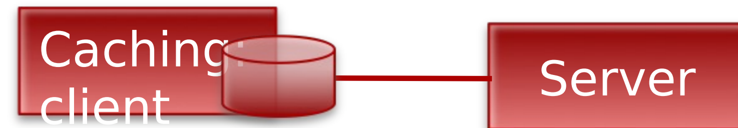
lasaris

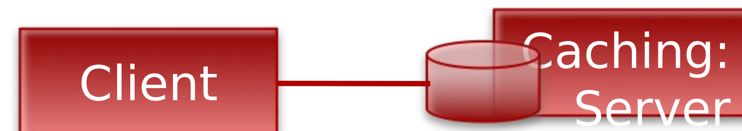
Caching

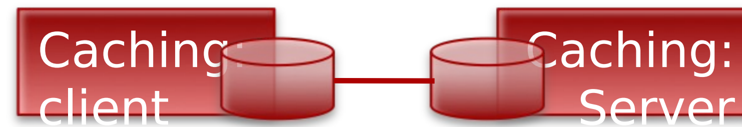
Basic setup



Caching options







lasaris

Caching Example 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";
 }
```

lasaris

Caching Example in Spring

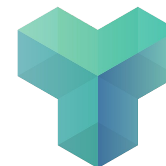
```
> curl -X GET -i http://localhost:8080/eshop-rest/users/1

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

> curl -i -X GET http://localhost:8080/eshop-rest/users/1
--header 'If-None-Match: "3242771"'

-Match: "3242771"'
HTTP/1.1 304 Not Modified
Server: Apache-Coyote/1.1
ETag: "3242771"
```

Documentation

- Several ways to document a Spring REST API
- More design-oriented
  - Example, Apiary https://apiary.io/
- By test invocation
  - REST Docs, http://projects.spring.io/spring-restdocs/
- By annotating controller methods
  - Swagger, http://swagger.io

lasaris

Documentation - Apiary



API Blueprint Syntax Tutorial

✔ Valid API Blueprint Preview On Save & Publish

```
1  FORMAT: 1A
2  HOST: http://localhost/pa165/api/v1
3
4  # PA165 eShop
5
6  PA165 eShop is a REST API to access operations performed on the project created as sample for the cours
7  It allows to access and interact with information related to Products, Order, Prices.
8
9  # PA165 eShop API Root [/]
10
11 This resource does not have any attributes: it offers the initial
12 list of API resources available from the API in JSON form, it is advised to follow the "url" link value
13 to keep your client decoupled from implementation details.
14
15 ## Retrieve the API Entry Points / [GET]
16
17 + Response 200 (application/json)
18
19         {
20             "products_url": "/products",
21             "orders_url": "/orders",
22             "users_url": "/users"
23
24         }
25
26 ## Group Product
27
28 Resources related to products in the API.
29 ** TODO: include image change for product?**
30
31 ## Product [/products/{product_id}]
32
33 The returned Product object has the following attributes:
34
35 + id - product id
36 + name - product name
37 + current_price - current price: value and currency
38 + image_url - link to the url for the product image
39 + categories - list of categories it belongs to
40 + price hist - history of prices
```

Introduction

PA165 eShop

INTRODUCTION

PA165 eShop is a REST API to access operations performed on the project created as sample for the course PA165. It allows to access and interact with information related to Products, Order, Prices.

REFERENCE

PA165 eShop API Root

This resource does not have any attributes: it offers the initial list of API resources available from the API in JSON form, it is advised to follow the "url" link values to retrieve the resources instead of constructing your own URLs, to keep your client decoupled from implementation details.

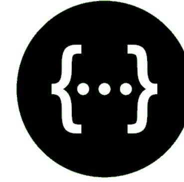
lasaris

Documentation – REST Docs

- Official Spring project
- Uses expected testing behaviour to describe the API

```
In a test class:
@Rule
public final RestDocumentation restDocumentation = new
RestDocumentation("build/generated-snippets");

@Before
public void setUp() {
    this.mockMvc = MockMvcBuilders.webAppContextSetup(this.context)
            .apply(documentationConfiguration(this.restDocumentation))
            .build();
}

Later on in a @Test method:
this.mockMvc.perform(get("/customers").accept(MediaType.APPLICATION_JSON))
    .andExpect(status().isOk())
    .andDo(document("customers"));
```

Documentation – Swagger



- Very popular documentation project for REST API, not officially endorsed by the Spring community
- Based on additional annotations on the controllers to describe the API

```
@RestController
@Api(value=ApiUris.ROOT_URI_CUSTOMERS, description="All operations related to
                                                   customer resources")
@RequestMapping(ApiUris.ROOT_URI_CUSTOMERS)
public class CustomersController {

  @ApiOperation(value = "Get information about one customer", note="additional notes")
  @RequestMapping(value="/{id}", method=RequestMethod.GET)
  public String getCustomer(@ApiParam(name="id", value="the id of the customer resource
                 to be retrieved", required=true) @PathVariable String productid) {
    ....
  }
 ...
}
```

See https://github.com/swagger-api/swagger-core/wiki/Annotations-1.5.X

References

- Roy Fielding PhD Thesis:
  https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf
- Review of REST API documentation tools:
  https://www.opencredo.com/2015/07/28/rest-api-tooling-review/
- Richardson Maturity model from Martin Fowler's website:
  See http://martinfowler.com/articles/richardsonMaturityModel.html
- Webber, Jim, Savas Parastatidis, and Ian Robinson. REST in practice:
  Hypermedia and systems architecture. " O'Reilly Media, Inc.", 2010.

- Pragmatic Best Practice about REST APIs
  http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api
- RESTful best practices
  http://www.restapitutorial.com/media/RESTful_Best_Practices-v1_1.pdf