Java

Paulo Baltarejo Sousa and Joaquim Peixoto dos Santos

{pbs,jpe}@isep.ipp.pt

2020

Server-side Programming

RESTful Web Services

Disclaimer

Material and Slides

Some of the material/slides are adapted from various:

- Presentations found on the internet;
- Books;
- Web sites;
- ...

Outline

RESTful web service

What is REST?

- While REST stands for **Representational State Transfer**, which is an architectural style for networked hypermedia applications.
- It is primarily used **to build web services that are lightweight, maintainable, and scalable**.
- **A web service based on REST** is called **a RESTful service**.
- REST is not dependent on any protocol, but almost every **RESTful service uses HTTP** as its underlying protocol.

Features of a RESTful Services

- Every system **uses resources**.
  - These resources can be pictures, video files, web pages, business information, or anything that can be represented in a computer-based system.
- The purpose of a **web service is to provide access to these resources**.
- Properties and Features:
  - Representations
  - Messages
  - Addressing Resources
  - Uniform interface
  - Stateless

Properties and Features: Representations (I)

- **The focus of a RESTful service is on resources** and how to provide access to these resources.
- The first thing to do is **identify the resources**.
- How to represent these resources in our system.
  - You can use any format for representing the resources, as REST does not put a restriction on the format of a representation.

```
<employees>
  <employee>
    <id>1</id>
    <firstName>John</firstName>
    <lastName>Doe</lastName>
  </employee>
  <employee>
    <id>2</id>
    <firstName>Anna</firstName>
    <lastName>Smith</lastName>
  </employee>
</employees>
```

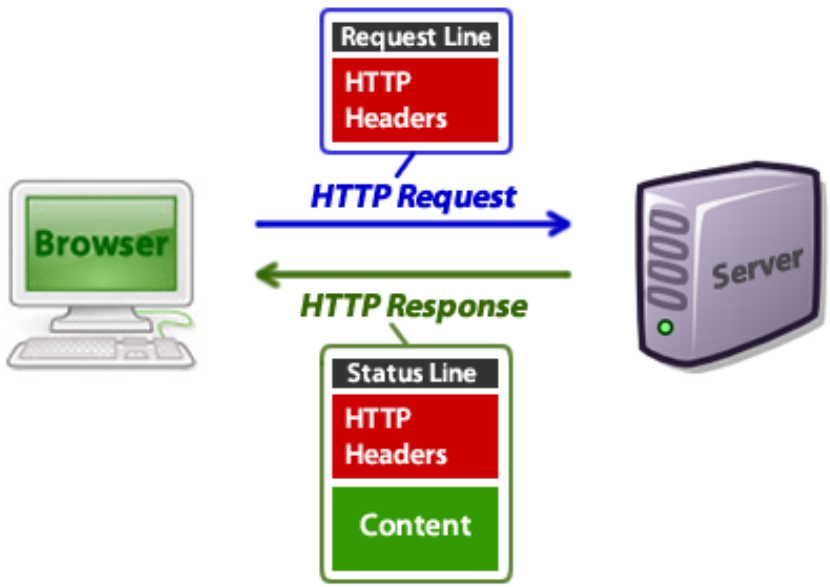
```
{"employees":
  [
    {"id": 1, "firstName":"John", "lastName":"Doe" },
    {"id": 2, "firstName":"Anna", "lastName":"Smith" },
  ]
}
```

Properties and Features: Representations (II)

- Both client and server should be able to **comprehend this format of representation**.
- A representation should be able to **completely represent a resource**.

Properties and Features: Messages (I)

- **The client and service talk to each other via messages.**
- Clients send a request to the server, and the server replies with a response.



- Apart from the actual data, these messages also contain some metadata about the message.

Properties and Features: Messages (II)

- An HTTP request has the format

  - `<VERB>` is one of the HTTP methods like GET, PUT, POST, DELETE, OPTIONS, etc
  - `<URI>` is the Uniform Resource Identifier (URI) of the resource on which the operation is going to be performed
  - `<HTTP Version>` is the version of HTTP, generally "HTTP v1.1" .
  - `<Request Header>` contains the metadata as a collection of `<key-value>` pairs of headers and their values.
    - These settings contain information about the message and its sender like client type, the formats client supports, format type of the message body, cache settings for the response, and a lot more information.

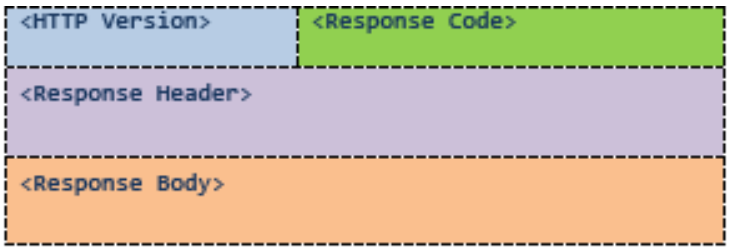
Properties and Features: Messages (III)

```
POST http://MyService/Person/ HTTP/1.1

Host: MyService
Content-Type: text/xml; charset=utf-8
Content-Length: 123

<?xml version="1.0" encoding="utf-8"?>
<Person>
  <ID>1</ID>
  <Name>M Vaqqas</Name>
  <Email>m.vaqqas@gmail.com</Email>
  <Country>India</Country>
</Person>
```

- **POST** method, which is followed by the **URI** and the **HTTP version**.
- It contains some **request headers**.
  - `Host` is the address of the server.
  - `Content-Type` tells about the type of contents in the message body.
  - `Content-Length` is the length of the data in message body.

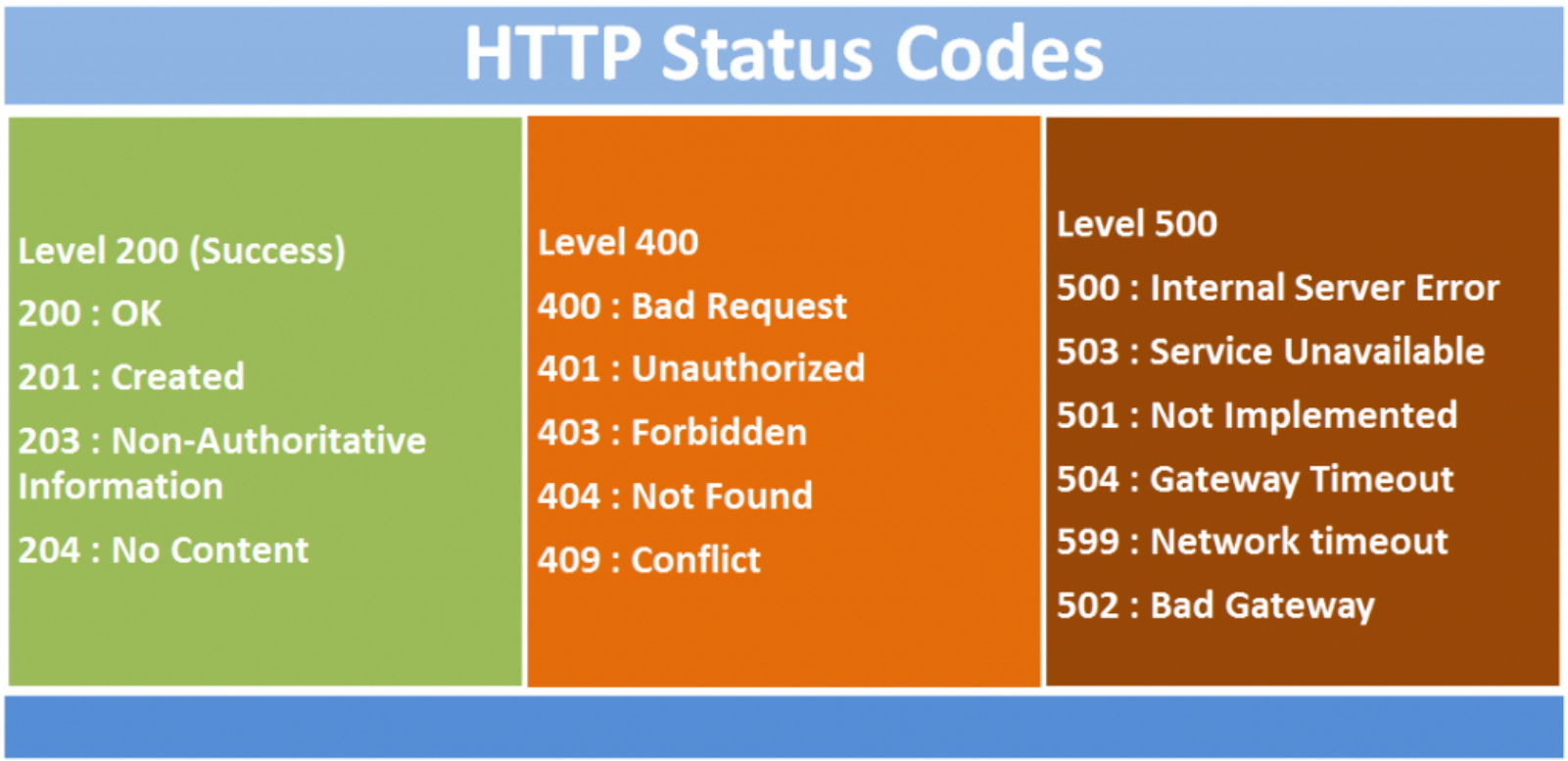
Properties and Features: Messages (IV)

- An HTTP response has the format



  - The server returns `<response code>`, which contains the status of the request.
    - This response code is generally the 3-digit HTTP status code.
  - `<Response Header>` contains the metadata and settings about the response message.
  - `<Response Body>` contains the representation if the request was successful.

HTTP status code



Properties and Features: Messages (V)

```
HTTP/1.1 200 OK

Date: Sat, 23 Aug 2014 18:31:04 GMT
Server: Apache/2
Last-Modified: Wed, 01 Sep 2004 13:24:52 GMT
Accept-Ranges: bytes
Content-Length: 32859
...
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/
      xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns='http://www.w3.org/1999/xhtml'>
<head><title>Hypertext Transfer Protocol -- HTTP/1.1</title></head>
<body>
...
```

- The **response code** `200 OK` means that everything went well
- **Response headers** contains some information such as `Content-Length` and `Content-Type` of the response body

Properties and Features: Messages (VI)

- **Response body** contains a valid representation of the requested resource.
  - In this case, the representation is an HTML document that is declared by `Content-Type` header in the Response Header.

Properties and Features: Addressing Resources (I)

- REST requires each **resource to have at least one URI**.
- A RESTful service uses a **directory hierarchy like human readable URIs** to address its resources.
  - The job of a URI is to identify a resource or a collection of resources.
  - It is a string of characters that unambiguously identifies a particular resource.
    - For instance `http://MyService/users/1` addresses the `user` resource.
- The actual **operation is determined by an HTTP method/verb**.
  - This enables us to **call the same URI with different HTTP verbs** to perform different operations.
    - `GET http://MyService/users/1`
    - `DELETE http://MyService/users/1`
- The **URI should not say anything about the operation or action**.

Properties and Features: Addressing Resources (II)

- Recommendations for well-structured URIs:
  - Use **plural nouns for naming your resources**.
  - **Avoid using spaces** as they create confusion.
    - Use an _ (underscore) or - (hyphen) instead.
  - A URI is **case insensitive**.
  - **Avoid verbs for your resource names**.
    - For example, a RESTful service should not have the URIs `http://MyService/FetcthUser/1` or `http://MyService/DeleteUser?id=1`.

Properties and Features: Uniform Interface (I)

- RESTful systems should have a uniform interface.

Method	Operation performed on server	Quality
GET	Read a resource.	Safe
PUT	Insert a new resource or update if the resource already exists.	Idempotent
POST	Insert a new resource. Also can be used to update an existing resource.	No safe, no idempotent
DELETE	Delete a resource.	Idempotent

- A **Safe** operation is an operation that does not have any effect on the original value of the resource.
- An **Idempotent** operation is an operation that gives the same result no matter how many times you perform it.
  - The problem with DELETE, which if successful would normally return a 200 (OK) or 204 (No Content), will often return a 404 (Not Found) on subsequent calls. However, the state on the server is the same after each DELETE call, but the response is different.

Properties and Features: Uniform Interface (II)

- The key difference between PUT and POST is that PUT is idempotent while POST is not.
  - No matter how many times you send a PUT request, the results will be same.
  - POST is not an idempotent method. Making a POST multiple times may result in multiple resources getting created on the server.

Request	**Operation**
`PUT http://MyService/users/1`	Update the existing resource with `UserID=1`
`POST http://MyService/users/`	Insert a new user every time this request is made and generate a new user.

Properties and Features: Statelessness

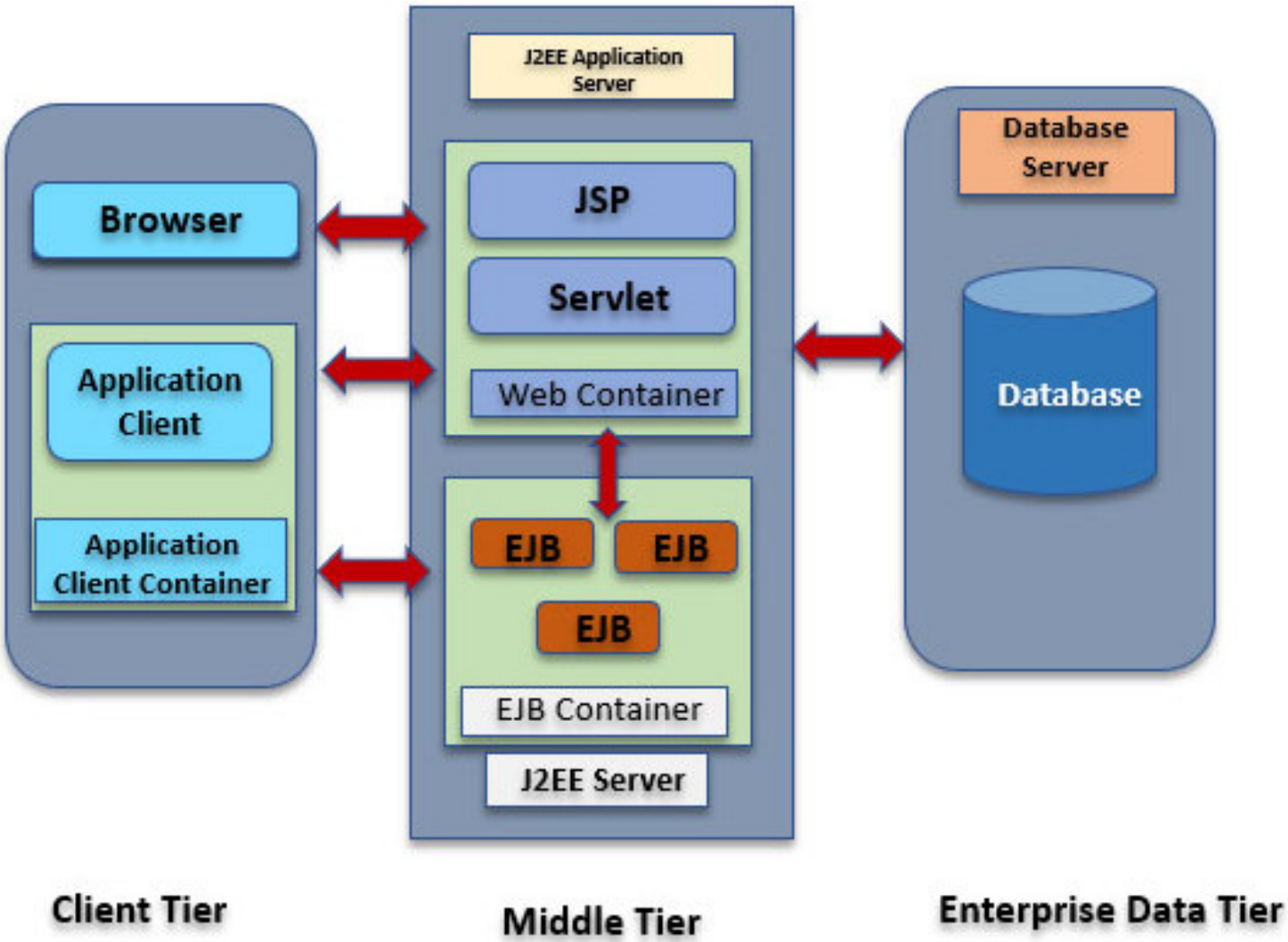
- A **RESTful service is stateless** and does not maintain the application state for any client.
- **A request cannot be dependent on a past request** and a service treats each request independently.
- A stateless design looks like so:
  - Request1: `GET http://MyService/users/1`
  - Request2: `GET http://MyService/users/2`
    - Each of these requests can be treated separately.
- A stateful design, on the other hand, looks like so:
  - Request1: `GET http://MyService/users/1`
  - Request2: `GET http://MyService/NextUser`
    - To process the Request2, **the server needs to remember the last** `UserID` **that the client fetched**.
    - In other words, the server needs to remember the current state - otherwise **Request2 cannot be processed**.

Java 2 Enterprise Edition (J2EE)

What is J2EE? (I)

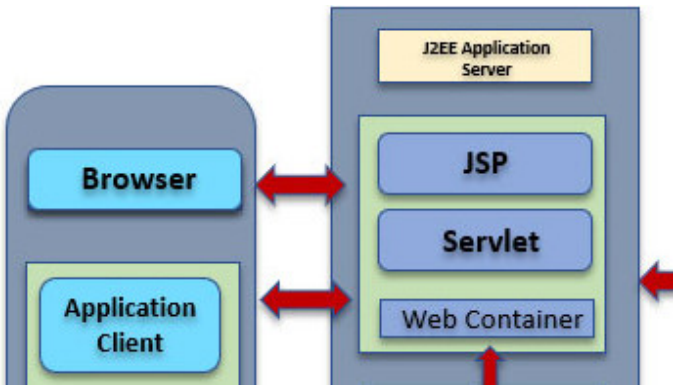
- Java 2 Enterprise Edition (J2EE) **is a platform that provides an environment to develop enterprise applications** using multi-tier architecture.
- It uses, basically, three tiers:
  - **Client Tier**
    - It consists of user programs that interact with the user for request and response.
    - A client can be a web browser, standalone application or server that runs on a different machine.
  - **Middle Tier**
    - It usually contains enterprise beans and web services that distribute business logic for the applications.
  - **Enterprise Data Tier**
    - It consists of database servers, enterprise resource planning systems and other data sources

What is J2EE? (II)



Web container

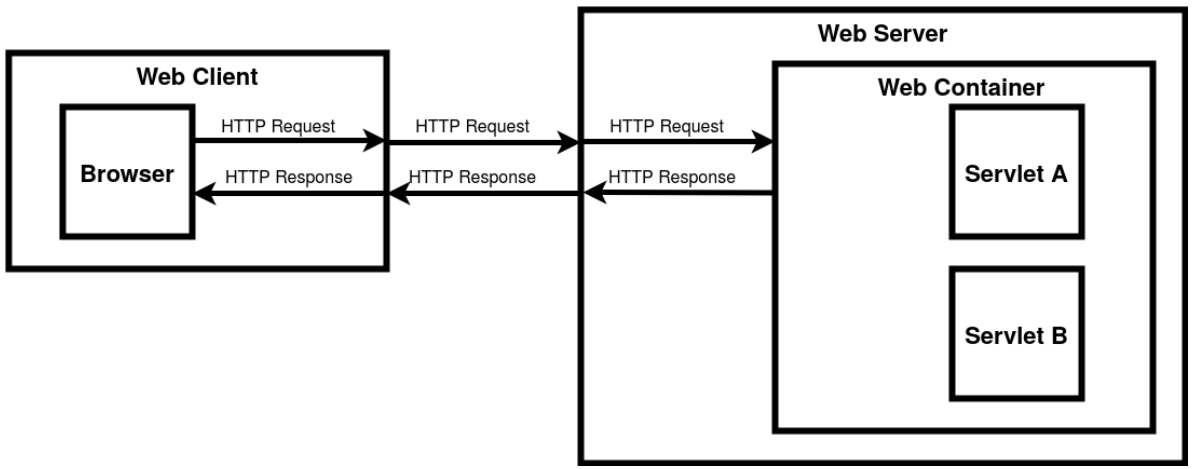
- A **web container** is built on top of the J2EE platform, and it **implements the Servlet API and the services required to process HTTP requests**.



- It is a **runtime environment for** Java Server Pages (JSPs) files and and **Servlets**.
- It is responsible for managing the lifecycle of servlets, mapping a URL (Uniform Resource Locator) to a particular servlet.

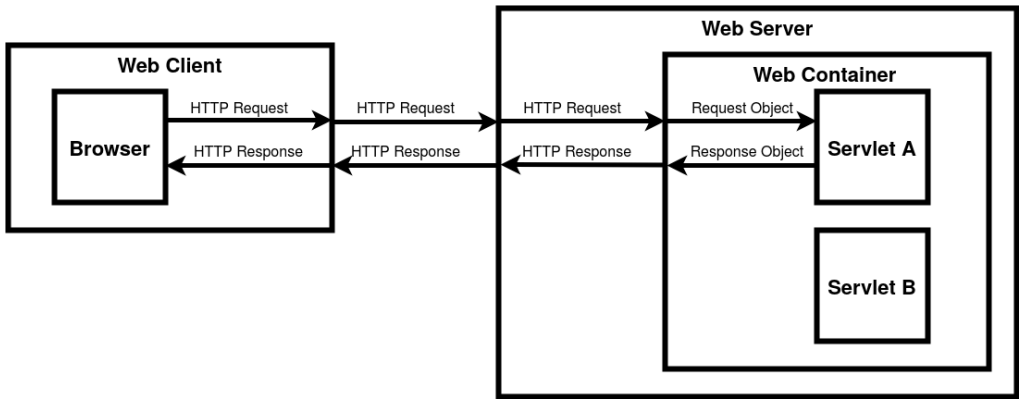
Servlet (I)

- Servlet is basically a **Java program that runs in a web container** on the web server.



- **Web container activates the Servlet that matches the requested URL**

Servlet (II)

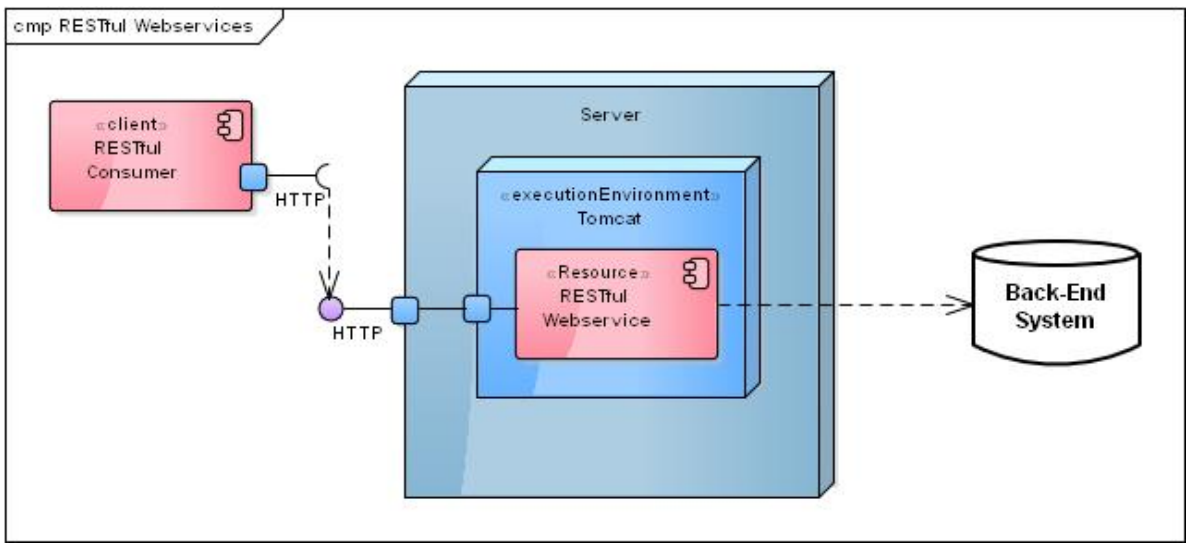


- The Web Client Browser sends an HTTP Request to the Web Server
- Web Server forwards it to Web Container
- Web Container forwards it to the Servlet in form of the Request Object
- Servlet generates Response Object and sends it to the Web Container
- Web Container converts it into equivalent HTTP Response to the Web Server
- The Web Server sends it back to the Web Client Browser.

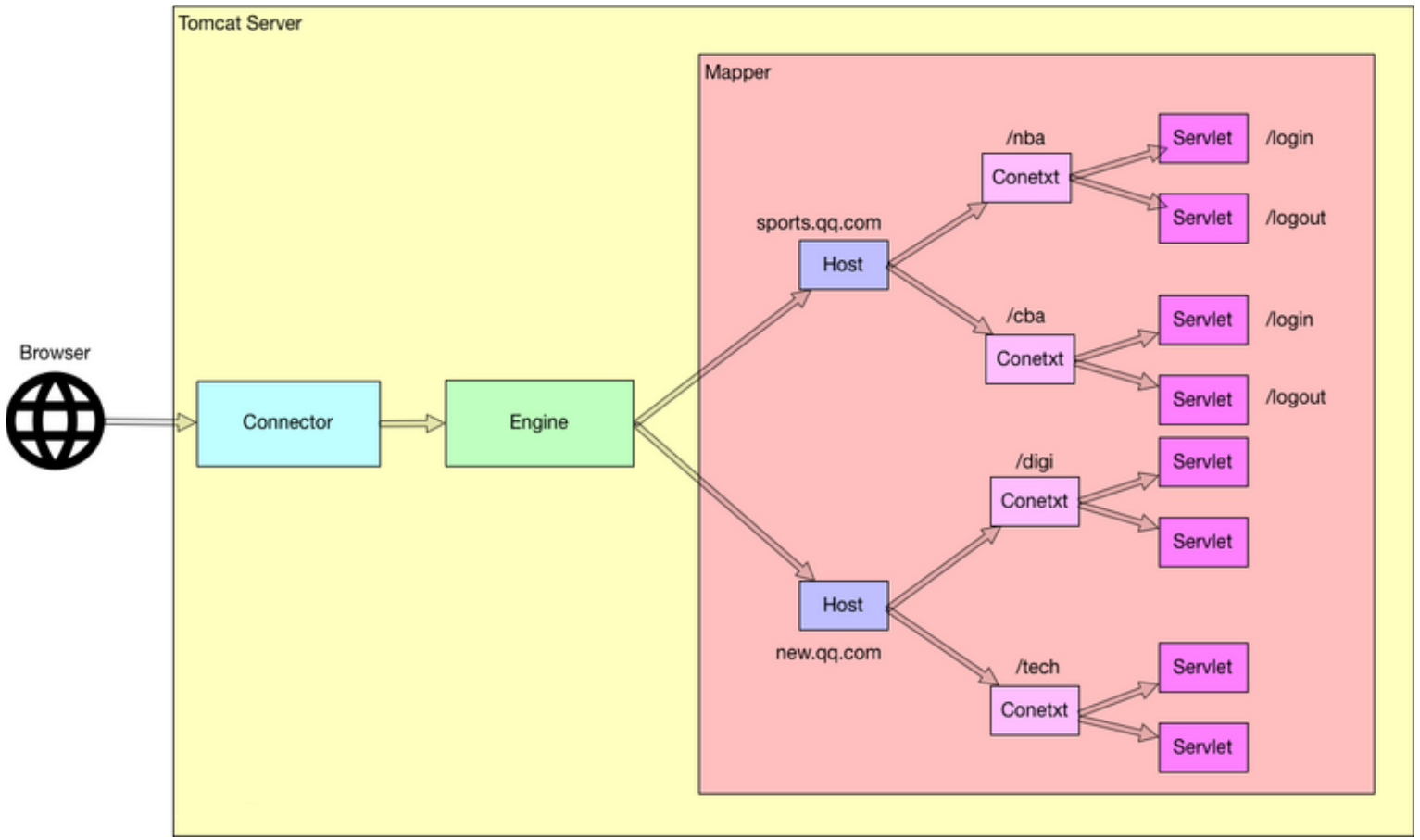
Spring

Spring Framework

- It **provides the infrastructure to develop full J2EE Applications**.
  - It provides everything you need to embrace the Java language in an enterprise environment and with the flexibility to create many kinds of architectures depending on an application's needs
- It provides Apache Tomcat, which is a long-lived, open source **web container**



Tomcat Architecture (I)



Tomcat Architecture (II)

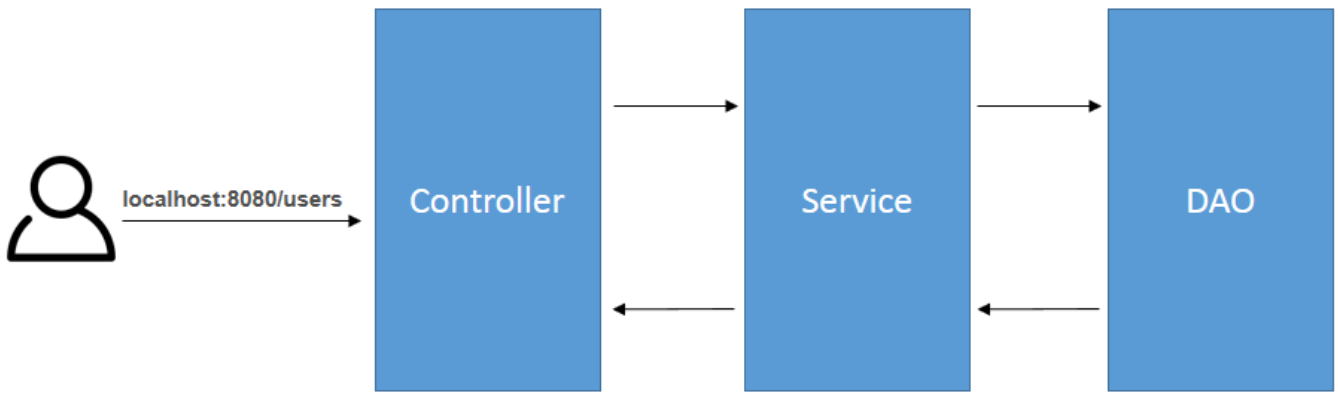
- Connector
  - Manages the HTTP requests/responses
- Engine
  - Perform some work on HTTP requests as well as on HTTP responses.
- Host
  - Represents a virtual host.
  - Each host has its own domain name.
- Context
  - Represents an application container.
  - Each application can configure multiple servlets.

Coding Servlet (I)

Flow

- `DispatcherServlet` is the front `Controller`.

- The process from a request to response



- `DispatcherServlet` it is provided by Spring framework

Controller

```
@RestController
@RequestMapping(value = "/api")
public class UserController {

}
```

- `UserController` **is a normal Java class with a set of annotations** (annotations are preceded by `@`).
  - `@RestController` is an annotation for creating Restful controllers
  - `@RequestMapping`
    - It is used to map HTTP requests to controllers (class)
    - `value`: the primary mapping expressed by this annotation.
- It is the **entry point of the Servlet**

Controller: GET

```
@RestController
@RequestMapping(value = "/api")
public class UserController {
  @RequestMapping(value ="/users", method = RequestMethod.GET)
  public ResponseEntity<Object> getUsers() {
    return new ResponseEntity<>("Mock Data: List of users", HttpStatus.OK);
  }
}
```

- `@RequestMapping`
  - It is used to map HTTP requests to controller's method.
    - `method`: the HTTP request methods to map to: GET, POST, PUT, and DELETE.
- `ResponseEntity` represents the whole HTTP response: status code, headers, and body.
- HTTP request

```
GET /api/users HTTP/1.1
Host: localhost:8080
...
```

Controller: GET with parameter (I)

```
@RestController
@RequestMapping(value = "/api")
public class UserController {
...

  @RequestMapping(value= "/users/{id}", method = RequestMethod.GET)
  public ResponseEntity<Object> getUser(@PathVariable("id") int id) {
    return new ResponseEntity<>("Mock Data: user: "+id, HttpStatus.OK);
  }

}
```

- `@PathVariable` is a Spring annotation which indicates that a method parameter should be bound to a URI template variable.
- HTTP request

```
GET /api/users/1 HTTP/1.1
Host: localhost:8080
...
```

Controller: GET with parameter (II)

```
@RestController
@RequestMapping(value = "/api")
public class UserController {
...

  @RequestMapping(value= "/users/{name}", method = RequestMethod.GET)
  public ResponseEntity<Object> getUser1(@PathVariable("name") String name) {
    return new ResponseEntity<>("Mock Data: user: "+name, HttpStatus.OK);
  }

}
```

- HTTP request

```
GET /api/users/maria HTTP/1.1
Host: localhost:8080
...
```

Controller: GET with parameter (III)

```
@RestController
@RequestMapping(value = "/api")
public class UserController {
...

  @RequestMapping(value= "/users/{name}/{email}", method = RequestMethod.GET)
  public ResponseEntity<Object> getUser2(@PathVariable("name") String name,
        @PathVariable("email") String email) {
    return new ResponseEntity<>("Mock Data: user: "+name+":"+email, HttpStatus.OK);
  }

}
```

- HTTP request

```
GET /api/users/maria/maria@mail.com HTTP/1.1
Host: localhost:8080
...
```

Controller: DELETE

```
@RestController
@RequestMapping(value = "/api")
public class UserController {
...

  @RequestMapping(value= "/users/{id}", method = RequestMethod.DELETE)
  public ResponseEntity<Object> deleteUser(@PathVariable("id") int id) {
    return new ResponseEntity<>(HttpStatus.OK);
  }

}
```

- HTTP request

```
DELETE /api/users/1 HTTP/1.1
Host: localhost:8080
...
```

Controller: POST

```
@RestController
@RequestMapping(value = "/api")
public class UserController {
...

  @RequestMapping(value= "/users", method = RequestMethod.POST)
  public ResponseEntity<Object> addUser(@RequestBody String name) {
    return new ResponseEntity<>(HttpStatus.CREATED);
  }

}
```

- @RequestBody maps the HttpRequest body to a method parameter
- HTTP request

```
POST /api/users HTTP/1.1
Host: localhost:8080
...

Maria
```

Controller: PUT

```
@RestController
@RequestMapping(value = "/api")
public class UserController {
...

  @RequestMapping(value= "/users/{id}", method = RequestMethod.PUT)
  public ResponseEntity<Object> updateUser(@PathVariable("id") int id, @RequestBody
        String name) {
    return new ResponseEntity<>(HttpStatus.OK);
  }

}
```

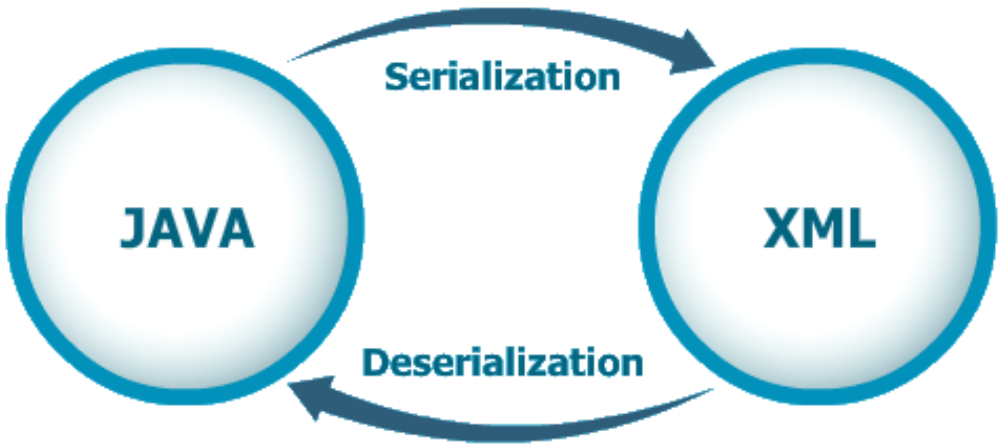
- HTTP request

```
PUT /api/users/1 HTTP/1.1
Host: localhost:8080
...

Maria
```

Transferring Data Format

Serialization and Deserialization

- **Serialization**
  - It is a mechanism of converting the state of an object into a byte stream (for instance, an XML document).
- **Deserialization**
  - It is the reverse process where the byte stream is used to recreate the actual Java object in memory.



Jackson Library

- Jackson library provides **an API for serializing and deserializing XML** (and other formats, like JSON) data to Java objects and vice-versa.
- It is based on annotations

Jackson annotations (I)

```
@JsonPropertyOrder({"id", "name", "email"})
@JacksonXmlRootElement(localName = "utilizador")
public class User {
  @JacksonXmlProperty(localName = "identificador")
  private int id;
  @JacksonXmlProperty(localName = "nome")
  private String name;
  @JacksonXmlProperty(localName = "email")
  private String email;

  public User() {
  }

  public int getId() {...}
  public void setId(int id) {...}
  public String getName(){...}
  public void setName(String name) {...}
  public String getEmail() {...}
  public void setEmail(String email) {...}
}
```

Jackson annotations (II)

- `@JsonPropertyOrder` annotation is used to specify the ordering of the serialized properties (class' attributes).
- `@JacksonXmlRootElement` annotation can be used to define name of root element used for the root-level object when serialized, which normally uses name of the type (class).
- `@JacksonXmlProperty` annotation can be used to provide XML-specific configuration for properties, class' attributes.

Jackson annotations (III)

```
@JacksonXmlRootElement(localName = "utilizadores")
public class UserList {
  @JacksonXmlProperty(localName = "descricao")
  private String description;
  @JacksonXmlElementWrapper(useWrapping = false)
  @JacksonXmlProperty(localName = "utilizador")
  private ArrayList<User> users;

  public UserList() {
  }
  public String getDescription() {...}
  public void setDescription(String description) {...}
  public ArrayList<User> getUsers() {...}
  public void setUsers(ArrayList<User> users) {...}
}
```

- `@JacksonXmlElementWrapper` annotation allows specifying XML element to use for wrapping ArrayList attributes

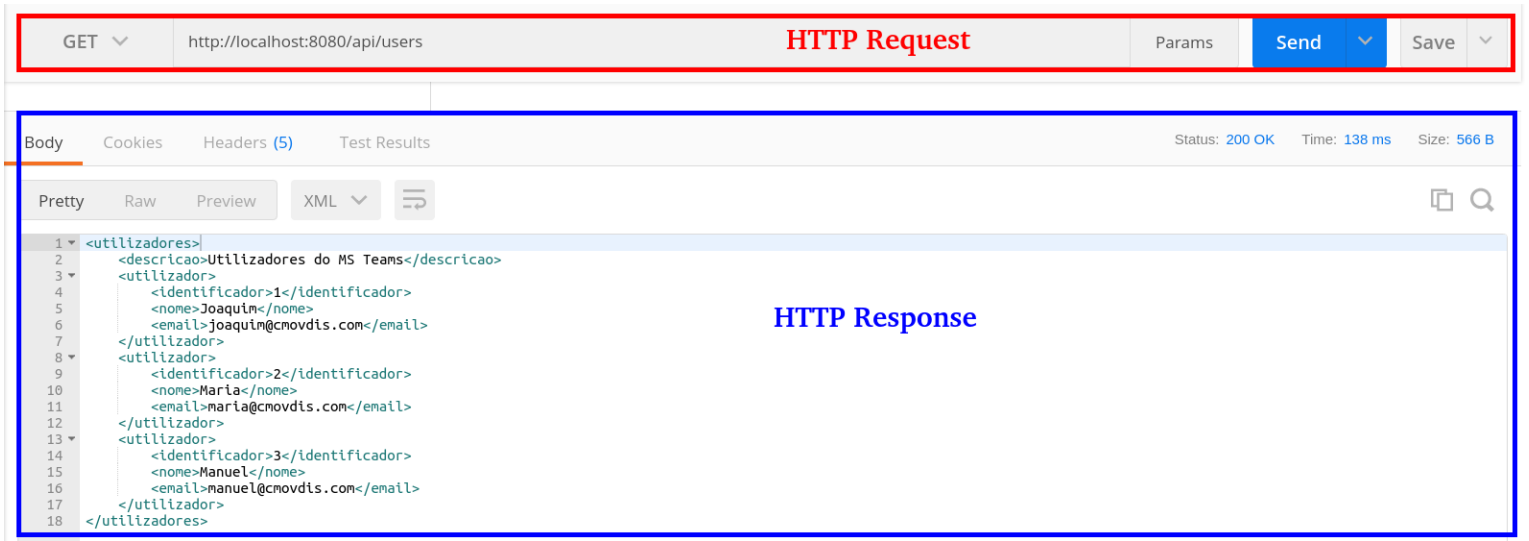
Coding Servlet (II)

Controller: GET producing XML (I)

```
@RestController
@RequestMapping(value = "/api")
public class UserController {
  ...
  @RequestMapping(value= "/users", method = RequestMethod.GET, produces = MediaType.
        APPLICATION_XML_VALUE)
  public ResponseEntity<Object> getUsers() {
    UserList list = getList();
    return new ResponseEntity<>(list, HttpStatus.OK);
  }
...
}
```

- produces narrows the primary mapping by media types that can be produced by the mapped handler.

Controller: GET producing XML (II)

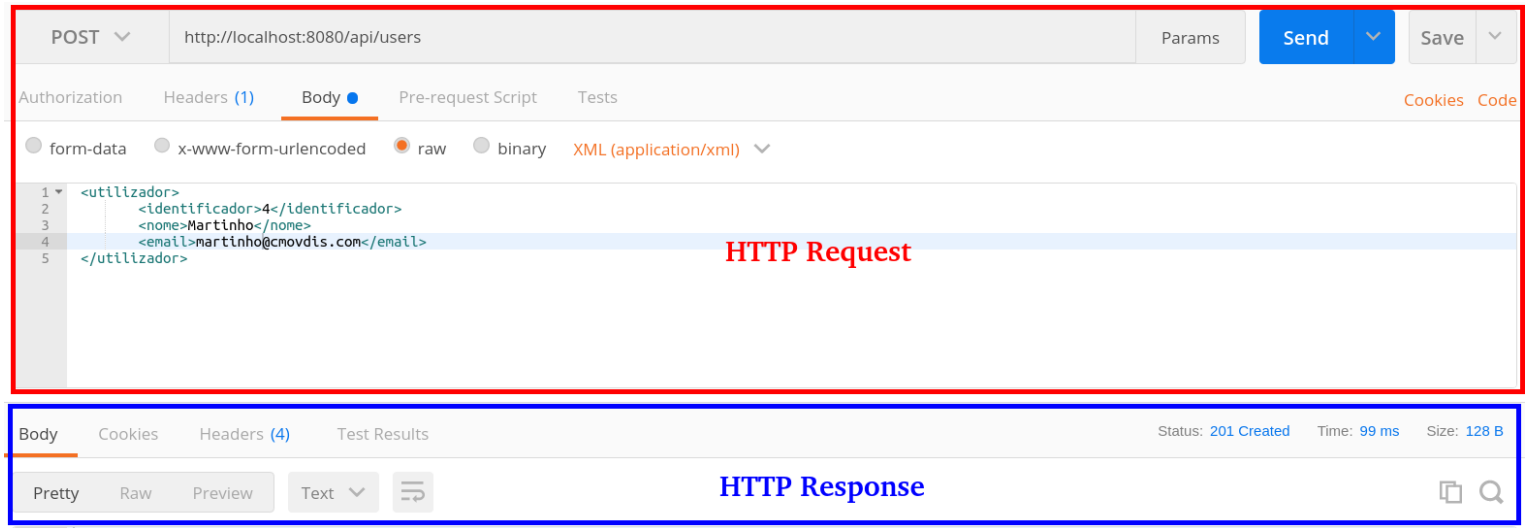


Controller: POST consuming XML (I)

```
@RestController
@RequestMapping(value = "/api")
public class UserController {
...
  @RequestMapping(value= "/users", method = RequestMethod.POST, consumes = MediaType.
        APPLICATION_XML_VALUE)
  public ResponseEntity<Object> addUser(@RequestBody User user) {
    addUser2List(user);
    return new ResponseEntity<>(HttpStatus.CREATED);
  }
...
}
```

- consumes narrows the primary mapping by media types that can be consumed by the mapped handler.

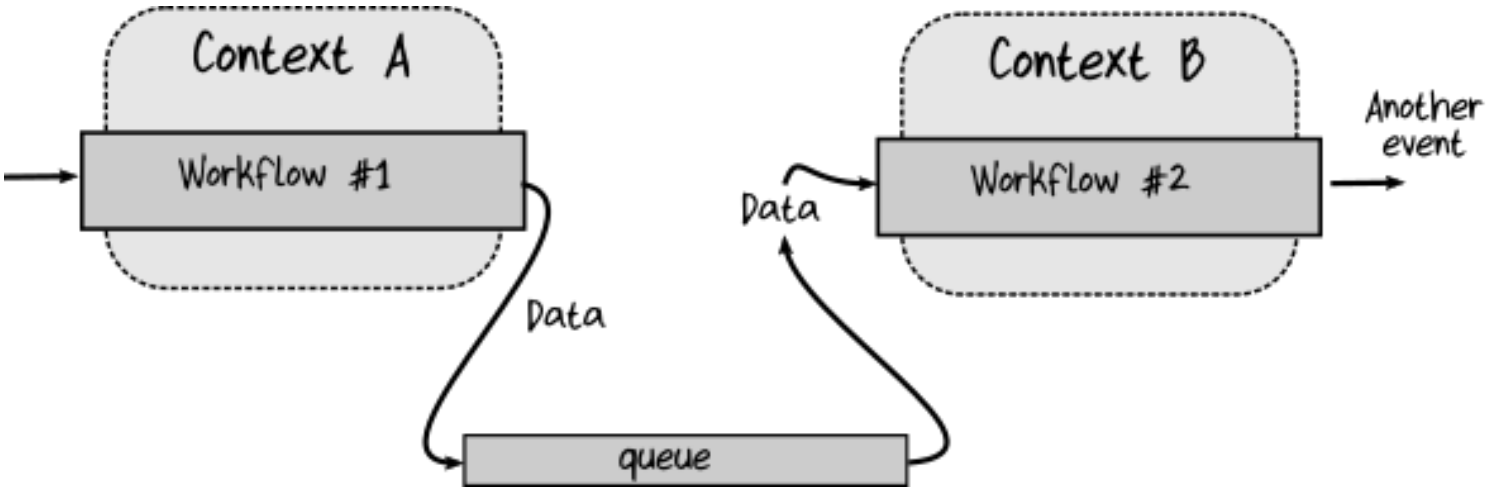
Controller: POST consuming XML (II)



Data Transfer Object (DTO)

Transferring Data

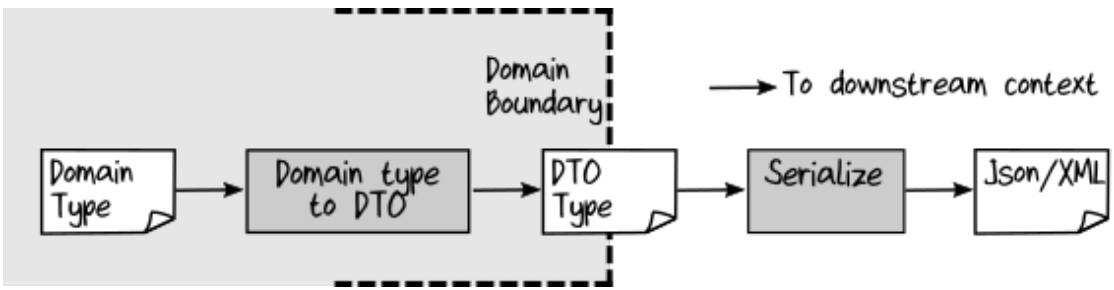
- Domain model must be known by other applications?



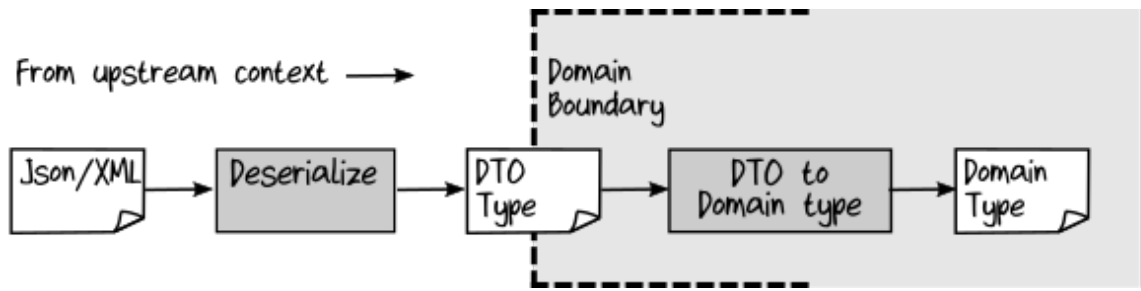
- (Answer) No. Domain model must be protected, hidden, unknown and ... from outside.
  - **Domain model is a secret**.
- But, it is required a shared (known) communication format?
  - (Answer) DTOs. **DTOs form a kind of contract** between bounded contexts.

DTOs as Contracts Between Bounded Contexts

- At the boundary of the upstream context then, the domain objects are converted into DTOs, which are in turn serialized into JSON, XML format:



- At the downstream context, the process is repeated in the other direction: the JSON or XML is deserialized into a DTO, which in turn is converted into a domain object:



What are DTOs?

- DTOs as in the **simple objects that carry data**, with no functionality at all.
- The difference between DTOs and domain (business) objects is that a **DTO does not have any behavior** except for serialization and deserialization of its own data.

Class for DTOs

- A DTO class must have:
  - Constructor with no parameter
  - Getters and setters for all attributes
- A DTO class cannot have:
  - Any business logic

```
public class PessoaDTO {
  private long nif;
  private String nome;
  private DataDTO nascimento;
  public PessoaDTO() {
  }
  public long getNif() {...}
  public void setNif(long nif) {...}
  public String getNome() {...}
  public void setNome(String nome)
        {...}
  public DataDTO getNascimento() {...}
  public void setNascimento(DataDTO
        nascimento) {...}
}
```

Bibliography

Resources

- "Professional Java Development with the Spring Framework" by Rod Johnson et al. John Wiley & Sons, 2005.
- "Expert One-on-One J2EE Development without EJB" by Rod Johnson and Juergen Hoeller. Wiley Publishing, Inc., 2004.
- `https://docs.oracle.com/javaee/6/tutorial/doc/giepu.html`
- `https://jersey.github.io/`
- `https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html`
- `https://github.com/FasterXML/jackson-dataformat-xml/wiki/Jackson-XML-annotations`