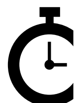
REpresentational State Transfer (REST)

PID_00275868

Silvia Llorente Viejo

Temps mínim de dedicació recomanat: 4 hores



Universitat Oberta de Catalunya

Silvia Llorente Viejo

L'encàrrec i la creació d'aquest recurs d'aprenentatge UOC han estat coordinats pel professor: Joan Manel Marquès Puig

Primera edició: setembre 2020
© d'aquesta edició, Fundació Universitat Oberta de Catalunya (FUOC)
Av. Tibidabo, 39-43, 08035 Barcelona
Autoria: Silvia Llorente Viejo
Producció: FUOC
Tots els drets reservats

Cap part d'aquesta publicació, incloent-hi el disseny general i la coberta, no pot ser copiada, reproduïda, emmagatzemada o transmesa de cap manera ni per cap mitjà, tant si és elèctric com mecànic, òptic, de gravació, de fotocòpia o per altres mètodes, sense l'autorització prèvia per escrit del titular dels drets.

Índex

Introducció 5

Objectius 6

1. Què és REpresentational State Transfer (REST)? 7
1.1. Visió general 7
1.2. Relació entre REST i HTTP 9
1.2.1. Mètodes HTTP i operacions REST 10
1.3. Implementació de serveis basats en REST 11
1.3.1. Format de les peticions 14
1.3.2. Javascript Object Notation (JSON) 18
1.3.3. Format de les respostes 19
1.3.4. eXtensible Markup Language (XML) 20
1.4. Comparació amb altres tipus de serveis web 22
1.4.1. Serveis web basats en SOAP 22

2. Serveis REST amb llenguatge Java 30
2.1. Operacions pròpies del servidor 34
2.1.1. Creació del servei 34
2.1.2. Definició de les operacions 35
2.1.3. Enviament de la petició 38
2.1.4. Enviament de la resposta 38
2.2. Operacions pròpies del client 39
2.2.1. Programació d'una aplicació client de servei REST 39

3. Serveis REST amb altres llenguatges de programació (Node.js, Python) 42
3.1. Node.js 42
3.1.1. Desenvolupament d'una API REST amb Node.js i ExpressJS 42
3.2. Python 45
3.2.1. Desenvolupament d'una API REST amb Python i Flask 45

Resum 48

Bibliografia 49

Introducció

La comunicació entre sistemes ha anat evolucionant molt ràpidament en els últims anys. Hem passat de fer comunicacions «a mida» entre sistemes a definir mecanismes oberts basats en estàndards i directrius.

L'objectiu d'aquest mòdul és descriure els serveis web basats en REpresentational State Transfer (REST). REST defineix una arquitectura de com implementar serveis web basant-se en els mètodes de l'Hypertext Transfer Protocol (HTTP), el protocol web per excel·lència.

Així, doncs, es descriu el funcionament general de REST, la seva relació amb HTTP i com es poden fer les peticions i rebre les respostes. També es comparen els serveis web REST amb els serveis web basats en Simple Object Access Protocol (SOAP), un dels mecanismes més utilitzats per a comunicar sistemes heterogenis abans de l'aparició de REST.

A més, es donen alguns detalls de com s'hauria d'implementar un servei web basat en REST donant alguns exemples concrets en diferents llenguatges de programació com són Java, Node.js i Python.

Objectius

Amb l'estudi d'aquest mòdul assolireu els objectius següents:

1. Conèixer què és REpresentational State Transfer (REST) i la seva arquitectura bàsica.

2. Conèixer la relació entre REST i els diferents mètodes del protocol HTTP.

3. Conèixer les bases per definir serveis REST.

4. Conèixer altres tipus de serveis web i la seva relació amb REST.

5. Aprendre a implementar serveis REST basats en Java i com connectar-se a ells.

6. Aprendre les bases per implementar clients i serveis REST basats en altres llenguatges de programació.

1. Què és REpresentational State Transfer (REST)?

REpresentational State Transfer (REST) defineix una arquitectura de com implementar serveis web distribuïts de forma més simple que altres arquitectures existents com poden ser *Simple Object Access Protocol* (SOAP) o *Remote Method Invocation* (RMI).

Sobre REST, SOAP i RMI

REST: *REpresentational State Transfer* és un estil de serveis web basats en HTTP.

SOAP: *Simple Object Access Protocol* és un estil de serveis web que fa un ús extensiu del llenguatge XML.

RMI: *Remote Method Invoation* és un mecanisme per invocar objectes remots en llenguatge Java.

REST fa servir els mètodes d'*HyperText Transfer Protocol* (HTTP)[1] per a invocar les operacions *Create, Read, Update* i *Delete* CRUD (en català, Creació, Lectura, Modificació i Esborrat) en comptes de definir operacions específiques com es fa en altres mecanismes d'invocació remota d'operacions (p. ex., SOAP o RMI). Això vol dir que, quan en un servei basat en REST es vol definir una operació de consulta d'un recurs, en comptes de definir una operació amb el nom `consultaRecurs` on es passa el codi del recurs com a paràmetre, es defineix una *Uniform Resource Locator* (URL) d'accés al recurs amb el mètode HTTP GET, com ara `www.example.com/recurs/codi`[2]. En aquest cas concret, el paràmetre es passa en l'URL, però es poden fer servir diferents mecanismes i/o formats per al pas de paràmetres. El resultat d'aquesta operació s'enviarà dins un missatge de resposta HTTP i pot tenir diferents formats, com veurem més endavant en aquest mòdul.

(1) **Internet Engineering Task Force (IETF)** (2014) Hypertext Transfer Protocol (HTTP/1.1), RFC 7230 a 7235, <https://tools.ietf.org/html/rfc7230> a <https://tools.ietf.org/html/rfc7235>

(2) Nota general: cap dels enllaços inclosos en el text porta guions de separació

Finalment, cal destacar que REST no és un estàndard, tot i que està basat en estàndards com ara HTTP, *eXtensible Markup Language* (XML)[3] o *Javascript Object Notation* (JSON)[4]. Al llarg d'aquest mòdul veurem com fer servir aquests estàndards per a oferir serveis web basats en REST.

(3) **World Wide Web Consortium (W3C)** (2006). Extensible Markup Language (XML) 1.1 (Second Edition), <https://www.w3.org/TR/xml11>

(4) **European Computer Manufacturers Association (ECMA)** (2017). The JSON (JavaScript Object Notation) Data Interchange Syntax ECMA – 404 Standard, <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>

1.1. Visió general

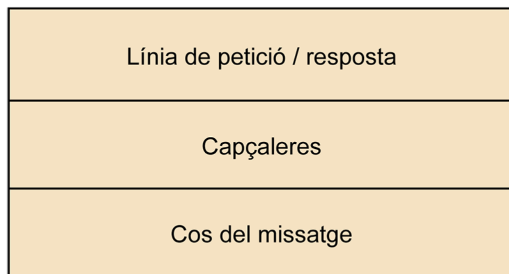
Com ja s'ha dit, REST no és un estàndard. Es tracta més aviat d'un estil arquitectural o patró de disseny d'una Interfície de Programació d'Aplicacions (API, de les seves sigles en anglès *Application Programming Interface*) que funciona sobre HTTP. El va definir inicialment en Roy Fielding a la seva tesi doctoral l'any 2000.

Referència bibliogràfica

Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. [en línia]: <https://www.ics.uci.edu/~fielding/pubs/ dissertation/fielding_dissertation.pdf>.

Per a entendre millor el funcionament de REST, explicarem breument el funcionament del protocol HTTP. HTTP és un protocol de tipus Client – Servidor on client i servidor es comuniquen mitjançant missatges de petició (del client cap al servidor) i de resposta (del servidor cap al client) amb un format predeterminat, tal i com es mostra a la Figura 1.

Figura 1. Estructura general dels missatges intercanviats mitjançant HTTP



La primera línia del missatge canvia segons si el missatge és de petició o de resposta. En el cas del missatge de petició, la línia de petició conté el mètode HTTP, l'URL del recurs (sense el protocol ni el nom del servidor) i la versió de protocol HTTP que s'està fent servir. A continuació es pot veure un exemple de línia de petició, on es fa servir el mètode GET i es demana la pàgina `index.html` amb la versió 1.1 del protocol HTTP.

```
GET index.html HTTP/1.1
```

En el cas del missatge de resposta, la línia de resposta conté la versió HTTP amb la que respon el servidor, el codi d'estat de la petició i una descripció textual associada a aquest codi. A continuació es pot veure com seria una resposta exitosa a la petició anterior, on s'indica que la versió del protocol és 1.1, que el codi de resposta és 200, que correspon a una resposta correcta indicant que s'ha trobat la pàgina `index.html` al servidor i s'envia el seu contingut en el cos del missatge de resposta.

```
HTTP/1.1 200 OK
```

L'apartat de capçaleres es pot trobar en els dos tipus de missatges, tant petició com resposta. Algunes capçaleres són comunes als dos tipus de missatges i altres són específiques de la petició o la resposta. A continuació es pot trobar un exemple de capçalera comuna als dos tipus de missatges, `Content-Length`, que ens indica la longitud en bytes del cos del missatge.

```
Content-Length: 2000
```

Finalment, el cos del missatge conté informació referent a la petició, com poden ser paràmetres per a poder-la realitzar, o el contingut de la resposta. En el cas concret del mètode GET, el cos del missatge de petició ha de ser obligatòriament buit.

A més, HTTP és un protocol sense estat, és a dir, el resultat de les peticions no depèn de peticions anteriors. En cas de necessitar emmagatzemar l'estat és necessari fer servir mecanismes externs com ara galetes (*cookies*, en anglès) o sessions.

Així doncs, REST fa servir els mecanismes proporcionats per HTTP per poder implementar serveis web. L'estil arquitectural de REST té algunes característiques específiques que detallem a continuació:

- **No té estat**: cada petició del client cap al servidor ha de contenir tota la informació necessària per entendre la petició i no pot aprofitar el context emmagatzemat al servidor.

- **Memòria cau**: per millorar l'eficiència de les respostes, aquestes han de poder indicar si es poden emmagatzemar o no a les memòries cau dels sistemes intermedis, com ara representants (*proxies*, en anglès) o fins i tot navegadors.

- **Interfície uniforme**: tots els recursos són accessibles per mitjà d'una interfície genèrica (p. ex., la que proporcionen els mètodes HTTP, GET, POST, PUT, DELETE, etc.).

- **Recursos amb nom**: el sistema està compost per recursos als quals se'ls dona nom amb una URL (p. ex., `www.example.com/recurs/codi`, donem accés a un recurs a partir del seu codi).

- **Representacions de recursos interconnectades**: els recursos s'interconnecten per mitjà d'URL i permeten al client passar d'un estat a un altre «navegant» entre representacions. Se segueix la mateixa lògica que en una aplicació web, on les pàgines web estan interconnectades entre elles i podem navegar seguint els enllaços. Per exemple, l'URL `www.example.com/recurs/llista` permet demanar el llistat de tots els recursos. En el resultat s'hauria de retornar una URL a cada recurs, per obtenir les seves dades bàsiques o fins i tot més detalls, amb URL del tipus `www.example.com/recurs/codi` o `www.example.com/recurs/codi/detalls`.

En les seccions següents s'explica amb més detall com REST fa servir HTTP per enviar i rebre dades i gestionar recursos fent servir les operacions CRUD.

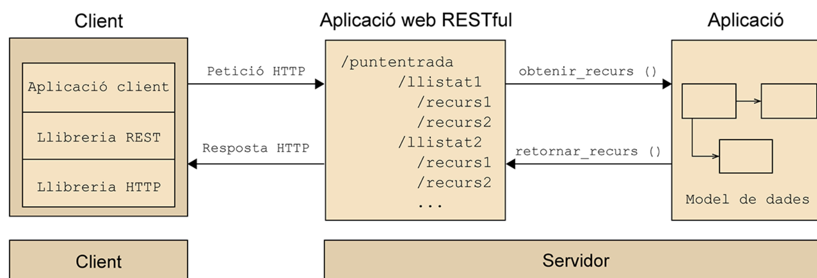
1.2. Relació entre REST i HTTP

Una aplicació web RESTful exposa la seva informació a través de recursos. A més, dona la possibilitat al client de fer diferents accions sobre aquests recursos, com ara crear-ne de nous (p. ex., crear un nou usuari o producte) o canviar els ja existents (p. ex., modificar la descripció d'un producte o el seu preu). Per fer-ho, es fan servir les ordres HTTP en comptes de definir operacions específiques. Així, per demanar les dades d'un producte, no es defineix una operació com ara `demanarDadesProducte(idProducte)`, sinó que farem servir el mètode HTTP GET, demanant les dades en la línia d'aquest

protocol `GET /dadesProducte?idProducte`. El resultat de l'operació pot tenir qualsevol format, ja que HTTP permet enviar qualsevol tipus de fitxer com a resposta d'una petició.

El servei implementat en una aplicació que hi ha al darrere d'una aplicació web RESTful pot no tenir cap mena d'interfície web i la seva comunicació amb l'aplicació RESTful es pot fer per mitjà d'altres mecanismes de comunicació com ara sockets, Remote Produce Call (RPC), Remote Method Invocation (RMI) o execució directa de programes des de l'aplicació RESTful.

Figura 2. Arquitectura d'un servei REST



El diagrama de la Figura 2 ens mostra l'arquitectura general d'un servei basat en REST, on una aplicació web RESTful ofereix accés a una aplicació que s'executa en el servidor a través del protocol HTTP. Els clients es connectaran amb l'aplicació web RESTful per mitjà d'aquest protocol. En el client, cal tenir una aplicació client, una llibreria REST (que pot ser opcional) i una llibreria HTTP, que estableixi la comunicació amb el servei.

1.2.1. Mètodes HTTP i operacions REST

Els mètodes HTTP que es fan servir en una API REST són els que apareixen en la Taula 1. En la Taula 1 es presenten els mètodes HTTP utilitzats en REST, l'operació CRUD a la qual correspon i una petita descripció del que ha de fer l'operació que fa servir cada mètode.

Taula 1. Mètodes HTTP i la seva funció en REST

Mètode	Operació CRUD	Descripció
POST	Create – Creació	S'ha d'utilitzar per crear un nou recurs que s'afegeix a la col·lecció de recursos actual. També es pot utilitzar per modificar, tot i que no se'n recomana l'ús[5].
GET	Read – Lectura	S'ha d'utilitzar per demanar les dades d'un recurs. Tot i que GET pot rebre paràmetres, no s'hauria de fer servir per operacions de creació i/o modificació.
PUT	Update – Actualització	S'ha d'utilitzar per actualitzar un recurs existent, que substitueix el recurs actual pel recurs que s'envia en el missatge de petició. En el cas que el recurs que envia el client sigui un recurs nou, aquest mètode el crearia i informaria convenientment el client.

[5] Una operació *Idempotent* és la que dona sempre el mateix resultat atesos els mateixos paràmetres. En HTTP, GET, PUT i DELETE són idempotents.
Per contra, una operació *No idempotent* és la que pot donar diferents resultats amb els mateixos paràmetres.

Mètode	Operació CRUD	Descripció
PATCH	Partial Update – Actualització parcial	S'ha d'utilitzar per modificar un recurs, sense enviar-lo sencer, només enviant les dades que s'han de modificar. És semblant a PUT, però ha de contenir instruccions que indiquin quines dades del recurs original s'han de modificar.
DELETE	Delete - Esborrat	S'ha d'utilitzar per esborrar un recurs.

En la Taula 2 es descriuen els possibles codis de resposta de cada una de les operacions de la Taula 1. Aquests codis corresponen a codis del protocol HTTP, tal com es defineixen a Internet Engineering Task Force (2014).

Referència bibliogràfica

Internet Engineering Task Force (IETF) (2014). Hypertext Transfer Protocol (HTTP/1.1), RFC 7230 a 7235. [en línia]: <https://tools.ietf.org/html/rfc7230> i <https://tools.ietf.org/html/rfc7235>.

Taula 2. Possibles codis de resposta HTTP a cada operació REST

Mètode	Codi de resposta correcta	Codi de resposta errònia
POST	201 Recurs creat	404 Recurs no trobat 409 Conflicte, recurs ja existeix
GET	200 Correcte	404 Recurs no trobat
PUT	200 Correcte 201 Recurs creat	204 No hi ha més contingut 404 Recurs no trobat
PATCH	200 Correcte	204 No hi ha més contingut 404 Recurs no trobat
DELETE	200 Correcte	404 Recurs no trobat

1.3. Implementació de serveis basats en REST

Per tal d'implementar un servei basat en REST, és important definir com ha de ser. De fet, un servei basat en REST el que fa és exposar una sèrie d'operacions sobre recursos emmagatzemats en un servidor per mitjà d'una interfície de programació d'aplicacions (API, d'ara endavant).

Així doncs, un dels conceptes més rellevants dins REST és el recurs. Qualsevol informació a la qual es pot donar un nom és un recurs: un document o imatge, un servei temporal (p. ex., «el temps durant els propers tres dies a Barcelona»), una col·lecció d'altres recursos, un objecte no virtual (p. ex., una persona), etc. D'aquesta manera, un concepte que es pugui representar com una referència a un hipertext pot ser un recurs. Els recursos es poden agrupar en col·leccions, on cada col·lecció pot contenir diversos recursos desordenats. Les col·leccions també són recursos per elles mateixes.

Veiem a continuació alguns exemples per aclarir el concepte de recurs. `clients` pot ser un recurs de tipus col·lecció i `client` un recurs únic (en un domini com ara el bancari). Es pot identificar la col·lecció `clients` amb l'URI `/clients`. I podem identificar un únic client amb l'URI `/clients/{clientId}`. A més, un recurs pot tenir subcol·leccions. Per exemple, la subcol·lecció `comptes` d'un `client` particular es pot identificar fent servir

l'URI `/clients/{clientId}/comptes`. De manera similar a les col·leccions, també es pot accedir a un recurs únic dins les subcol·leccions. Així, un `compte` concret dins els `comptes` d'un `client` es podria accedir amb `/clients/{clientId}/comptes/{compteId}`.

Com ja hem vist als exemples, les API REST fan servir Uniform Resource Identifiers (URI) per accedir als recursos. El dissenyador d'una API REST ha de definir un model de recursos que sigui entenedor per als desenvolupadors dels seus clients potencials. Quan als recursos se'ls donen noms amb lògica, una API és intuïtiva i fàcil d'utilitzar. Si es fa malament, aquesta mateixa API pot ser difícil de fer servir i entendre.

Tot i que no hi ha un estàndard a l'hora de definir els noms dels recursos d'una API REST, posem aquí una sèrie de directrius per tal de fer-ho de la manera més entenedora possible.

Directrius per a la definició d'URI per a recursos d'una API REST

1) Fer servir noms (coses) en comptes de verbs (accions) per definir els recursos. Els noms tenen una característica que el verbs no tenen, com són els atributs. Alguns exemples de noms poden ser usuaris del sistema, comptes d'usuari, dispositius de xarxa, etc. Dintre dels noms podem distingir entre recursos únics (en singular) i col·leccions (en plural).

Exemples:

```
http://api.com/gestio-dispositius
http://api.com/gestio-dispositius/dispositius-
gestionats
http://api.com/gestio-dispositius/dispositius-
gestionats/{id}
```

2) La consistència és clau. S'han de fer servir convencions de noms consistents i formatació d'URI amb la mínima ambigüitat i màxima llegibilitat i manteniment. Alguns consells de disseny són: fer servir la barra (/) per indicar relacions jeràrquiques, no posar la barra (/) al final de l'URI, no fer servir majúscules i no fer servir extensions en els recursos.

Exemples:

Jerarquia

```
http://api.com/gestio-dispositius/dispositius-
gestionats
http://api.com/gestio-dispositius/dispositius-
gestionats/{id}
```

Referència bibliogràfica

Internet Engineering Task Force (IETF) (2005). Uniform Resource Identifier (URI): Generic Syntax, RFC 3986, <https://tools.ietf.org/html/rfc3986>.

Ús de barra al final no recomanat (la segona opció no seria correcta)

```
http://api.com/gestio-dispositius/dispositius-
gestionats
http://api.com/gestio-dispositius/dispositius-
gestionats/
```

Ús de majúscules. Es poden fer servir en el protocol i el servidor, però no en la part del recurs (la tercera URI no seria vàlida ja que posa Meva en majúscula)

```
HTTP://API.ORG/meva-carpeta/meu-doc
http://api.org/meva-carpeta/meu-doc
http://api.org/Meva-carpeta/meu-doc
```

No fer servir extensions de fitxers (la segona opció no seria correcta)

```
http://api.com/gestio-dispositius/dispositius-
gestionats
http://api.com/gestio-dispositius/dispositius-
gestionats.xml
```

3) No fer servir els noms de les operacions CRUD en els noms dels recursos. El que s'ha de fer es utilitzar el mètode HTTP corresponent, com ara GET per obtenir les dades, POST per crear-les, etc. Es fa servir la mateixa URI, però diferent mètode HTTP i aleshores l'efecte sobre els recursos és diferent.

Exemples:

Per obtenir tots els dispositius

```
HTTP GET http://api.com/gestio-dispositius/dispositius-
gestionats
```

Per crear un dispositiu nou

```
HTTP POST http://api.com/gestio-dispositius/
dispositius-gestionats
```

4) Fer servir l'apartat de consulta de l'URI per fer filtratge o ordenació dels recursos en comptes de crear noves entrades de l'API. La secció de query és tot el que hi ha darrera de `?` i per posar més d'un paràmetre es posa `&`. El format d'aquesta secció és `?clau1=valor1&clau2=valor2`.

Exemples:

```
http://api.com/gestio-dispositius/dispositius-
gestionats
```

```
http://api.com/gestio-dispositius/dispositius-
gestionats?regio=cat
```

```
http://api.com/gestio-dispositius/dispositius-
gestionats?regio=cat&data=2020-03-18
```

1.3.1. Format de les peticions

Les peticions a un servei REST han d'anar en un missatge de petició HTTP. Segons de quin mètode HTTP es tracti, podrem enviar les dades de manera diferent. Primer descriurem com es poden enviar les dades i després indicarem amb quins mètodes HTTP es pot fer servir cada mecanisme.

URL

Les dades es poden enviar en la mateixa URL de petició. Així, quan parlem de recursos concrets dintre de col·leccions, podem fer servir identificadors per referir-nos a un recurs concret. En un exemple anterior, hem fet servir la col·lecció `clients` per referir-nos als clients d'una entitat bancària. L'URI `/clients` identifica la col·lecció sencera, és a dir, tots els clients. La manera d'identificar un únic client seria una URI del tipus `/clients/{clientId}`, on hem de donar un identificador de client per fer la petició en el camp `{clientID}`. L'URI resultant seria `/clients/11111111H`, on es demanen les dades del client identificat per l'identificador `11111111H`, que correspon a un document nacional d'identitat (DNI).

Secció query de l'URL

També dintre de la mateixa URL es poden enviar dades en la secció query de l'URL. La secció query comença amb el símbol `?` i pot contenir diverses parelles `clau=valor`, separades entre elles pel símbol `&`. Així, seguint amb l'exemple dels clients d'un banc, podríem demanar els detalls d'un usuari de la manera següent: `/clients/11111111H?info=detalls`, on la clau `info` pot prendre diferents valors, en aquest cas concret es demanen tots els detalls de l'usuari. Si posem com a exemple el compte bancari d'un usuari, podríem filtrar els moviments per data de la manera següent: `/clients/11111111H/comptes/12341234?data-ini=1-1-2020&data-fi=31-03-2020`

Cos del missatge de petició

Hi ha diferents mecanismes per enviar les dades dins el cos del missatge de petició:

- Format `application/x-www-form-urlencoded`, que és el mateix format que a la secció query, fent servir el format `clau=valor`. Amb aquest format s'envien tots els camps necessaris separats pel símbol `&`. En aquest cas, un exemple d'URL podria ser `/clients/11111111H/comptes/12341234` i els paràmetres dins el cos del missatge serien `data-ini=1-1-2020&data-fi=31-03-2020`.

Sobre MIME

Internet Engineering Task Force (IETF) (2015). Returning Values from Forms: multipart/form-data, RFC 7578, <https://tools.ietf.org/html/rfc7578>.

- Format `multipart/form-data`, que és un missatge multipart de MIME (Multipurpose Internet Mail Extensions), on s'envien els paràmetres dins el cos del missatge separats per una cadena de caràcters anomenada `boundary`. L'avantatge d'aquest format respecte als anteriors és que permet enviar més dades, fins i tot dades en format binari. Aquest format està orientat a enviar fitxers amb un formulari que es mostra a un usuari en un navegador.

- Continguts de tipus `application/xml` (o `text/xml`), `application/json` i altres suportats pel servei REST (que s'han d'indicar en les capçaleres del servei, com veurem més endavant).

XML i JSON

XML vol dir *eXtensible Markup Language* i els seus tipus MIME poden ser application/xml (quan està adreçat a un servidor) o text/xml (quan està adreçat a ser entès per una persona).

JSON vol dir *JavaScript Object Notation* i el seu tipus MIME és application/json.

Exemple de missatge `multipart/form-data` enviat dins un missatge HTTP d'una ordre POST:

```
Content-Type: multipart/form-data; boundary=--73532303139
Content-Length: 834
--73532303139
Content-Disposition: form-data; name="text1"
text 123 abc
--73532303139
Content-Disposition: form-data; name="text2"
xyz
--73532303139
Content-Disposition: form-data; name="file1"; filename="a.txt"
Content-Type: text/plain
Contingut fitxer a.txt.
--73532303139
Content-Disposition: form-data; name="file2"; filename="a.html"
Content-Type: text/html
DOCTYPE html><title>Contingut a.html.</title>
--73532303139
Content-Disposition: form-data; name="file3"; filename="fish.jpg"
Content-Type: image/jpeg
Dades binàries aquí en format base64
--73532303139--
```

En aquest missatge s'estan enviant dos camps de text (text1 i text2), un fitxer de text pla anomenat `a.txt`, un fitxer HTML anomenat `a.html` i una imatge anomenada `fish.jpg`. Cada fitxer indica quin és el seu tipus MIME amb la capçalera `Content-Type`. Cada tipus de dades està separat pel camp `boundary`, que té el valor `--73532303139`, tal i com es defineix a l'inici del missatge a la capçalera `Content-Type` del missatge complet.

A continuació, posarem uns exemples de com poden ser les peticions de cada un dels mètodes HTTP. S'inclouen tant les línies de petició com les capçaleres del missatge i el contingut del cos del missatge, si n'hi ha.

Exemple petició POST

```
POST /client/json HTTP/1.1
Accept: application/json
Content-Type: application/json
Content-Length: 85
Host: exemple.com
{
  "Id": 12345,
  "Client": "Arnau Sola",
  "Quantitat": 3,
  "Preu": 10.00
}
```

En aquest exemple fem servir el format JavaScript Object Notation (JSON) (ECMA, 2017) per enviar les dades. A l'apartat «JavaScript Object Notation» s'explica en detall aquest format, agafant aquest exemple de dades com a referència. En aquest cas, les dades s'envien dins el cos del missatge.

A més, trobem diverses capçaleres de petició: `Accept`, que ens indica en quin format ens pot arribar la resposta, `Content-Type`, que ens indica quin és el tipus de contingut que enviem, `Content-Length`, que ens indica la longitud del cos del missatge i `Host`, que ens diu a quin servidor ens hem de connectar.

Exemple petició GET

```
GET /assigs/json HTTP/1.1
Accept: application/json
Host: exemple.com
```

En aquesta petició GET podem veure que no hi ha cap contingut en el missatge (el mètode GET no ho permet) i que s'accepta que la resposta sigui en format JSON (`application/json`). També es pot veure el servidor a qui fem la petició.

Exemple petició PUT

```
PUT /client/json HTTP/1.1
Accept: application/json, application/xml
Content-Type: application/json
Content-Length: 85
Host: exemple.com
{
```

```
  "Id": 12345,
  "Client": "Arnau Sola",
  "Quantitat": 1,
  "Preu": 10.00
}
```

Aquesta petició PUT és molt semblant a la petició POST. Veiem que s'envien les dades completes del client, canviant el valor 3 que hi havia a POST per un 1. Això farà que es modifiqui tot el recurs al servidor.

Exemple de petició PATCH

```
PATCH /client/json HTTP/1.1
Accept: application/json, application/xml
Content-Type: application/json
Content-Length: 85
Host: exemple.com
{
  "Id": 12345,
  "Preu": 15.00
}
```

Aquesta petició PATCH és un subconjunt de les fetes a POST i PUT. Aquí només s'envia l'id i el preu, que s'ha de modificar. La resta de les dades del recurs es mantenen.

Exemple de petició DELETE

```
DELETE /client/json/1234 HTTP/1.1
Accept: application/json, application/xml
Host: exemple.com
```

Aquesta petició DELETE conté les dades del recurs que s'ha d'esborrar a l'URL. En aquest cas, es vol esborrar el client amb id 1234. No hi ha contingut del missatge.

La Taula 3 mostra un resum de com es poden enviar les dades segons el mètode HTTP utilitzat. En alguns casos es poden combinar diverses opcions. Alguns mètodes HTTP podrien suportar més maneres d'enviar les dades, però aleshores no seguirien les directrius REST. És per això que no apareixen a la taula, tot i que l'especificació del mètode en HTTP ho suportaria.

Taula 3. Com enviar les peticions segons el mètode HTTP utilitzat

Mètode	Opcions de petició
POST	URL, que indica el recurs a crear. Secció query, que permet passar paràmetres. Cos del missatge de la petició. En aquest cas, es pot enviar un fitxer del tipus suportat pel servei REST, `application/x-www-form-urlencoded o multipart/form-data`.
GET	URL, que indica el recurs que s'ha de consultar. Secció query, que permet passar paràmetres. En aquest cas no es pot enviar res al cos del missatge perquè el mètode no ho suporta.
PUT	URL, que indica el recurs a reemplaçar o crear. Cos del missatge de la petició amb un fitxer del tipus suportat pel servei REST.
PATCH	URL, que indica el recurs que s'ha de modificar parcialment. Cos del missatge de la petició amb un fitxer del tipus suportat pel servei REST.
DELETE	URL, que indica el recurs que s'ha d'esborrar.

1.3.2. Javascript Object Notation (JSON)

JavaScript Object Notation (JSON) (European Computer Manufacturers Association, 2017) és un format lleuger d'intercanvi de dades. Es tracta d'un format fàcil de generar per part de les persones i d'interpretar per part de les màquines. A més, és un format de text independent del llenguatge de programació, però, alhora, fa servir convencions habituals per a llenguatges com ara C, C++, Java, Python i molts altres. Aquestes propietats el fan molt adient com a llenguatge d'intercanvi de dades.

JSON es basa en dues estructures:

- Una col·lecció de parells nom/valor. En molts llenguatges de programació això es representa com un objecte, registre, estructura, diccionari, taula de hash o llista amb claus, entre altres.

- Una llista ordenada de valors. En molts llenguatges això es representa com un vector, matriu, llista o seqüència.

```
{
  "Id": 12345,
  "Client": "Arnau Sola",
  "Quantitat": 3,
  "Preu": 10.00
}
```

En l'exemple, es representa un objecte amb les dades d'un client. Les claus ({}) indiquen l'inici i el final de l'objecte. A dins, trobem un conjunt de noms/valors. El primer nom que trobem és «`Id`» amb valor 12345. El nom se separa

del valor amb el símbol «:». Quan un nom o un valor té el símbol «"», és una cadena de caràcters (p. ex., «Id» o «Arnau Sola»). Per separar parelles nom/valor entre sí, fem servir el símbol «,». També podem representar nombres, tant valors enters (p. ex., 12345 o 3) com reals (p. ex., 10.00). Es poden incloure objectes dintre d'objectes i també fer servir altres tipus de dades com poden ser valors booleans (true, false). Per a una descripció més detallada, consultar European Computer Manufacturers Association (2017).

1.3.3. Format de les respostes

Les respostes proporcionades per un servei REST es poden indicar a les capçaleres de petició (vegeu l'apartat 1.3.1). Per exemple, si volem que el servei retorni un fitxer de tipus text, a la petició ho indicarem mitjançant la capçalera Accept, on hem de posar com a valor text/plain. En aquesta capçalera, l'aplicació client indica quins són els formats de resposta del servei acceptats. Es poden posar diferents valors, separats per comes.

Exemple de capçalera Accept de la petició al servei REST que indica el format o formats suportats pel client.

Els més habituals són fitxers en format JSON o XML, que són els que apareixen en l'exemple. A la primera línia s'indiquen tots dos formats, a la segona línia només JSON i a la tercera només XML. En una petició només s'envia una capçalera Accept, tot i que pot contenir més d'un valor, com es veu en el primer exemple.

```
Accept: application/json, application/xml
Accept: application/json
Accept: application/xml
```

A continuació es mostra un exemple de resposta d'un servei REST per a cada mètode HTTP.

Exemple de resposta POST

```
HTTP/1.1 200 OK
Content-Length: 19
Content-Type: application/json
{"success":"true"}
```

En aquest exemple es respon amb el codi que indica que la petició ha anat bé. A més, trobem dues capçaleres que indiquen la longitud i el tipus de contingut. Finalment, hi ha el contingut del missatge, que també indica que la petició ha funcionat correctament. En aquest cas les dades s'envien en format JSON, amb un únic parell nom/valor dins de l'objecte amb un valor de tipus booleà.

Exemple de resposta GET

```
HTTP/1.1 200 OK
Content-Type: application/json
{
    "public": {
        "alumnes": "delegats",
        "assignatures": "totes",
        "professors": "tots"
    },
    "privat": {
        "jo": "perfil",
        "assignatures": "XAI"
    }
}
```

En aquest exemple es respon amb el codi que indica que la petició ha anat bé. En aquest cas, es retornen dades en format JSON referents a objectes associats a assignatures, professors i alumnes. En aquest cas hi ha dos objectes, un anomenat «`public`» i un altre anomenat «`privat`», que contenen diversos parells d’objectes nom/valor. Tots els valors són de tipus cadena de caràcters.

Exemple resposta PUT, PATCH i DELETE

```
HTTP/1.1 200 OK
Content-Type: application/xml; charset=utf-8
<?xml version="1.0" encoding="utf-8"?>
<Response>
    <ResponseCode>0</ResponseCode>
    <ResponseMessage>Success</ResponseMessage>
</Response>
```

En aquest exemple es respon amb el codi que indica que la petició ha anat bé, i les dades es retornen en format XML. Aquest format està explicat amb més detall a l'apartat «eXtensible Markup Language (XML)».

1.3.4. eXtensible Markup Language (XML)

eXtensible Markup Language (XML) (World Wide Web Consortium, 2006) és un format estandarditzat per W3C[6], que descriu com han de ser els documents XML. Aquests documents estan formats per unitats d’emmagatzematge anomenades entitats, que poden contenir diferents tipus de dades. Les dades

(6)World Wide Web Consortium (W3C), <https://www.w3.org/> [Data de consulta: març de 2020]

formalitzades estan compostes per caràcters, alguns d'ells definint el marcatge (*markup*, en anglès) i altres definint les dades. El marcatge codifica una descripció de la distribució de l'emmagatzematge del document, com també l'estructura lògica. Amb XML es proporciona un mecanisme per poder imposar restriccions tant a la distribució com a l'estructura lògica de les dades.

Nota

XML deriva de SGML, que vol dir *Standard Generalized Markup Language*. Per definició, els documents XML són conformes amb SGML.

```
<?xml version="1.0" encoding="utf-8"?>
<Response>
    <ResponseCode>0</ResponseCode>
    <ResponseMessage>Success</ResponseMessage>
</Response>
```

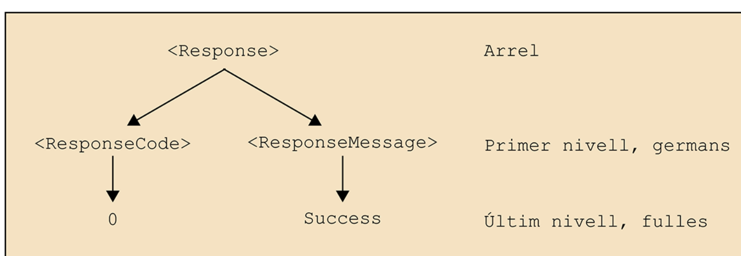
En l'exemple, es representa un codi de resposta exitós a una petició PUT, PATCH o DELETE que hem vist a l'apartat 1.3.2. La primera línia indica que es tracta d'un document XML de la versió 1.0 i codificat amb UTF-8. Aquesta línia és obligatòria en els documents XML vàlids. Després trobem l'element arrel `Response`. L'inici d'un element es marca amb els símbols «<» i «>» i el final amb «</» i «>». A dins dels símbols hi ha d'haver el nom de l'element. Dins l'element arrel trobem els subelements `ResponseCode` i `ResponseMessage`, que són fills de `Response` i germans entre ells. Els elements tenen una relació jeràrquica des de l'arrel a les fulles. Tots els subelements d'un mateix nivell són germans (*siblings*, en anglès). Finalment, trobem els valors `0` i `Success`, que corresponen a les fulles, que són elements finals, que no tenen fills i que poden contenir valors de tipus cadena de caràcters, nombres decimals o reals i altres tipus de dades bàsics definits a l'estàndard XML.

UTF-8

UTF-8 vol dir *8-bit Unicode Transformation Format*. És un format de codificació de caràcters definit per ISO (*Internation Organization for Standardization*).

La Figura 3 mostra la relació entre els diferents elements XML que apareixen a l'exemple.

Figura 3. Jerarquia d'un document XML



En aquest cas, hem vist que la resposta té format XML. També seria possible rebre una resposta en format JSON com en l'exemple amb el mètode POST vist a l'apartat «JavaScript Object Notation (JSON)».

1.4. Comparació amb altres tipus de serveis web

Els altres serveis web molt utilitzats han estat els basats en el Simple Object Access Protocol (SOAP)[7]. SOAP és un estàndard del World Wide Web Consortium (W3C), que defineix un marc de treball basat en XML per comunicar processos remots. Tant els missatges que s'intercanvien, com el mecanisme per definir el servei fan servir XML per a l'enviament i la descripció de les dades. En aquest apartat fem un petit resum de SOAP i altres tecnologies relacionades i comparem les seves característiques amb els serveis web REST.

(7)World Wide Web Consortium (W3C) (2007). Simple Object Access Protocol (SOAP) Versió 1.2, <https://www.w3.org/TR/soap12/>

1.4.1. Serveis web basats en SOAP

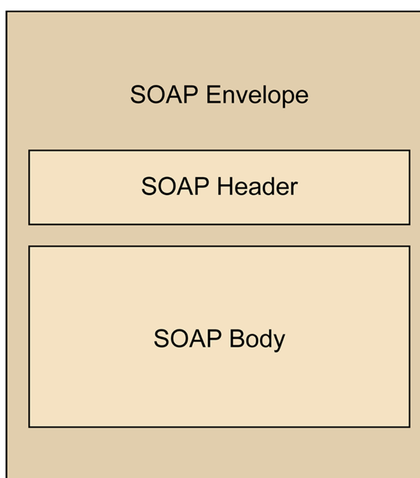
L'estàndard SOAP és un protocol de comunicació d'aplicacions que defineix el format dels missatges de petició i resposta. Aquest protocol és independent de la plataforma i el llenguatge de programació en què estan implementats client i servidor, ja que la comunicació es basa en missatges expressats en format XML. Aquests missatges s'intercanvien principalment sobre el protocol HTTP, tot i que podrien enviar-se sobre altres protocols de nivell d'aplicació.

En SOAP els missatges entre client i servei web sempre s'envien en format XML. A continuació es descriuen algunes de les característiques dels serveis SOAP, com ara l'enviament de peticions i respostes o com es defineixen les operacions d'un servei.

Missatges de petició i resposta SOAP

La Figura 4 mostra un diagrama del format de les peticions i respostes SOAP. El primer element és el SOAP Envelope (`<Envelope>`), que envolta totes les dades d'una petició i/o resposta. Els altres elements presents en un missatge SOAP són el SOAP Header i el SOAP Body.

Figura 4. Diagrama del format de peticions i respostes SOAP



El SOAP Header (<Header>) és un element opcional del missatge SOAP i conté informació relacionada amb el servei que ha de ser processada pels nodes SOAP al llarg del flux del missatge. La informació definida només depèn del servei web, és a dir, les dades que s'envien a la capçalera només les entén aquest servei concret.

El SOAP Body (<Body>) és un element obligatori que conté informació destinada al receptor últim del missatge. Aquest element i els seus subelements s'utilitzen per intercanviar informació entre l'emissor SOAP i el receptor últim del missatge SOAP. SOAP defineix un subelement, el SOAP Fault (<Fault>), que es fa servir per enviar errors que s'hagin produït. La resta d'elements que apareixen al SOAP Body els defineix el servei web i són les operacions, els paràmetres i els resultats de les operacions que s'ofereixen.

A continuació veiem un exemple de missatge de petició SOAP que s'envia dins una ordre POST d'HTTP. El cos del missatge s'envia en format `application/soap+xml`, que indica que s'envia en format XML, seguint les directrius de l'estàndard SOAP.

Dins el missatge SOAP podem veure que hi ha un element <Header> que conté un subelement que indica que el temps màxim per respondre a la petició és de 10.000 i que tots els nodes que processin aquesta petició SOAP han d'entendre aquesta capçalera, tal com s'indica amb l'atribut `mustUnderstand`. En cas que un node no l'entengui, retornarà un SOAP Fault i no es continuarà amb el processament del servei web.

A continuació trobem l'element `Body`, on s'envia el nom d'una acció, expressat com a cadena de caràcters, a un servei web que ha de retornar el seu preu.

```
POST /accions HTTP/1.1
Host: www.exemple.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
    <m:tempsMax value="10000" xmlns:m="http://www.exemple.org/accio"
        mustUnderstand="true"/>
  </soap:Header>
  <soap:Body xmlns:m="http://www.exemple.org/accio">
    <m:PreuAccio>
      <m:NomAccio>Inditex</m:NomAccio>
    </m:PreuAccio>
```

```
  </soap:Body>
```

La resposta viatja dins un missatge de resposta HTTP, també amb format `application/soap+xml`. El missatge de resposta SOAP també conté un element `Envelope` i dins trobem un element `Body` que porta la resposta del servei web que retorna el valor de l'acció, en aquest cas amb format de nombre real.

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope">

  <soap:Body xmlns:m="http://www.exemple.org/accio">
    <m:RespostaPreuAccio>
      <m:Preu>34.5</m:Preu>
    </m:RespostaPreuAccio>
  </soap:Body>

</soap:Envelope>
```

Ara que ja hem vist com viatgen tant la petició com la resposta d'un servei web SOAP dins els missatges del protocol HTTP, explicarem amb una mica més de detall com s'han de definir les operacions del servei, el format dels paràmetres i respostes del servei i també com s'associa aquest servei a un protocol de nivell d'aplicació concret, en aquest cas, HTTP.

Definir serveis web amb Web Services Description Language (WSDL)

Web Services Description Language (WSDL) 2.0 és una recomanació del W3C que descriu un llenguatge basat en XML per descriure serveis web. Per fer-ho es defineix un model abstracte de què ofereix el servei. A més, es defineixen els criteris de conformitat per als documents descrits en aquest llenguatge.

Sobre WSDL

World Wide Web Consortium (W3C) (2007). Web Services Description Language (WSDL) Versió 2.0, <https://www.w3.org/TR/wsdl20/>

Els elements principals d'un WSDL són:

- **description**. És l'element arrel. La resta d'elements són dins d'ell.
- **documentation**. És un element opcional que pot contenir una descripció llegible per part de les persones.

- **types**. Conté l'especificació dels tipus de dades intercanviats entre el client i el servei web. Per defecte, aquests tipus de dades es defineixen amb un XML Schema[8].

(8)**World Wide Web Consortium (W3C)** (2012). XML Schema Definition Language (XSD) 1.1, <https://www.w3.org/TR/xmlschema11-1/>

- **interface**. Descriu quines operacions té el servei i quins missatges s'intercanvien per a cada operació (input / output). També permet definir els possibles missatges d'error.

- **binding**. Defineix com s'accedeix al servei web a través de la xarxa. Normalment aquest element defineix com connectar al servei a través d'HTTP (tot i que no és l'única possibilitat).

- **service**. Defineix on es pot accedir al servei web a la xarxa. Normalment conté una URL on es pot trobar el servei.

A continuació, es pot veure un exemple d'WSDL on apareixen tots els elements descrits.

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    xmlns=           "http://www.w3.org/ns/wsdl"
    targetNamespace= "http://www.exemple.org/accio"
    xmlns:tns=       "http://www.exemple.org/accio"
    xmlns:stns =     "http://www.exemple.org/accio/esquema"
    xmlns:wsoap=     "http://www.w3.org/ns/wsdl/soap"
    xmlns:wsdlx=     "http://www.w3.org/ns/wsdl-extensions" >
  <documentation>
    This is the web service documentation.
  </documentation>
  <types>
    <xs:schema
        xmlns:xs=        "http://www.w3.org/2001/XMLSchema"
        targetNamespace= "http://www.exemple.org/accio/esquema"
        xmlns=           "http://www.exemple.org/accio/esquema">
      <xs:element name="PeticioPreuAccio" type="tipusPreuAccio"/>
      <xs:complexType name="tipusPreuAccio">
        <xs:sequence>
          <xs:element  name="NomAccio"   type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
      <xs:element name="RespostaPreuAccio" type="tipusRespostaPreuAccio"/>
      <xs:complexType name="tipusRespostaPreuAccio">
        <xs:sequence>
          <xs:element  name="Preu"   type="xs:float"/>
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
```

```
    </types>
    <interface  name = "PreuAccioInterface" >
      <operation name="PreuAccio"
              pattern="http://www.w3.org/ns/wsdl/in-out"
              style="http://www.w3.org/ns/wsdl/style/iri"
              wsdlx:safe = "true">
        <input    messageLabel="In"  element="stns:PeticioPreuAccio" />
        <output   messageLabel="Out" element="stns:RespostaPreuAccio" />
      </operation>
    </interface>
    <binding name="PreuAccioSOAPBinding"
            interface="tns:PreuAccioInterface"
            type="http://www.w3.org/ns/wsdl/soap"
            wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/"
            wsoap:mepDefault="http://www.w3.org/2003/05/soap/mep/request-response">
      <operation ref="tns:PreuAccio"/>
    </binding>
    <service
         name     ="ServeiPreuAccio"
         interface="tns:PreuAccioInterface">
       <endpoint name ="PreuAccioEndpoint"
              binding ="tns:PreuAccioSOAPBinding"
              address ="http://www.exemple.org/accio"/>
    </service>
  </description>
```

Els elements del WSDL estan relacionats entre ells. Per exemple, els atributs `interface` i `binding` de l'element `service` són els elements `interface` i `binding` (el que coincideix són els valors que es defineixen en l'atribut `name`), respectivament. L'element `operation` dins el `binding` (en aquest cas només n'hi ha una, però n'hi podria haver més) coincideix amb l'element `operation` definit a dins de l'element `interface`. Finalment, l'`input` i l'`output` definits dins l'element `operation`, han de ser tipus de dades definits dins l'element `types`.

Els valors `tns` i `stns` que hi ha al davant d'alguns dels noms corresponen a les referències als espais de noms on es defineixen els diferents elements. La referència als espais de noms la trobem normalment a l'element arrel del document XML, en aquest cas concret, a `description`.

A continuació trobem els espais de noms definits en aquest WSDL:

```
    xmlns=           "http://www.w3.org/ns/wsdl"
    xmlns:tns=       "http://www.exemple.org/accio"
    xmlns:stns =     "http://www.exemple.org/accio/esquema"
    xmlns:wsoap=     "http://www.w3.org/ns/wsdl/soap"
```

```
      xmlns:wsdlx=     "http://www.w3.org/ns/wsdl-extensions"
```

El primer és l'espai de noms per defecte d'aquest document, que fa referència a l'espai de noms de WSDL. Després de `xmlns` (que vol dir espai de noms XML, xml namespace), no hi ha cap nom qualificat. Això és perquè volem que els elements de l'espai de noms WSDL no necessitin cap prefix per referenciar-los dins el document. El següent que trobem és `xmlns:tns`, que es correspon amb l'espai de noms que definim en aquest document XML. Així, qualsevol element que pertany a aquest espai de noms, caldrà identificar-ho com a `tns:nomElement`. Passa el mateix amb `xmlns:stns`, que fa referència als tipus de dades que es defineixen en aquest document i que es fan servir en l'operació. Finalment, tenim altres dos espais de noms externs, `xmlns:wsoap` i `xmlns:wsdlx`, que fan referència a l'espai de noms de SOAP i al de les extensions de WSDL, respectivament. Alguns elements d'aquests espais de noms són necessaris per definir característiques específiques del servei.

Comparació entre serveis web SOAP i REST

Per comparar aquests dos tipus de serveis web, ens fixarem en algunes de les seves característiques principals:

- Format de peticions i respostes
- Protocol de nivell d'aplicació en el qual s'envien les dades
- Definició de les operacions del servei
- Seguretat

En primer lloc, parlarem del format de les peticions i les respostes en ambdós tipus de serveis. Mentre que SOAP defineix estructures complexes per enviar aquestes peticions i respostes, sempre en format XML, REST és molt més flexible. En REST, no hi ha un format únic per enviar les peticions o les respostes. De fet, en alguns casos no és ni tan sols necessari enviar cap missatge de petició i amb la mateixa URL d'accés al recurs es poden passar totes les dades necessàries. Vegem-ho amb un exemple:

```
Petició SOAP, per demanar els detalls de l'usuari amb ID 12345
(viatjarà en una comanda POST d'HTTP)
<soap:Envelope ...>
   <soap:body pb="http://www.exemple.org/agenda">
      <pb:GetDetallsUsuari>
   <pb:IDUsuari>12345</pb:IDUsuari>
      </pb:GetDetallsUsuari>
   </soap:Body>
</soap:Envelope>

Petició REST, per demanar les dades d'aquest mateix usuari
GET http://www.exemple.org/agenda/DetallsUsuari/12345 HTTP/1.1
```

Com es pot veure, la petició feta amb REST, on només és necessària l'URL, és molt més simple tant des del punt de vista de la quantitat de dades enviades com del processament d'aquestes que la de SOAP, on s'envien molts elements XML que no són rellevants per al servei.

Els missatges de resposta poden ser igual de complexos en els dos tipus de servei web, ja que REST pot enviar qualsevol tipus de fitxer, fins i tot en format XML. La diferència principal és que SOAP només accepta XML, i REST pot acceptar diferents tipus de dades.

Tots dos tipus de serveis web poden funcionar sobre el protocol HTTP. De fet REST només funciona sobre HTTP mentre que SOAP podria funcionar sobre altres protocols, tal com es defineix a SOAP Version 1.2 Part 2: Adjuncts, on es descriuen diferents estratègies d'enviament i recepció de missatges.

També hem vist que SOAP defineix les característiques dels seus serveis (format dels missatges d'enviament i resposta, tipus d'enviament utilitzat, protocol de nivell d'aplicació, adreça del servei, etc.) en un WSDL. Es podria fer això mateix amb REST? La resposta és que sí, a la versió 2.0 de WSDL. La manera de fer-ho és a l'element `binding`, on podem indicar que fem servir el tipus de `binding` corresponent a HTTP amb `http://www.w3.org/ns/wsdl/http` i com a operació del servei un mètode HTTP, com per exemple, GET (`whttp:method="GET"`). A continuació vegem un exemple de com podria ser un element `binding` que defineixi aquestes dades per a un servei REST model.

```
<wsdl:binding name="ServeiHTTPBinding"
     type="http://www.w3.org/ns/wsdl/http"
     interface="tns:InterficieServei">
    <wsdl:operation ref="tns:Servei" whttp:method="GET"/>
  </wsdl:binding>
```

Des del punt de vista de la seguretat, REST funciona sobre HTTP, que suporta autenticació bàsica i seguretat en la comunicació mitjançant Transport Layer Security (TLS)[9]. En canvi, SOAP té el WS-SECURITY[10], què és un estàndard que defineix tot un seguit de mecanismes de seguretat per a serveis web.

(9) **Internet Engineering Task Force (IETF)** (2018). The Transport Layer Security (TLS) Protocol Versió 1.3, <https://tools.ietf.org/html/rfc8446>.

(10) **Organization for the Advancement of Structured Information Standards (OASIS)** (2006). Web Services Security: SOAP Message Security 1.1 (WS-Security), <https://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf>.

Conclusions dels serveis REST i SOAP

En aquest apartat, hem vist que els serveis web SOAP i REST comparteixen algunes característiques, com ara l'ús d'HTTP com a protocol preferit per a l'enviament de les seves dades i d'altres força diferents com són els formats d'enviament de les peticions i respostes. Aquest últim punt ha fet que els serveis web basats en REST siguin els més utilitzats actualment. La flexibilitat que proporcionen amb peticions molt més simples o la no obligació de fer servir un format com XML han permès que aquest tipus de serveis s'hagin expandit i que sigui possible processar-los fàcilment amb qualsevol tipus de

dispositiu, especialment des de dispositius mòbils. A més, la possibilitat de rebre només una part de les dades i anar enllaçant amb la resta a través d'URL també els permet gestionar la transferència de dades més fàcilment.

2. Serveis REST amb llenguatge Java

El llenguatge Java proporciona una especificació anomenada JAX-RS, definida en el document JSR 311 (Sun Microsystems, 2008). Aquesta especificació defineix com el llenguatge Java dona suport als serveis web basats en REST. En aquesta especificació es descriu com accedir als recursos, definint anotacions Java[11], que apunten a classes de tipus recurs, a mètodes HTTP i a paràmetres i resultats a les crides als mètodes HTTP.

(11)Oracle Corporation (2014). The Java Tutorial, Annotations, <https://download.oracle.com/otn-pub/jcp/jaxrs-1.0-fr-eval-oth-JSpec/jaxrs-1.0-final-spec.pdf>.

Referència bibliogràfica

Sun Microsystems (2008). JAX-RS: Java™ API for RESTful Web Services. [en línia]: <https://download.oracle.com/otn-pub/jcp/jaxrs-1.0-fr-eval-oth-JSpec/jaxrs-1.0-final-spec.pdf>.

En la taula 4 es descriuen algunes de les anotacions utilitzades a JAX-RS i una petita descripció d'aquestes[12].

Les anotacions Java

Les anotacions en el llenguatge de programació Java tenen el format (@nom), on el símbol @ indica al compilador Java que el que trobarà és una anotació.

Una anotació és una metadada sintàctica que es pot aplicar a una classe, mètode o altres elements d'aquest llenguatge.

(12)Oracle Corporation (2013). Java Enterprise Edition 7 Specification APIs, <https://javaee.github.io/servlet-spec/downloads/servlet-4.0/servlet-4_0_FINAL.pdf>

Taula 4. Com enviar les peticions segons el mètode HTTP utilitzat

Anotació	Descripció
@Path	La classe arrel d'accés a un recurs amb llenguatge Java ha de tenir aquesta anotació per indicar en quina URL es pot trobar aquest recurs. Exemple: `@Path("/hola")` l'adreça base del nostre recurs serà `http://servidor/hola`. Aquesta anotació també es pot trobar dins dels mètodes, i aleshores el valor que hi hagi dins de l'anotació es concatenarà al @Path original.
@mètode_HTTP	Cada mètode HTTP té la seva pròpia anotació. Així, podem trobar `@POST, @GET`, etc. Aquesta anotació dins del codi del servei REST indica amb quin mètode HTTP respondrem a una operació concreta.
@tipusParam	Les anotacions acabades en Param ens indiquen com es poden rebre els paràmetres. Així, QueryParam indica que es reben a la part de la query de l'URL, després del símbol `?`, FormParam que es reben dins el cos del missatge amb format clau=valor o PathParam que són part de l'URL (s'identifiquen dins de claudàtors {} a l'anotació @Path corresponent a aquell mètode).
@Consumes, @Produces	Aquestes dues anotacions indiquen quines dades es reben (`@Consumes`) i quines es retornen (`@Produces`). Dintre d'aquestes anotacions s'ha de posar el tipus MIME corresponent. Exemples: `application/json` o `text/html`. La classe Java MediaType defineix alguns d'aquests tipus de dades.

Vegem un exemple de com es pot implementar un servei REST on es respon als mètodes GET i POST amb diferents configuracions de pas de paràmetres. Ens centrarem en aquests dos mètodes perquè són els més fàcils de provar directament des d'un navegador (tant des de la barra de direccions com des d'un formulari HTML).

El primer que trobem al servei són les classes JAX-RS que farem servir dins la nostra classe Java, que s'anomena «hola». Els noms dels paquets (*packages*, en anglès) Java comencen amb `javax.ws.rs`, que identifica la llibreria Java (`javax`) que implementa serveis web (`ws`) basats en REST (`rs`).

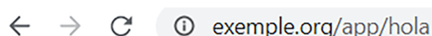
Les classes que incloem són algunes de les que es corresponen amb les anotacions que han aparegut a la Taula 4. Tenim el `Path`, els mètodes `GET` i `POST`, els tipus de paràmetres `FormParam`, `PathParam` i `QueryParam` i `Consumes` i `Produces`. També tenim la classe del subpaquet `core`, `MediaType`, que permet identificar els tipus de dades dintre de `Consumes` i `Produces`.

```
import javax.ws.rs.Consumes;
import javax.ws.rs.Produces;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.FormParam;
import javax.ws.rs.PathParam;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;
@Path("hola")
public class hola {
   /* Mètodes Java que responen a GET i a POST */
}
```

El primer mètode que implementem és el `GET`. Aquest mètode no rep cap dada i retorna (`Produces`) dades de tipus `MediaType.TEXT_HTML`. La implementació del mètode és molt senzilla, ja que retorna una pàgina HTML que mostra la paraula GET. L'URL per cridar aquest servei podria ser `http://exemple.org/app/hola`. En la Figura 5 trobem un exemple de resposta.

```
@GET
@Produces(MediaType.TEXT_HTML)
public String getHtml() {
    return "<html><head/><body>GET</body></html>";
}
```

Figura 5. Exemple de resposta de l'operació getHtml en un navegador



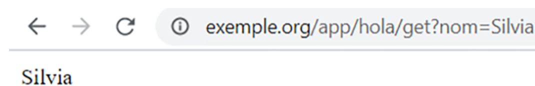
GET

El segon mètode que implementem també és el `GET`. No és possible tenir dos mètodes que responguin a GET amb la mateixa URL, així que en aquest hem afegit el path «get». A més, passem un paràmetre per la secció query de l'URL. Aquest paràmetre es diu «nom» i s'associa a una variable de tipus String que també s'anomena nom. També retorna (`Produces`) dades de tipus `MediaType.TEXT_HTML`.

La implementació del mètode és molt senzilla, ja que retorna una pàgina HTML que mostra el nom de l'usuari que s'ha passat com a paràmetre. L'URL per cridar aquest servei podria ser `http://exemple.org/app/hola/get?nom=Silvia`. A la Figura 6 trobem un exemple de resposta.

```
@GET
@Path("/get")
@Produces(MediaType.TEXT_HTML)
public String getQuery(@QueryParam("nom") String nom) {
    return "<html><head/><body>"+nom+"</body></html>";
}
```

Figura 6. Exemple de resposta de l'operació getQuery en un navegador

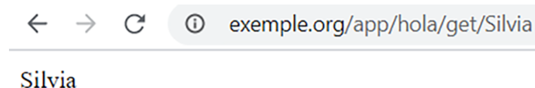


Silvia

El tercer mètode que implementem també és el `GET`. En aquest cas, a més d'afegir el path «get», indiquem que hi ha un paràmetre que es passa directament per l'URL (PathParam), que dona com a path resultant «get/{nom}». Aquest paràmetre es diu nom i s'associa a una variable de tipus String que també s'anomena nom. També retorna (`Produces`) dades de tipus `MediaType.TEXT_HTML`. La implementació del mètode és molt senzilla, ja que retorna una pàgina HTML que mostra el nom de l'usuari que s'ha passat com a paràmetre. L'URL per cridar aquest servei podria ser `http://exemple.org/app/hola/get/Silvia`. En la Figura 7 trobem un exemple de resposta.

```
@GET
    @Path("/get/{nom}")
    @Produces(MediaType.TEXT_HTML)
    public String getPath(@PathParam("nom") String nom) {
        return "<html><head/><body>"+nom+"</body></html>";
    }
```

Figura 7. Exemple de resposta de l'operació getPath en un navegador



Silvia

L'últim mètode que implementem és el `POST`. En aquest cas, es reben dades en el missatge de petició (`Consumes`) i també se'n retornen (`Produces`). Les dades es passen com a dades de formulari HTML (`application/x-www-form-urlencoded`) i es retornen com a `MediaType.TEXT_HTML`. En el formulari es passa un paràmetre que s'anomena nom com a FormParam. Aquest paràmetre s'associa a una variable de tipus String que també s'anomena nom. La implementació del mètode és molt senzilla, ja que retorna una pàgina HTML que mostra el nom de l'usuari que s'ha passat com a paràmetre. L'URL

per cridar aquest servei podria ser `http://exemple.org/app/hola`, però en aquest cas s'ha d'enviar la petició en un missatge POST d'HTTP. El servidor sap a quin mètode del servei ha de cridar a partir de l'URL i el mètode HTTP utilitzat, per això té la mateixa URL d'accés al recurs que el primer mètode GET que hem vist.

```
@POST
    @Consumes("application/x-www-form-urlencoded")
    @Produces(MediaType.TEXT_HTML)
    public String postHtml(@FormParam("nom") String nom) {
        return "<html><head/><body>"+nom+"</body></html>";
    }
```

Finalment, posem un extracte d'un formulari HTML que permet connectar-se amb el servei REST que hem implementat en aquest apartat.

```
<form action="http://exemple.org/app/hola" method="POST"
     enctype="application/x-www-form-urlencoded">
  <input type="text" name="nom"/>
  <input type="submit" value="Enviar"/>
```

En aquest formulari hem d'indicar l'URL del servei (atribut *action*, `http://exemple.org/app/hola`), el mètode d'enviament (atribut *method*, en aquest cas ha de ser POST), el tipus de codificació de les dades (atribut *enctype*, `application/x-www-form-urlencoded`) i dins del formulari hi ha d'haver un camp que es digui `nom`. En aquest cas, és un camp de tipus text i l'atribut *name* té el valor `nom`, què és el valor esperat pel servei. Si no ho definim exactament així, el missatge HTTP enviat no coincidirà amb el que espera el servei i la petició fallarà. A les figures 8 i 9 trobem el formulari i la resposta donada pel servei.

Figura 8. Formulari

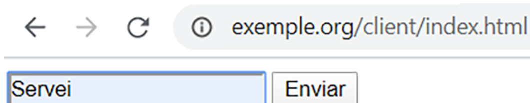
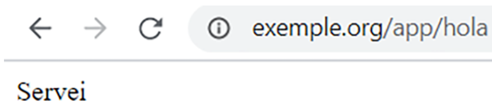


Figura 9. Exemple de resposta de l'operació postHtml en un navegador



Un possible error que es podria produir en la crida amb POST és que el camp de formulari no es digui `nom`. En aquest cas el servei no trobarà el valor de la variable corresponent i ens retornarà un valor *null* a la pàgina de resposta. Un altre problema que es podria produir és que, si posem com a mètode GET en comptes de POST, es cridarà a la primera operació GET del nostre servei i ens mostrarà el missatge GET en comptes de mostrar les dades que hem

escrit al formulari. Un altre possible error és que cridem al servei amb una URL no definida (p. ex., `http:// exemple.org/app/hola/hola3`) i això ens donaria un error HTTP de recurs no trobat (Codi 404). Finalment, si cridem al servei amb una URL que no correspon amb el mètode HTTP, l'error que ens donaria és mètode no suportat (Codi 405).

2.1. Operacions pròpies del servidor

Hem mostrat un exemple molt simple de com implementar un servei web REST amb llenguatge Java. En aquest apartat veurem què més es pot fer amb llenguatge Java per tenir un servei web REST que respongui a la resta de mètodes HTTP, quins altres tipus de format de dades de peticions i respostes podem tenir, etc.

Cal destacar que el que hem vist fins ara és independent del tipus de servidor amb el qual es treballi, ja que es tracta d'una especificació Java que els diferents fabricants poden implementar a la seva manera.

2.1.1. Creació del servei

Per crear un servei amb JAX-RS, cal crear una classe que amplii la classe abstracta `Application` (dins el Package `javax.ws.rs.core`). En aquesta classe s'ha de definir el recurs arrel del servei i enregistrar les classes que responen als diferents mètodes HTTP. Per al servei que s'ha creat en l'apartat 2, la classe derivada d'`Application` podria ser la següent:

```
import java.util.Set;
import javax.ws.rs.core.Application;
@javax.ws.rs.ApplicationPath("app")
public class ApplicationConfig extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> resources = new java.util.HashSet<>();
        resources.add(hola.class);
        return resources;
    }
}
```

En aquesta classe s'està definint l'arrel del servei (`app`) i s'afegeix la classe `hola.class`, que implementa les operacions del servei de l'apartat 2, als recursos disponibles en aquest servei.

2.1.2. Definició de les operacions

Ja hem vist en l'apartat 2 com es defineixen les operacions d'un servei web REST que utilitza JAX-RS. Aquí detallarem tots els components que hi apareixen i la seva funció, com també les alternatives que ens podem trobar.

```
@METODE_HTTP
@Path("/URL_Access/{PATH_PARAM}")
@Consumes(TIPUS_MIME_PETICIO)
@Produces(TIPUS_MIME_RESPOSTA)
public String nomMetode(@XParam("nomParam") String nomParam) {
    return <MISSATGE EN FORMAT TIPUS_MIME_RESPOSTA>;
}
```

La forma més general que hi ha de respondre a una petició REST amb JAX-RS és la de tenir una anotació de mètode HTTP. A continuació, podem tenir una anotació amb un `@Path` específic per a aquest mètode, que pot contenir (o no, és opcional), un o diversos paràmetres de tipus `@PathParam`, encerclats per «{}». Poden aparèixer en qualsevol posició dins del path, com ara `{param1}/path/{param2}`. Cada paràmetre d'aquest tipus apareixerà com a paràmetre del mètode, i serà del tipus `@PathParam`. També podem tenir paràmetres del tipus `@QueryParam` (els que s'envien dins la secció query de l'URL) o `@FormParam`, en cas que el tipus MIME de la petició fos `application/x-www-form-urlencoded` o `multipart/form-data`. Cal dir que podríem arribar a trobar els tres tipus de paràmetres en un mateix mètode.

A més dels paràmetres esmentats, també podem tenir `@Consumes` i `@Produces` amb tipus MIME que representin dades estructurades com ara `application/xml` o `application/json`. Per tractar amb dades estructurades, cal tenir classes Java de suport que tinguin els elements corresponents a les dades que s'han de rebre. Hi ha diferents alternatives per implementar aquestes classes de suport de manera més o menys automàtica segons les llibreries disponibles i les necessitats del servei que s'ha d'implementar (emmagatzemar aquestes dades en fitxers, bases de dades, etc.). A continuació, presentarem un exemple d'implementació d'un mètode que rep i retorna dades XML i un altre que rep i retorna dades JSON. Tots dos fan servir la mateixa classe Java de suport, ja que tenen la mateixa estructura.

Hem anomenat `Item` a aquesta classe de suport. Conté una sèrie d'anotacions Java perquè es faci el tractament automàtic del llenguatge XML. Les anotacions són `@XmlRootElement`, que identifica l'element arrel del document, `@XmlAccessorType`, que permet indicar com es fa la connexió entre els elements de la classe Java i el document XML, i `@XmlRootElement`, que identifica la resta d'elements. Hi ha dos camps, un anomenat `id`, de tipus enter

i un altre anomenat `name`, de tipus cadena de caràcters. Abans del codi de la classe `Item`, vegem un exemple de fitxer XML i un altre de fitxer JSON amb el format de les dades i uns valors possibles.

```
<?xml version="1.0" encoding="UTF-8"?>
<item>
    <id>4</id>
    <name>Camera</name>
</item>
{
    "id": 4,
    "name": "Camera"
}
```

La classe `Item` conté les anotacions ja descrites i alguns mètodes de suport per modificar i consultar els camps de la classe.

```
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlElement;
@XmlRootElement(name="item")
@XmlAccessorType(XmlAccessType.FIELD)
public class Item {
    @XmlElement
    private int id;
    @XmlElement
    private String name;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @Override
    public String toString(){
        return "Id: " + this.id + " Nom: " + this.name;
    }
}
```

A continuació, posem dos exemples d'implementació amb el mètode HTTP PUT on rebem i enviem dades tant en format XML com JSON.

En el primer mètode, el que treballa amb XML, cal destacar que ara rebem un paràmetre de tipus `Item` al mètode, que converteix automàticament les dades XML en camps de la classe Java. En aquest cas, el resultat en XML es genera dins del mateix mètode, però aquí podríem afegir tot tipus de tractament de dades XML, emmagatzematge en altres suports (fitxer a disc, base de dades, reenviament a altres serveis, etc.).

```
@PUT
@Path("/xml")
@Consumes("application/xml")
@Produces("application/xml")
public String putXML(Item i) {
    String resultat;
    resultat = "<?xml version=\"1.0\" encoding=\"UTF-8\"?><item><id>"
               +(i.getId()+1)+"</id><name> "
               + i.getName() +" Perez</name></item>";
    return resultat;
}
```

En el segon mètode, el que treballa amb JSON, l'estratègia és una mica diferent. Com a paràmetre rebem un `java.io.InputStream`, que conté un canal d'accés a les dades JSON enviades per l'aplicació client. Amb aquest `InputStream`, fent servir una classe Java de suport de JSON (`com.google.gson.Gson`)[13], aconseguim un objecte de tipus `Item`, que després modifiquem per enviar la resposta al client. El mateix que s'ha comentat per al cas XML (bases de dades, disc, etc.) es podria fer aquí, fent el tractament corresponent de les dades JSON. Hi ha moltes llibreries Java de suport a la creació, la lectura i l'escriptura de dades JSON. S'ha triat aquesta per la seva simplicitat d'ús.

(13) **Gson Java library**, <https://github.com/google/gson> [Data de consulta: març de 2020]

```
@PUT
@Path("/json")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public String putJSON(java.io.InputStream is) {
    Item test;
    com.google.gson.Gson gson = new com.google.gson.Gson();
    java.io.BufferedReader reader = new java.io.BufferedReader
                                (new java.io.InputStreamReader(is));
    test = gson.fromJson(reader, Item.class);
    test.setId (test.getId()+1);
    test.setName (test.getName() + " photo");
    return gson.toJson(test);
```

```
}
```

2.1.3. Enviament de la petició

L'enviament de la petició es farà dins d'un missatge HTTP. A continuació veiem un exemple d'invocació als mètodes PUT implementats en l'apartat anterior amb XML i JSON, respectivament. Després de la línia de petició del servei trobem les capçaleres HTTP que identifiquen el tipus de contingut que s'envia, la seva longitud, el tipus de dades que acceptem per a la resposta i el nom del host al qual ens volem connectar.

```
PUT /app/hola/xml HTTP/1.1
Accept: application/xml
Content-Type: application/xml
Content-Length: 95
Host: www.exemple.com
<?xml version="1.0" encoding="UTF-8"?>
<item>
    <id>4</id>
    <name>Camera</name>
</item>
```

El segon missatge de petició conté les mateixes dades, però ara per al format JSON d'enviament i recepció.

```
PUT /app/hola/json HTTP/1.1
Accept: application/json
Content-Type: application/json
Content-Length: 35
Host: www.exemple.com
{
    "id": 4,
    "name": "Camera"
}
```

2.1.4. Enviament de la resposta

L'enviament de la resposta es farà també dins d'un missatge HTTP. A continuació veiem un exemple de resposta al mètode PUT amb XML i JSON, respectivament. Després de la línia de resposta del servei trobem les capçaleres HTTP que ens identifiquen el tipus de contingut que s'envia i la seva longitud.

```
HTTP/1.1 200 OK
Content-Length: 95
Content-Type: application/xml
<?xml version="1.0" encoding="UTF-8"?>
<item>
```

```
    <id>5</id>
    <name>Camera photo</name>
</item>
```

El segon missatge de petició conté les mateixes dades, però ara per al format JSON d'enviament i recepció.

```
HTTP/1.1 200 OK
Content-Length: 41
Content-Type: application/json
{
    "id": 5,
    "name": "Camera photo"
}
```

2.2. Operacions pròpies del client

Invocar un servei web basat en REST des d'una aplicació client implica obrir una connexió HTTP amb el servei i enviar una petició en el format esperat pel mateix (mètode HTTP, capçaleres i paràmetres o dades). Això, ho podríem fer amb una connexió via *sockets* o fent servir classes de suport disponibles en llenguatges de programació com ara Java.

Referència bibliogràfica

March Hermo, M. I. (2020). *Programació de sockets en Java*. Barcelona: UOC.

L'aplicació client pot ser de qualsevol tipus: una aplicació d'escriptori, una aplicació web i, fins i tot, una aplicació mòbil. Els requisits que ha de complir és que sàpiga enviar el missatge de petició HTTP amb les dades requerides i després sigui capaç d'entendre la resposta enviada pel servidor. La complexitat d'aquesta aplicació client dependrà sobretot del format en què s'enviïn les dades en aquestes peticions i respostes. Així, com més complex sigui el format de les dades (p. ex., enviament de fitxers en formats XML o JSON), més complicat serà tractar-les. Tot i això, l'existència de llibreries tant per fer les connexions com per tractar les dades facilita molt la feina d'implementació.

A la resta de l'apartat s'explica amb una mica més de detall com implementar dues aplicacions client, una aplicació Java d'escriptori i una aplicació web basada en J2EE.

Referència bibliogràfica

Java Community Process (2006). JSR 88: Java™ EE (J2EE) Application Deployment, <https://www.jcp.org/en/jsr/detail?id=88&showPrint>

2.2.1. Programació d'una aplicació client de servei REST

En aquesta apartat s'explicarà com implementar un client de prova per al nostre servei web basat en REST, però l'estratègia seria molt similar per connectar-se a un altre servei REST ja existent. Primer veurem com fer-ho des d'una aplicació web amb un servlet (`javax.servlet.http.HttpServlet`) i després amb una aplicació Java d'escriptori.

Referència bibliogràfica

Oracle Corporation (2017). Java Servlets Specification v4.0, <https://javaee.github.io/servlet-spec/downloads/servlet-4.0/servlet-4_0_FINAL.pdf>

Aplicació web basada en J2EE

Un servlet és una classe Java que pot respondre a mètodes HTTP fets per un navegador. En aquest sentit, és una mica com un servei web REST, però limitat a GET i a POST i orientat a un usuari final. Dona suport a la recepció de dades de formularis HTML i fitxers. També té accés a la petició HTTP que es rep des del client (en aquest cas, un navegador) i a la resposta HTTP que s'ha de retornar, que ha de ser en llenguatge HTML perquè el navegador ho entengui.

La declaració d'una classe de tipus servlet amb suport per a serveis web REST podria ser com segueix:

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.core.MediaType;
@WebServlet(name = "Client", urlPatterns = {"/Client"})
public class Client extends HttpServlet {
    /* Mètodes del servlet per contactar amb el servei */
}
```

En el codi que veiem hi ha importació de totes les classes Java necessàries i la declaració de la classe, que deriva de `HttpServlet`. Trobem també una anotació Java que ens indica que això és un servlet (`@WebServlet`) i l'URL d'accés a aquest des de l'aplicació web.

```
/* Extracte del codi de connexió amb el servidor */
Client client;
String REST_URL = "http://www.exemple.org/app/hola";
String resultat;
try {
    client   = ClientBuilder.newClient();
    resultat = client.target(REST_URL)
               .path("json")
               .request(MediaType.APPLICATION_JSON)
               .put(Entity.json("{\"id\":4,\"name\":\"camera\"}"),
                                 String.class);
}
```

Aplicació Java d'escriptori

El codi de l'aplicació d'escriptori per connectar amb un servei web basat en REST és molt semblant al codi que hem implementat des d'una aplicació web. La diferència més important és que hem de crear una classe Java amb el mètode `public static void main(String[] args)` per a poder-ho executar.

Cal destacar que l'únic problema que podem trobar en aquest cas és el de disposar de totes les llibreries Java que donin suport a una aplicació client de servei web basat en REST, ja que la llibreria que conté les classes `javax.ws.rs.client` té força dependències amb altres llibreries Java que qualsevol entorn de programació pot resoldre sense massa problema. No posem cap versió concreta, ja que el llenguatge Java evoluciona tan ràpidament que de seguida quedarien obsoletes.

```
/* Codi per fer la connexió des d'una aplicació Java d'escriptori */
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.core.MediaType;
public class ClientJavaREST {
    public static void main(String[] args) {
        Client client;
        String REST_SERVICE_URL = "http://www.exemple.org/app/hola";
        String resultado;
        try {
            client = ClientBuilder.newClient();
            resultado = client.target(REST_SERVICE_URL)
                 .path("json")
                 .request(MediaType.APPLICATION_JSON)
                 .put(Entity.json("{\"id\":3,\"name\":\"camera\"}"),
                      String.class);
            /* Mostrem el resultat. Hem hagut de rebre dades en format
               JSON, que podríem tractar convenientment */
            System.out.println (resultado);
        } catch (Exception e)
        {
            System.out.println (e.getMessage());
        }
    }
}
```

3. Serveis REST amb altres llenguatges de programació (Node.js, Python)

3.1. Node.js

Node.js és un entorn de servidor de codi obert que funciona sobre diferents sistemes operatius i que fa servir el llenguatge Javascript en el servidor (l'ús habitual de Javascript es troba en el cantó client, inclòs en el navegador).

Sobre Node.js

OpenJS Foundation, Node.js, <https://nodejs.org/es/>. [Data de consulta: març de 2020].

Per poder-lo fer servir com a servidor REST, cal utilitzar el que es coneix com la pila d'eines MEAN, que inclou les eines MongoDB, ExpressJS, Angular i Node.js (MEAN)[14]. Totes aquestes eines es basen en llenguatge Javascript. Com a mínim hem de tenir Node.js i ExpressJs, però per tenir un servei REST complet és recomanable tenir-les totes. A continuació les descrivim breument.

(14)**IBM, MEAN (MongoDB Express Angular Node)** stack explained, <https://www.ibm.com/cloud/learn/mean-stack-explained> [Data de consulta: març de 2020]

MongoDB és una base de dades no relacional que permet emmagatzemar dades en format JSON.

ExpressJS permet que Node.js es pugui comportar com un servidor REST. Es tracta d'un entorn flexible i que proporciona tot un seguit de característiques per donar suport a aplicacions web i mòbils, incloent-hi la creació d'API.

Angular permet desenvolupar aplicacions web, mòbils i d'escriptori fent servir llenguatge Javascript.

Referències bibliogràfiques

MongoDB Inc., MongoDB, <https://www.mongodb.com/es>. [Data de consulta: març de 2020].

StrongLoop, Inc., ExpressJS, <https://expressjs.com/es/>. [Data de consulta: març de 2020].

Google, Angular, <https://angular.io/>. [Data de consulta: març de 2020].

En la subapartat següent expliquem amb una mica més de detall com implementar un servei web bàsic basat en REST amb Node.js.

3.1.1. Desenvolupament d'una API REST amb Node.js i ExpressJS

Per desenvolupar un servei REST amb Node.js cal instal·lar-lo des del seu web oficial. Un cop instal·lat, cal instal·lar també ExpressJS per poder oferir les diferents operacions via web.

Un cop aquestes dues eines estan instal·lades, implementar una API bàsica és força senzill.

El primer que cal fer és declarar les dependències que tindrà la nostra API. Això es pot fer amb el codi Javascript següent:

```
/* Declaració de dependències i variable app */
var express = require('express');
var bodyParser = require('body-parser');
```

```
var app = express();
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
```

Aquí s'indica que hi ha dues dependències, `express` i `body-parser`, que farem servir després. A continuació declarem la variable `app` que serà l'aplicació web que farà servir express.

En el tros de codi següent veiem que l'aplicació fa servir algunes característiques de `body-parser`. A més, declara la variable `port` que defineix el port on escoltarà aquest servidor. Si no se n'ha definit un altre en els fitxers de configuració, el port que es farà servir és el 8080. A continuació, es declara una constant que contindrà l'encaminador que proporciona ExpressJS. Amb aquest encaminador es defineixen les diferents operacions que proporciona el servei.

La primera de totes és respondre a GET en l'URL base del servei i retornar les dades `{"missatge":"API respon a GET!"}`. La següent és respondre a POST en l'URL base del servei i retornar les dades `{"missatge":"API respon a POST!"}`.

```
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
var port = process.env.PORT || 8080;
const router = express.Router();
router.get('/', function(req, res) {
   res.json({ message: 'API respon a GET!' });
});
router.post('/', function(req, res) {
   res.json({ message: 'API respon a POST!' });
   res.end("end");
});
```

A continuació veiem una operació que ofereix aquesta API. Donada una lletra, retorna com a resultat la mateix lletra en majúscules. Si ja és majúscula, retorna la mateixa lletra que ha escrit l'usuari. L'URL d'accés (que s'ha de concatenar a l'URL del servei) té la forma `/lletres/a` i el resultat que retorna és `{"resultat":"A"}`.

Finalment, es posa en marxa l'aplicació web a l'URL `/api` que escoltarà pel port indicat anteriorment. Així, per accedir a la funcionalitat de l'operació lletres amb aquesta aplicació hauríem de fer servir l'URL següent: `http://exemple.org/api/lletres/a`. L'URL base del servei seria `http://exemple.org/api` i donaria una resposta diferent als mètodes GET i POST, tal com hem vist anteriorment.

```
router.route('/lletres/:lletra').get((req, res) => {
   res.json({resultat: req.params.lletra.toUpperCase()})
});
app.use('/api', router);
app.listen(port);
console.log('Escoltant al port ' + port);
```

El servei es pot posar en marxa des d'una línia d'ordres, indicant el nom de l'ordre `node` i a continuació el nom del fitxer amb extensió `.js` on tinguem el nostre codi. Podem veure un exemple just a sota.

```
linia_comandes> node nom_fitxer.js
```

Codi complet servei REST amb Node.js

```
/* Codi per fer una API REST bàsica amb Node.js */
var express = require('express');
var bodyParser = require('body-parser');
var app = express();
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
var port = process.env.PORT || 8080;
const router = express.Router();
router.get('/', function(req, res) {
   res.json({ message: 'API respon a GET!' });
});
router.post('/', function(req, res) {
   res.json({ message: 'API respon a POST!' });
   res.end("end");
});
router.route('/lletres/:lletra').get((req, res) => {
   res.json({resultat: req.params.lletra.toUpperCase()})
});
app.use('/api', router);
app.listen(port);
console.log('Escoltant al port ' + port);
```

3.2. Python

Python és un llenguatge de programació interpretat (necessita un intèrpret per poder-se executar). Permet implementar molts tipus d'aplicacions, fins i tot serveis basats en REST. Per fer-ho, necessita un entorn web, com ara Flask o Django. En els exemples següents farem servir Flask per al suport web.

Flask és un *framework* web escrit en llenguatge Python. Se l'anomena microframework perquè no necessita eines o llibreries extra per al seu funcionament. Tot i això, es poden afegir extensions fàcilment per tal de proporcionar funcionalitat extra.

Django és un altre *framework* web escrit en Python. És gratuït i de codi obert, i segueix el model arquitectural Model-Template-View (MTV).

En la subapartat següent expliquem amb una mica més de detall com implementar un servei web bàsic basat en REST amb Python.

Referències bibliogràfiques

Python Software Foundation, Python. [en línia]: <https://www.python.org/> [Data de consulta: març de 2020]

Pallets. *Flask*. [en línia]: <https://flask.palletsprojects.com/en/1.1.x/> [Data de consulta: març de 2020]

Django Software Foundation. *Django*. [en línia]: <https://www.djangoproject.com/> [Data de consulta: març de 2020]

Django Software Foundation. *Model-Template-View* (MTV) *architectural model in Django*. [en línia]: <https://docs.djangoproject.com/en/3.0/ faq/general/> [Data de consulta: març de 2020]

3.2.1. Desenvolupament d'una API REST amb Python i Flask

Per desenvolupar un servei REST amb Python cal instal·lar-lo des del seu web oficial. Un cop instal·lat, cal instal·lar també Flask per poder oferir les diferents operacions via web.

Un cop aquestes dues eines estan instal·lades, implementar una API bàsica és força senzill.

El primer que cal fer és declarar les dependències que tindrà la nostra API. Això es pot fer amb el codi Python següent:

```
from flask import Flask, jsonify, request
app = Flask(__name__)
comptes = [
        {'nom': "Anna", 'balanc':450.0},
        {'nom': "Pau", 'balanc':250.0},
    ]
```

Les dependències que tenim són `Flask`, `jsonify` i `request`. Totes aquestes dependències provenen de `Flask`. Això s'indica a `from Flask`. Després declarem la variable `app`, que contindrà la nostra aplicació web. Finalment, declarem la variable `comptes`, que contindrà les dades d'uns comptes bancaris en format JSON.

Després, declarem els mètodes de la nostra API. Primer implementarem les funcions que responen a GET. En aquest cas en tenim dues, una que ens retornarà tots els comptes que tenim a la nostra aplicació i una altra que ens permetrà saber les dades d'un compte a partir del seu identificador.

Per definir les operacions, fem servir la variable `app` declarada anteriorment i cridem el mètode `route`, on definim l'URL d'accés a l'operació i el mètode HTTP al qual respondrà. En el primer cas, l'URL és `/comptes` i en el segon cas, l'URL és `/comptes/<id>`, on `id` és un paràmetre de l'URL que indica l'identificador del compte que es vol consultar. Després cal implementar el codi de l'operació, que retorna les dades demanades per l'usuari.

```
@app.route("/comptes", methods=["GET"])
def getComptes():
    return jsonify(comptes)
@app.route("/comptes/<id>", methods=["GET"])
def getCompte(id):
    id = int(id) - 1
    return jsonify(comptes[id])
```

L'última operació respon al mètode POST i ens permet donar d'alta un nou compte. L'URL d'accés serà `/compte` i les dades es passaran en format JSON al cos del missatge HTTP. Finalment, l'aplicació web es posa en marxa al port 8080.

```
@app.route("/compte", methods=["POST"])
def addAccount():
    nom = request.json['nom']
    balanc = request.json['balanc']
    dades = {'nom': nom, 'balanc': balanc}
    comptes.append(dades)
    return jsonify(dades)
if __name__=='__main__':
    app.run(port=8080)
```

El servei es pot posar en marxa des d'una línia d'ordres, indicant el nom de l'ordre `python3` (corresponent a la versió 3 de Python) i a continuació el nom del fitxer amb extensió `.py` on tinguem el nostre codi. Podem veure un exemple just a sota.

```
linia_comandes> python3 nom_fitxer.py
```

Codi complet del servei Python

```
/* Codi per fer una API REST bàsica amb Python */
from flask import Flask, jsonify, request
app = Flask(__name__)
```

```
comptes = [
        {'nom': "Anna", 'balanc':450.0},
        {'nom': "Pau", 'balanc':250.0},
    ]
@app.route("/comptes", methods=["GET"])
def getComptes():
    return jsonify(comptes)
@app.route("/comptes/<id>", methods=["GET"])
def getCompte(id):
    id = int(id) - 1
    return jsonify(comptes[id])
@app.route("/compte", methods=["POST"])
def addAccount():
    nom = request.json['nom']
    balanc = request.json['balanc']
    dades = {'nom': nom, 'balanc': balanc}
    comptes.append(dades)
    return jsonify(dades)
if __name__=='__main__':
    app.run(port=8080)
```

Resum

El mòdul ha presentat els serveis web basats en REST (REpresentational State Transfer), que defineixen una arquitectura de com s'ha d'accedir i manipular recursos remots mitjançant els mètodes HTTP (POST, GET, PUT, PATCH i DELETE).

Hem vist també la comparació entre els serveis web basats en SOAP i REST, incidint en les característiques de cada un i explicant algunes de les diferències principals entre ells.

Finalment, s'ha parlat de com desenvolupar serveis web en llenguatge Java, donant també algunes pautes de com es podria fer amb altres llenguatges de programació com són Node.js i Python.

Bibliografia

Django Software Foundation. *Django*. [en línia]: <https://www.djangoproject.com/> [Data de consulta: març de 2020].

Django Software Foundation. *Model-Template-View (MTV) architectural model in Django*. [en línia]: <https://docs.djangoproject.com/en/3.0/faq/general/> [Data de consulta: març de 2020].

European Computer Manufacturers Association (ECMA) (2017). The JSON (JavaScript Object Notation) Data Interchange Syntax ECMA – 404 Standard. [en línia]: <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>.

Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. [en línia]: <https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf>.

Google. *Angular*. [en línia]: <https://angular.io/> [Data de consulta: març de 2020].

Gson Java library. [en línia]: https://github.com/google/gson [Data de consulta: març de 2020].

IBM. *MEAN (MongoDB Express Angular Node) stack explained*. [en línia]: <https://www.ibm.com/cloud/learn/mean-stack-explained> [Data de consulta: març de 2020].

Internet Engineering Task Force (IETF) (2005). Uniform Resource Identifier (URI): Generic Syntax, RFC 3986. [en línia]: <https://tools.ietf.org/html/rfc3986>.

Internet Engineering Task Force (IETF) (2014). Hypertext Transfer Protocol (HTTP/1.1), RFC 7230 a 7235. [en línia]: <https://tools.ietf.org/html/rfc7230> i <https://tools.ietf.org/html/rfc7235>.

Internet Engineering Task Force (IETF) (2015). Returning Values from Forms: multipart/form-data, RFC 7578. [en línia]: <https://tools.ietf.org/html/rfc7578>.

Internet Engineering Task Force (IETF) (2018). The Transport Layer Security (TLS) Protocol Versió 1.3. [en línia]: <https://tools.ietf.org/html/rfc8446>

Introducing JSON. [en línia]: <https://www.json.org/json-en.html> [Data de consulta: març de 2020].

Java Community Process (2006). JSR 88: Java™ EE (J2EE) Application Deployment. [en línia]: <https://www.jcp.org/en/jsr/detail?id=88&showPrint>.

March Hermo, M. I. (2020). *Programació de sockets en Java*. Barcelona: UOC.

MongoDB Inc. *MongoDB*. [en línia]: <https://www.mongodb.com/es> [Data de consulta: març de 2020].

OpenJS Foundation *Node.js* [en línia]: <https://nodejs.org/es/> [Data de consulta: març de 2020].

Oracle Corporation (2013). Java Enterprise Edition 7 Specification APIs. [en línia]: <https://docs.oracle.com/javaee/7/api/overview-summary.html>

Oracle Corporation (2014). *The Java Tutorial, Annotations*. [en línia]: <https://docs.oracle.com/javase/tutorial/java/annotations/index.html>.

Oracle Corporation (2017). Java Servlets Specification v4.0. [en línia]: <https://javaee.github.io/servlet-spec/downloads/servlet-4.0/servlet-4_0_FINAL.pdf>.

Organization for the Advancement of Structured Information Standards (OASIS) (2006). Web Services Security: SOAP Message Security 1.1 (WS-Security). [en línia]: <https://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf>.

Pallets *Flask*. [en línia]: <https://flask.palletsprojects.com/en/1.1.x/> [Data de consulta: març de 2020].

Python Software Foundation *Python*. [en línia]: <https://www.python.org/> [Data de consulta: març de 2020].

StrongLoop, Inc. *ExpressJS*. [en línia]: <https://expressjs.com/es/> [Data de consulta: març de 2020].

Sun Microsystems (2008). JAX-RS: Java™ API for RESTful Web Services. [en línia]: <https://download.oracle.com/otn-pub/jcp/jaxrs-1.0-fr-eval-oth-JSpec/jaxrs-1.0-final-spec.pdf>.

World Wide Web Consortium (W3C) [en línia]: <https://www.w3.org/> [Data de consulta: març de 2020].

World Wide Web Consortium (W3C) (2006). Extensible Markup Language (XML) 1.1 (Second Edition). [en línia]: <https://www.w3.org/TR/xml11>.

World Wide Web Consortium (W3C) (2007). Simple Object Access Protocol (SOAP) Versió 1.2. [en línia]: <https://www.w3.org/TR/soap12/>.

World Wide Web Consortium (W3C) (2007). SOAP Version 1.2 Part 2: Adjuncts (Second Edition). [en línia]: <https://www.w3.org/TR/soap12-part2/>.

World Wide Web Consortium (W3C) (2007). Web Services Description Language (WSDL) Versió 2.0. [en línia]: <https://www.w3.org/TR/wsdl20/>.

World Wide Web Consortium (W3C) (2012). XML Schema Definition Language (XSD) 1.1. [en línia]: <https://www.w3.org/TR/xmlschema11-1/>.