

Integración de aplicaciones

Tema 3. SOA

© 2020 Javier Esparza Peidro - jesparza@dsic.upv.es

Contenido

- Introducción
- Orientación a servicios
- Diseño orientado a servicios
- Servicios Web (SOAP)
- Servicios RESTful

Introducción

- Integrar con bases de datos compartidas es un anti-patrón
 - Fuerte acoplamiento entre aplicaciones (evolución)
 - Cuello de botella (escalabilidad)
 - Punto de fallo único (fiabilidad)
- Arquitecturas SOA: romper en **servicios** independientes
- Cada servicio publica una **interfaz**
- Existen distintas aproximaciones y tecnologías

Orientación a servicios

SOC – Orientación a servicios - SOA



Orientación a servicios

SOC - Service Oriented Computing

- Plataforma de computación distribuida: conceptos, principios diseño, patrones, tecnologías, etc.
- Abarca todo lo que tiene que ver con servicios



Orientación a servicios

Orientación a servicios

- Nuevo paradigma de diseño de sistemas
- Sigue el principio “separación de asuntos”
- El sistema se rompe en fragmentos: servicios
- Cada servicio resuelve un aspecto concreto
- Cada servicio publica su funcionalidad y evoluciona de manera independiente



Orientación a servicios

SOA – Service Oriented Architecture

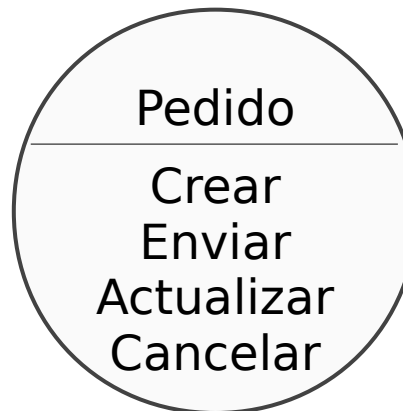
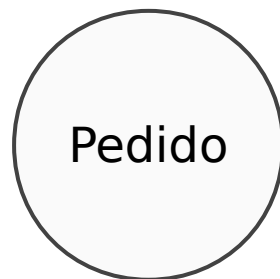
- Arquitectura software que sigue los principios de diseño de orientación a servicios
- Los servicios son piezas reusables que se combinan para obtener procesos de negocio ágiles y flexibles
- Se pueden utilizar diversas tecnologías: SOAP, procedimiento remoto, RESTful, etc.



Orientación a servicios

Servicios

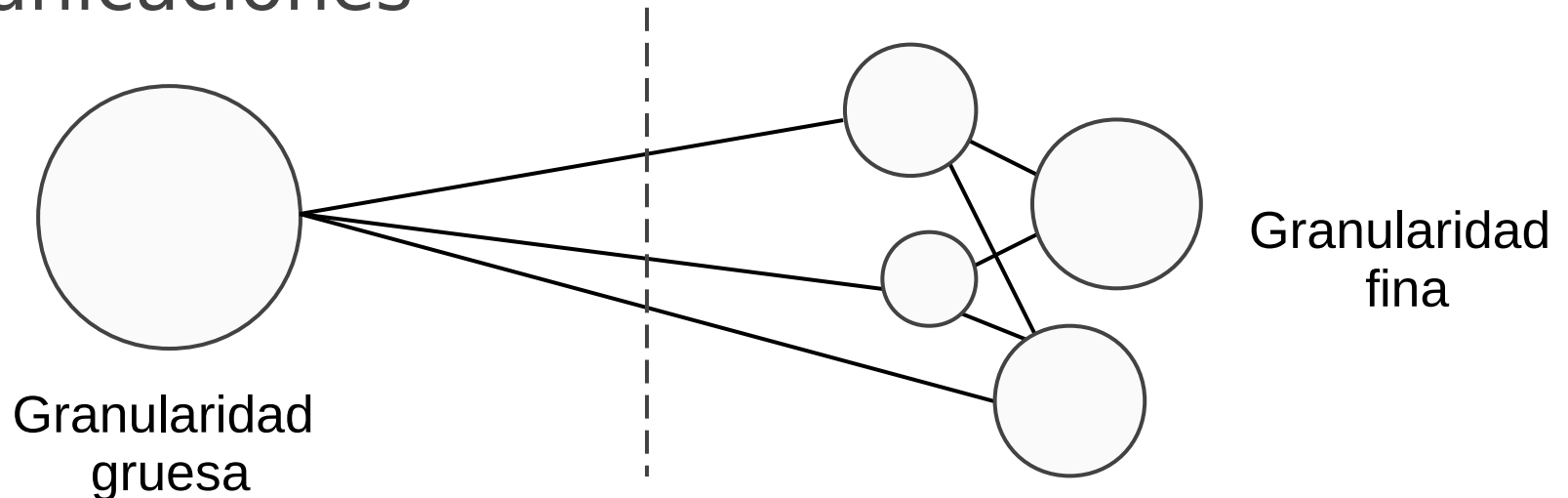
- Fragmento de software que ofrece una colección de **capacidades**, que se describen en un **contrato de servicio**
- Las capacidades están relacionadas por un contexto funcional común (cohesión)



Orientación a servicios

Servicios

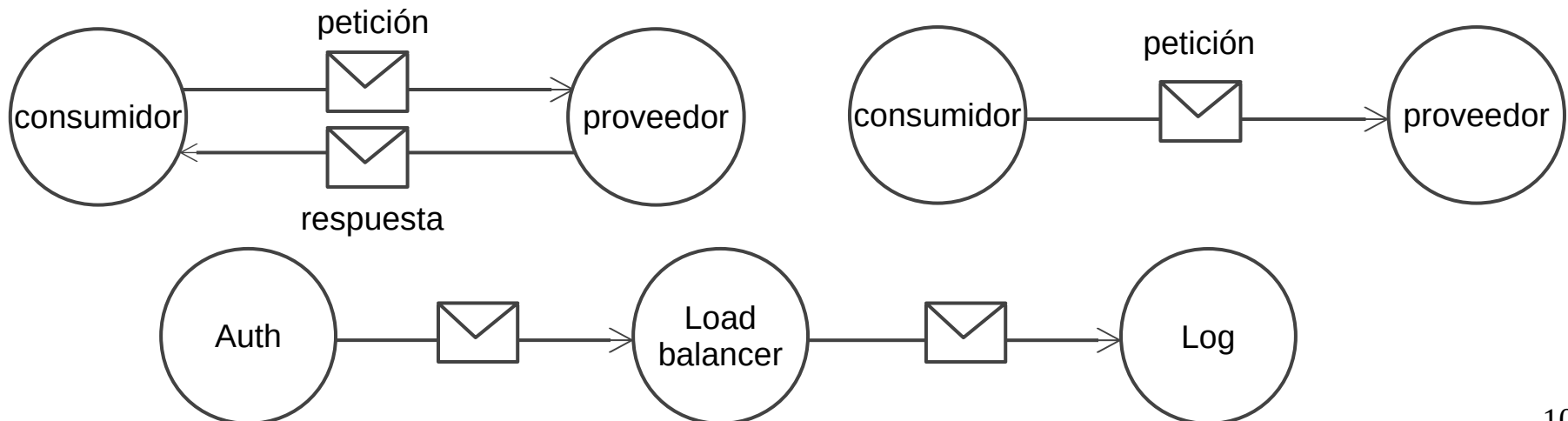
- **Granularidad** fina vs gruesa: funciones muy concretas y acotadas vs con gran alcance
- Se recomienda granularidad gruesa para simplificar la arquitectura y optimizar comunicaciones



Orientación a servicios

Servicios

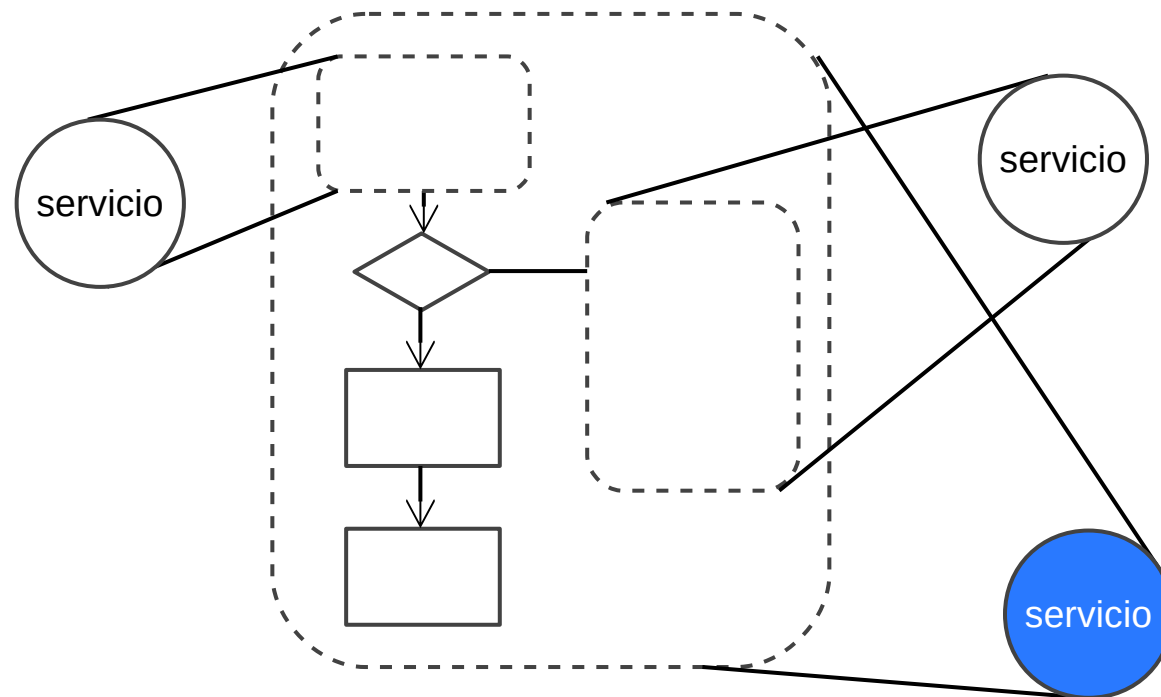
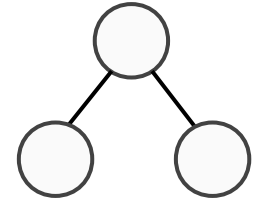
- Consumidor vs proveedor
- Comunicación (**síncrona** vs **asíncrona**) por paso de mensajes
- Comunicación directa o mediante intermediarios (**agentes**)



Orientación a servicios

Composición de servicios

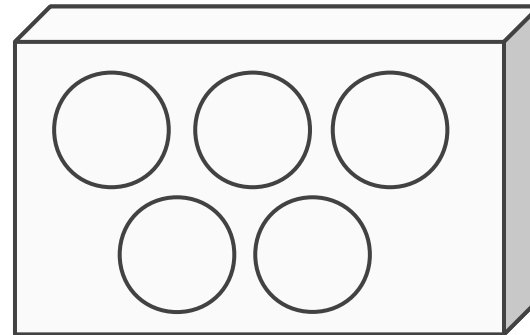
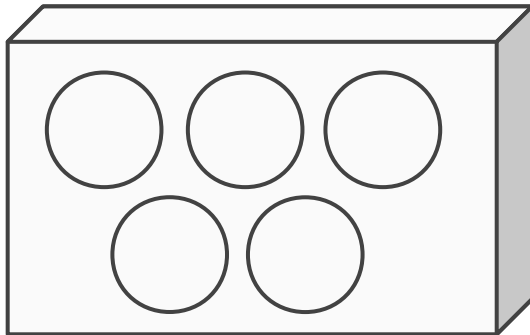
- Los servicios son reutilizables
- Se combinan en agregados para implementar una tarea común (LEGO)



Orientación a servicios

Inventario de servicios

- Grupos de servicios publicados por una misma organización



Orientación a servicios

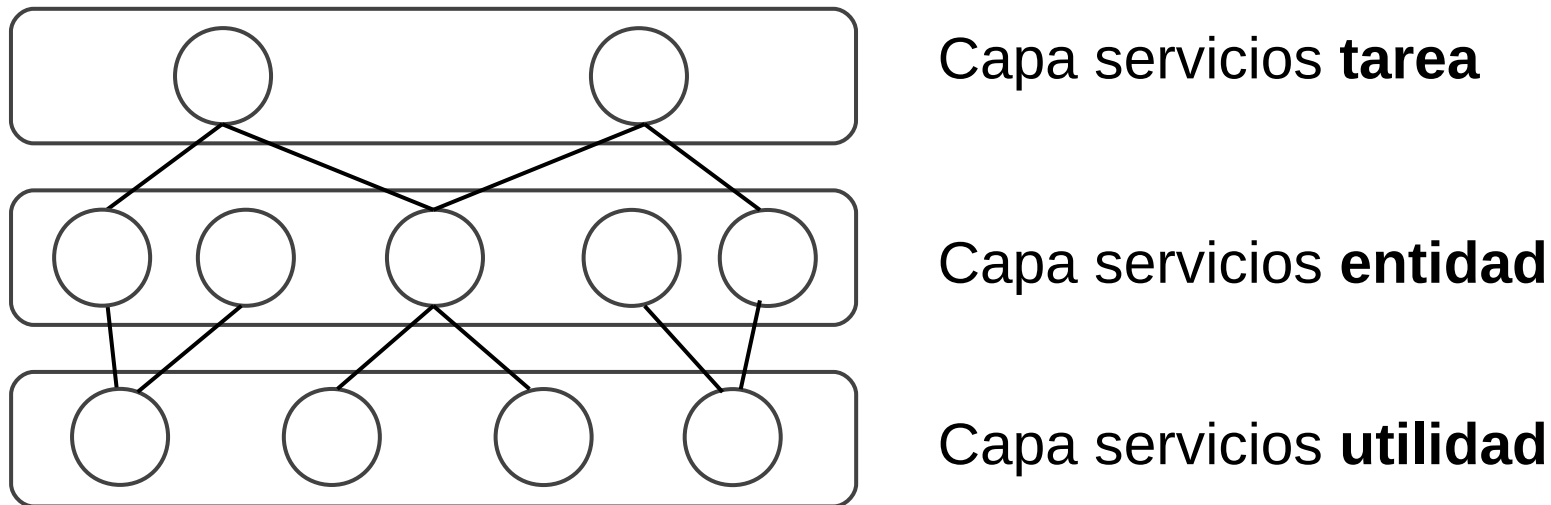
Modelos de servicio

- Tres modelos fundamentales
 - Servicios de **utilidad**: lógica multi-propósito, altamente reutilizables
 - Servicios de **entidad**: operaciones CRUD sobre entidad de negocio, altamente reutilizables
 - Servicios de **tarea** (**proceso**): implementan un proceso de negocio, composiciones de servicios, dinámicos, no reutilizables

Orientación a servicios

Modelos de servicio

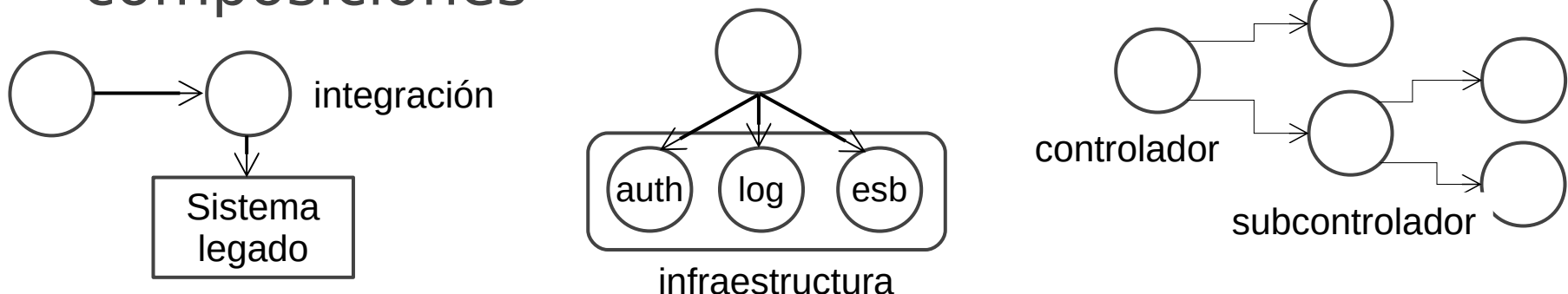
- Suelen agruparse en distintas **capas** de abstracción
- Servicios en capas superiores usan inferiores



Orientación a servicios

Modelos de servicio

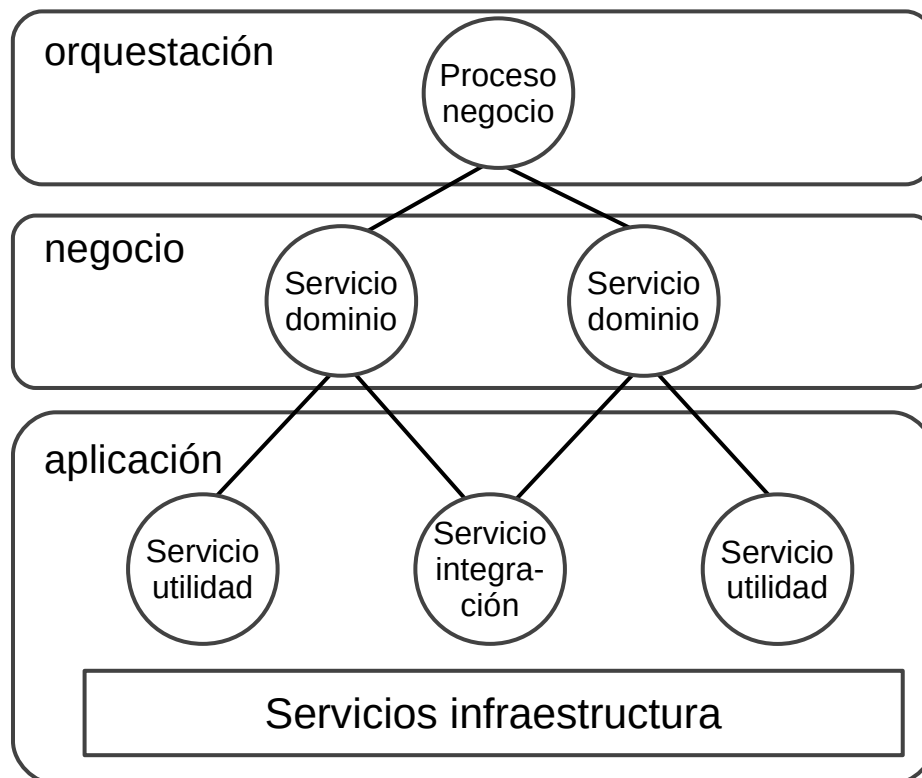
- Otros tipos de servicios muy comunes:
 - Servicios de **integración** (wrappers): envuelven sistemas legados
 - Servicios de **infraestructura**: servicios básicos (autenticación, log, etc.)
 - Servicios **controladores**: controlan a otros en composiciones



Orientación a servicios

Capas

- En una arquitectura SOA los servicios se organizan en capas



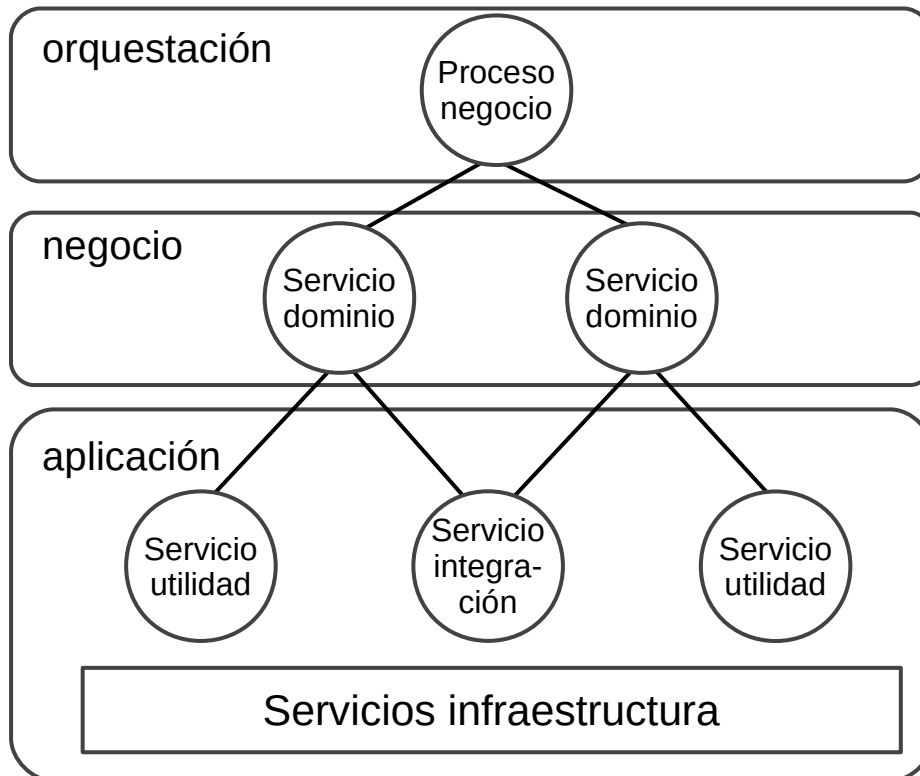
Servicios de orquestación

- Implementan los procesos de negocio
- Granularidad gruesa
- Cambian con frecuencia

Orientación a servicios

Capas

- En una arquitectura SOA los servicios se organizan en capas



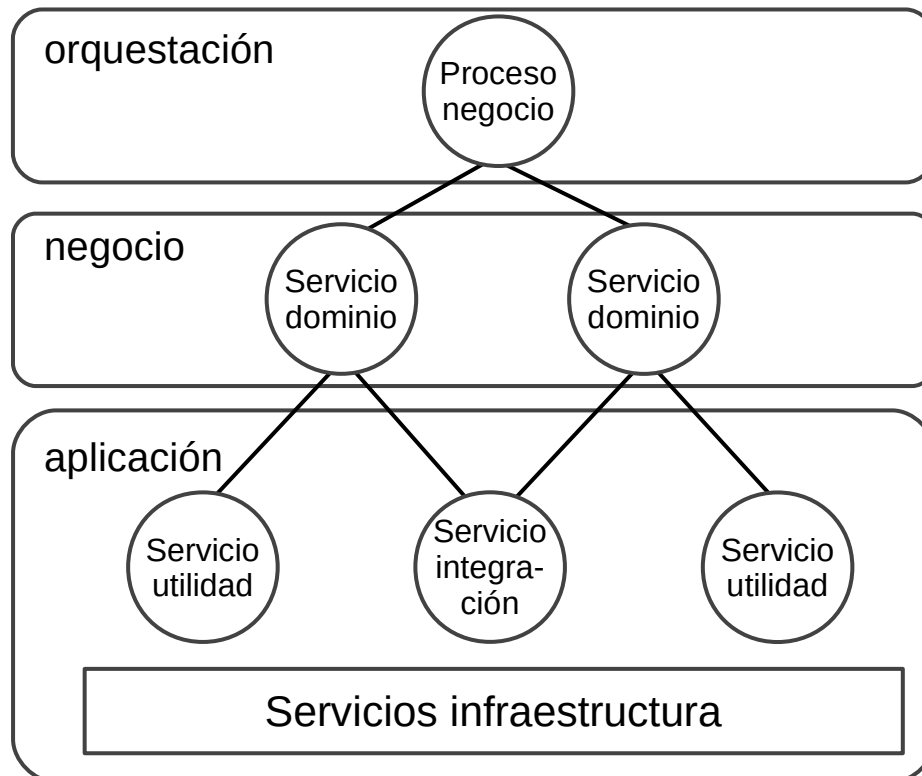
Servicios de negocio

- Implementan capacidades de dominios concretos
- Se organizan en subcapas: las superiores con servicios de tarea, las inferiores con servicios de entidad

Orientación a servicios

Capas

- En una arquitectura SOA los servicios se organizan en capas



Servicios de aplicación

- Implementan servicios de utilidad para el resto de capas
- Granularidad fina
- Servicios de integración, infraestructura y de propósito general

Diseño orientado a servicios

Hoja de ruta

- Principios de diseño
- Arquitectura SOA
- Proceso de desarrollo

Diseño orientado a servicios

Principios de diseño

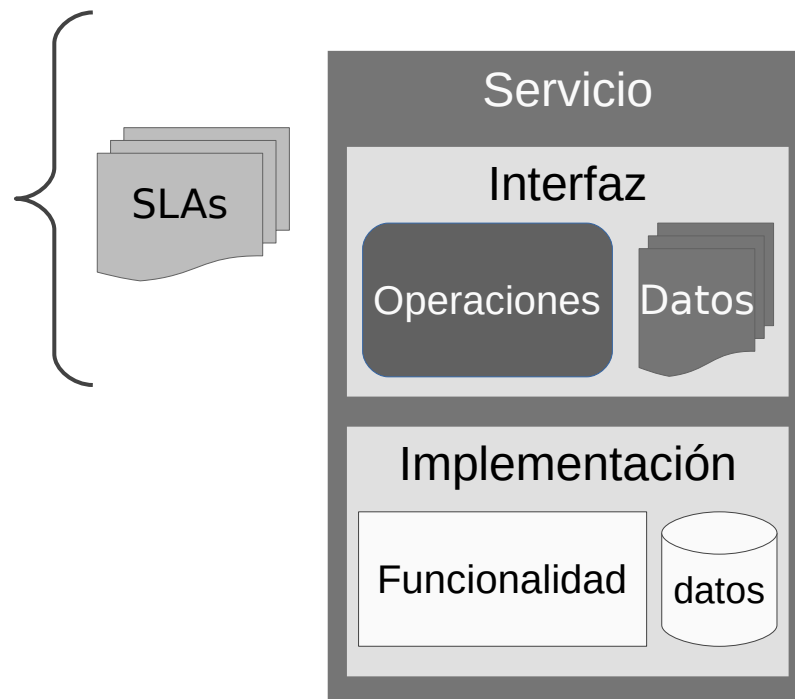
- Prácticas generalizadas y aceptadas para obtener arquitecturas SOA con características deseables
- Se identifican 8 principios interrelacionados
 1. Contrato de servicio estándar
 2. Bajo acoplamiento
 3. Abstracción
 4. Reusabilidad
 5. Autonomía
 6. Sin estado
 7. Descubrimiento
 8. Composición

Diseño orientado a servicios

Principios de diseño (8)

1. Contrato de servicio estándar

- Debe describir las capacidades, limitaciones, interfaz (API), garantías de servicio (QoS), ...



Diseño orientado a servicios

Principios de diseño (8)

2. Bajo acoplamiento

- El contrato desacopla interfaz de implementación
- Además, no impone dependencias sobre el consumidor, ni sobre la implementación
- El servicio evoluciona de manera independiente

3. Abstracción

- El contrato no contiene detalles de implementación
- Se favorece la reutilización y evolución independiente de la implementación

Diseño orientado a servicios

Principios de diseño (8)

4. Reusabilidad

- Las capacidades expuestas son genéricas y pueden reutilizarse en distintos procesos/tecnologías
- Servicios de entidad/utilidad vs de tarea

5. Autonomía

- Auto-gobierno, control sobre el entorno, recursos, sin dependencias externas
- El servicio es más predecible y fiable, evoluciona de manera independiente

Diseño orientado a servicios

Principios de diseño (8)

6. Sin estado

- Delegar almacenamiento a entidad externa (aumenta acoplamiento y reduce autonomía)
- Minimiza el consumo de recursos y favorece escalabilidad

7. Descubrimiento

- Contratos con metadatos para habilitar descubrimiento en repositorios
- Tiempo diseño vs ejecución

Diseño orientado a servicios

Principios de diseño (8)

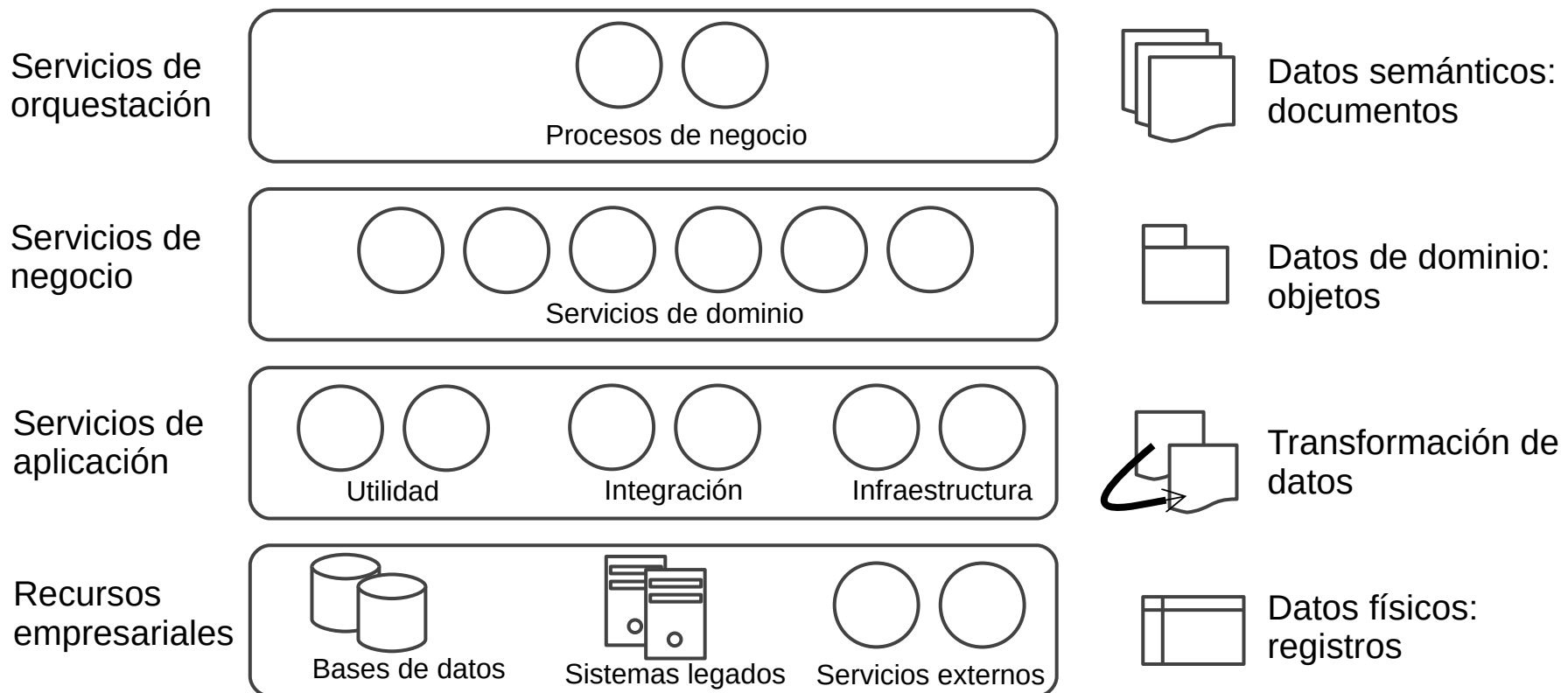
8. Composición

- Fácilmente agregables en servicios compuestos
- Depende en gran medida del resto de principios

Diseño orientado a servicios

Arquitectura SOA

- Arquitectura por capas



Diseño orientado a servicios

Arquitectura SOA

- Cada capa tiene un propósito
 - Servicios de orquestación: procesos de negocio (e.g. crear nuevo empleado, ordenar pedido, ...)
 - Servicios de negocio: funcionalidades de negocio de alto nivel (e.g. gestionar clientes, ...)
 - Servicios de aplicación: utilidades, integración, infraestructura
 - Recursos empresariales: bases de datos, sistemas legados, etc.

Diseño orientado a servicios

Arquitectura SOA

- Cada capa trabaja con un tipo de datos
 - Datos semánticos: información de alto nivel que utilizan los procesos de negocio, documentos
 - Datos de dominio: datos con los que trabaja internamente cada servicio de dominio, requiere traducción previa
 - Datos físicos: datos que se almacenan en disco

Diseño orientado a servicios

Proceso de desarrollo

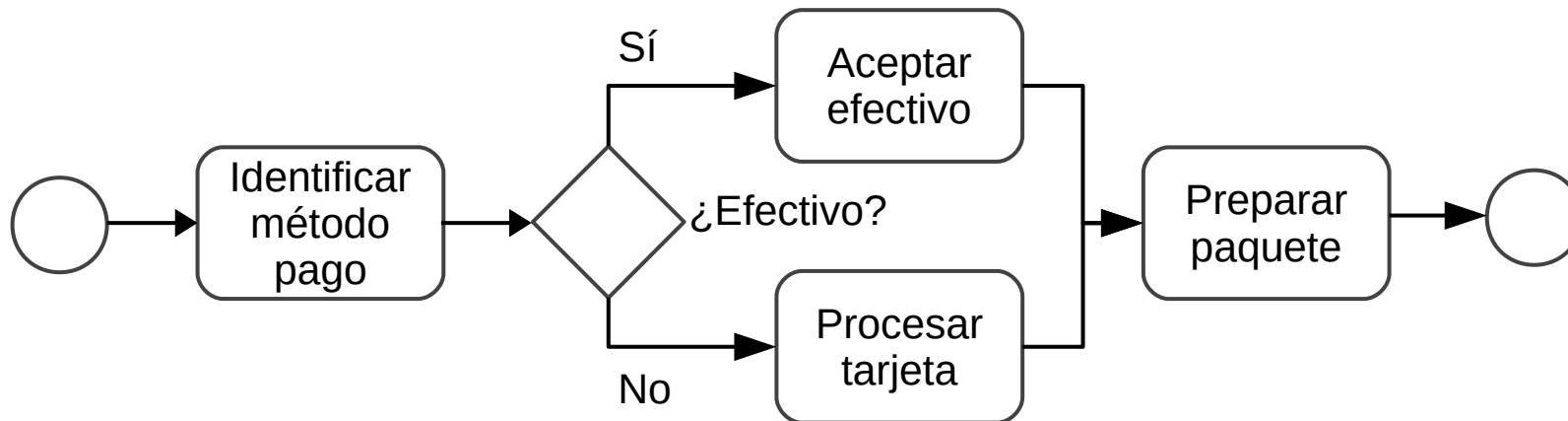
- Para obtener una arquitectura SOA solvente es necesario seguir un proceso bien definido
- Fases globales:
 1. Definición del modelo de negocio
 2. Definición de los datos semánticos
 3. Identificación de servicios
 4. Especificación de servicios
 5. Implementación de servicios

Diseño orientado a servicios

Proceso de desarrollo

1. Definición del modelo de negocio

- Se definen los recursos de la empresa
- Se definen los procesos de negocio (BPMN)

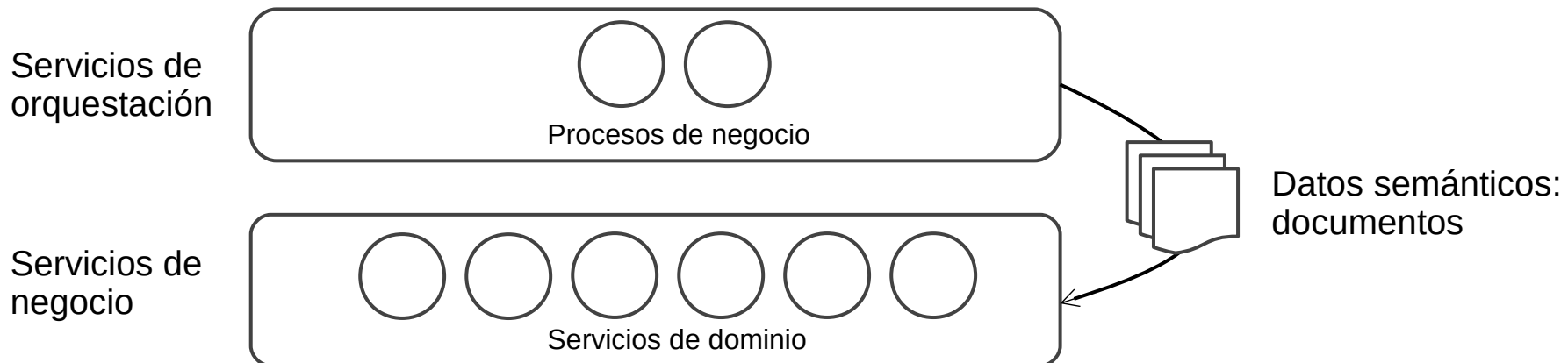


Diseño orientado a servicios

Proceso de desarrollo

2. Definición de los datos semánticos

- Se identifica la información de alto nivel, que necesitan todos los procesos de negocio (e.g. usuario, pedido, ...)
- Esta información se utiliza de entrada/salida para los servicios de dominio (en la capa de negocio)

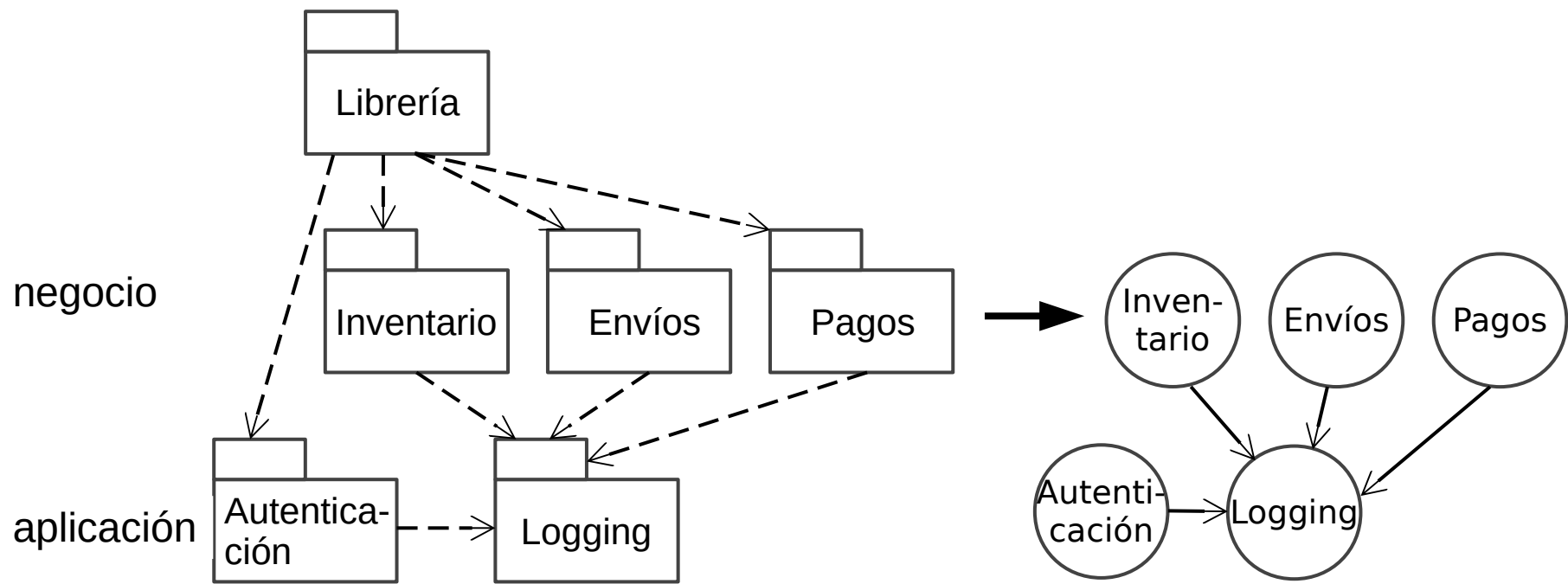


Diseño orientado a servicios

Proceso de desarrollo

3. Identificación de servicios

- Identificación de dominios y grafo de dependencias

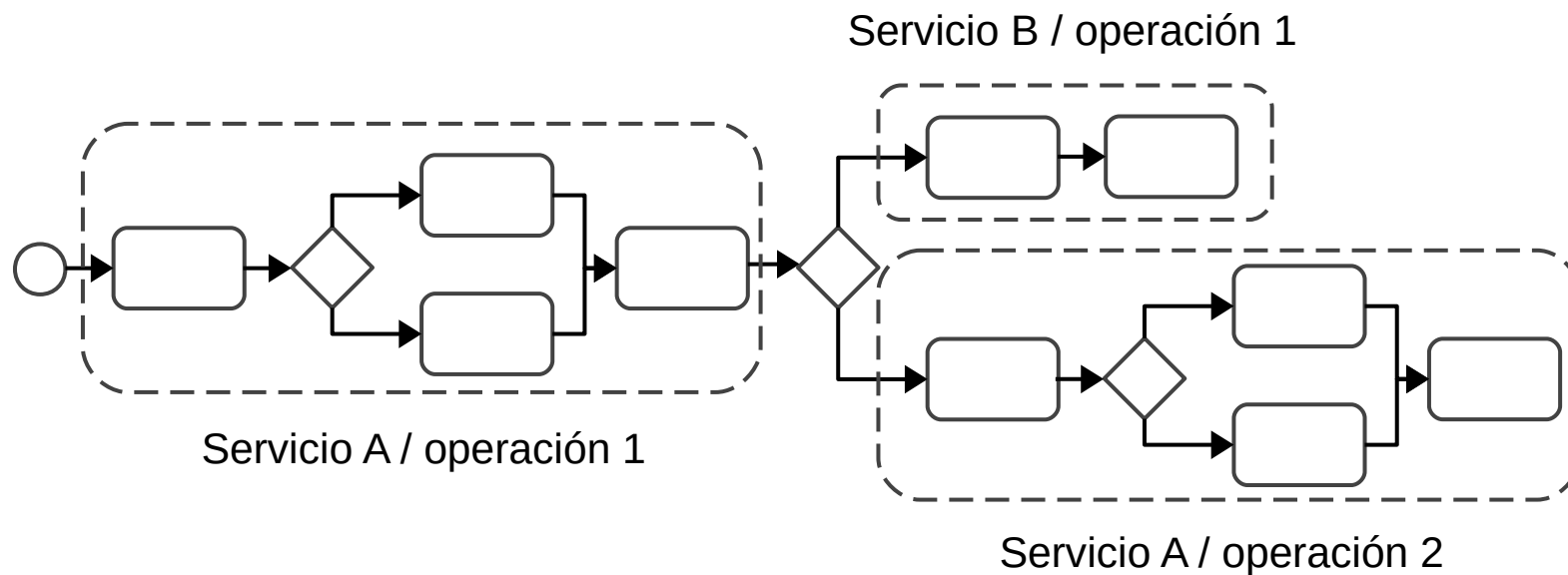


Diseño orientado a servicios

Proceso de desarrollo

3. Identificación de servicios

- Descomposición jerárquica de los procesos en operaciones de servicios

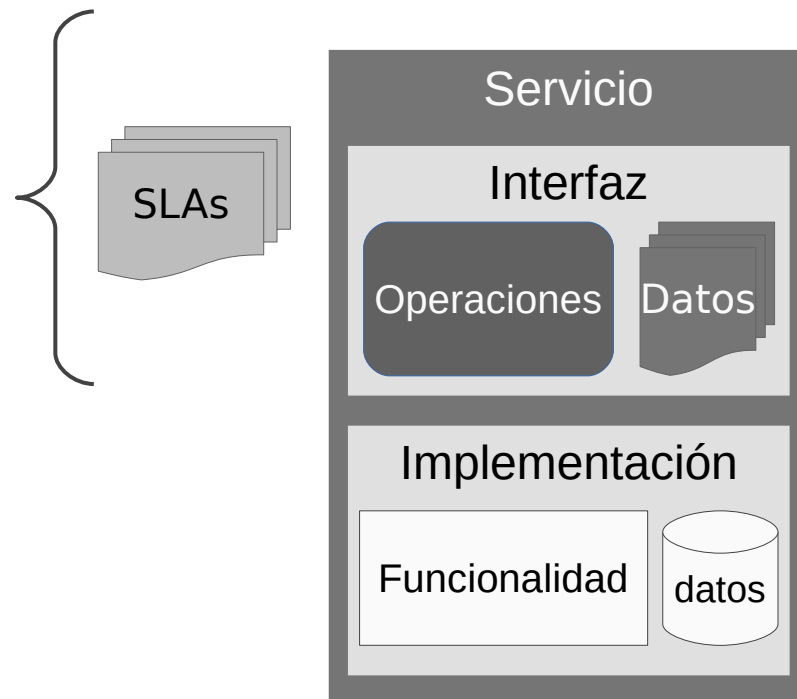


Diseño orientado a servicios

Proceso de desarrollo

4. Especificación de servicios

- Se definen los contratos de servicio, incluyendo su interfaz, limitaciones, calidad de servicio, etc.

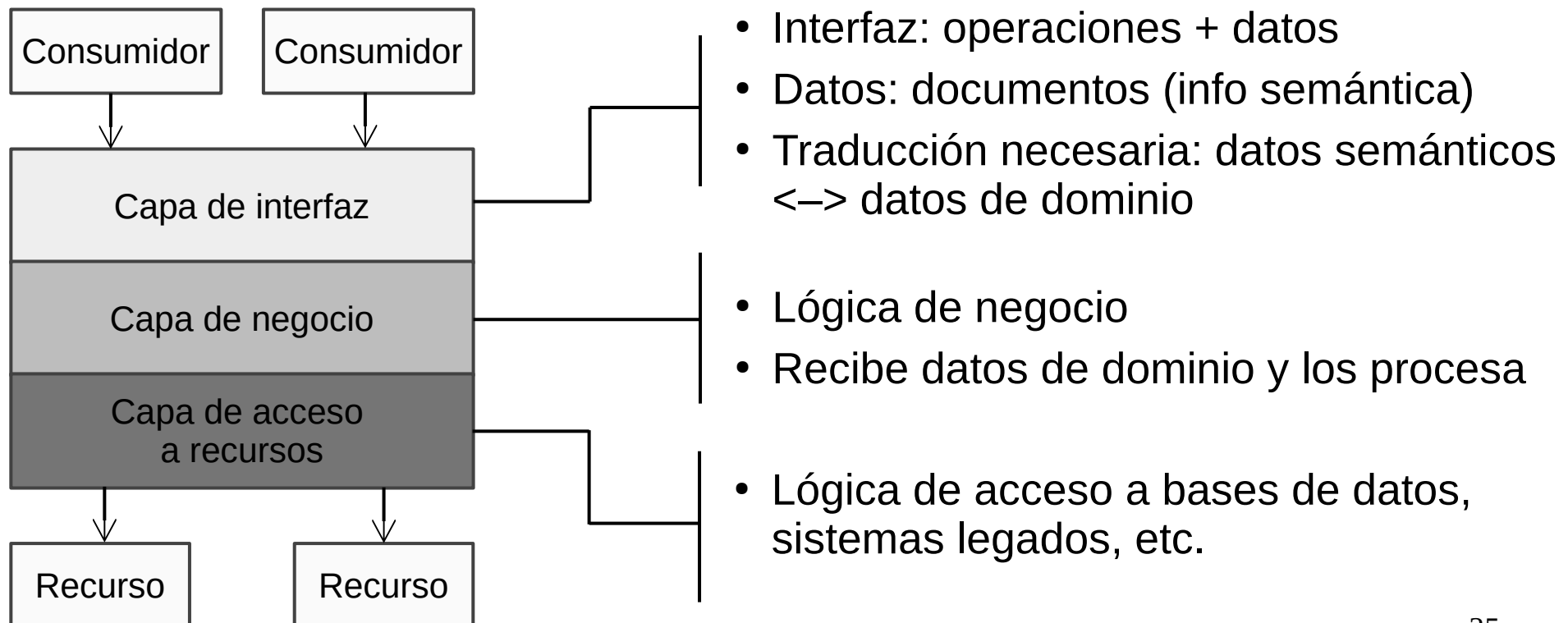


Diseño orientado a servicios

Proceso de desarrollo

5. Implementación de servicios

- Cada servicio es implementado independientemente



Diseño orientado a servicios

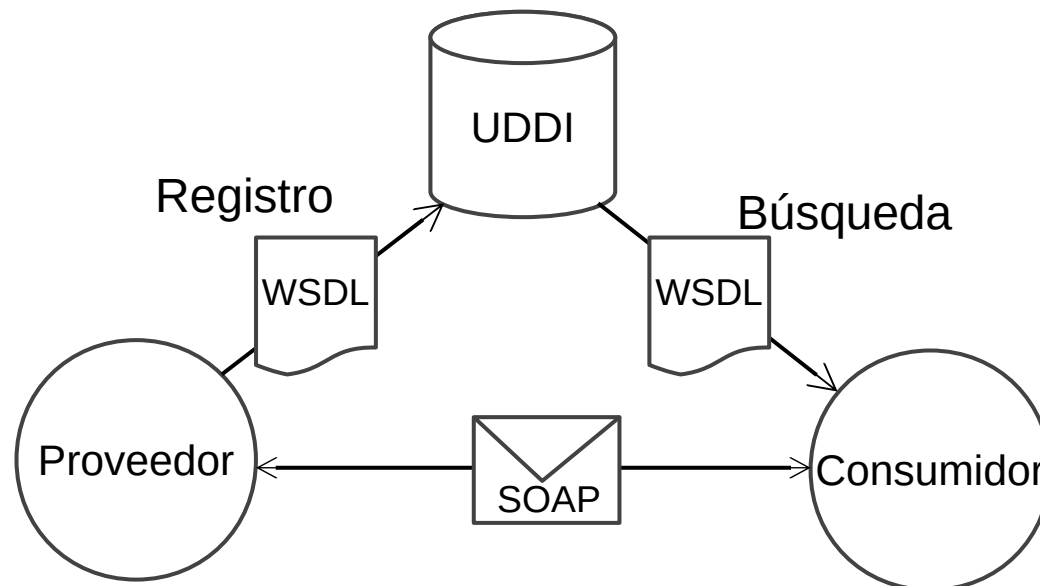
Aproximaciones diseñar arquitectura SOA

- Top-down: se sigue el proceso presentado
 - El resultado es un inventario de alta calidad
 - Requiere mucho esfuerzo
- Bottom-up: integrar sistemas existentes
 - Se obtienen soluciones inmediatas
 - No es una solución SOA, es una adaptación
- Middle-out: aproximación intermedia
 - Proceso complejo: análisis + implementación ágil

Servicios web (SOAP)

Introducción

- Tecnología basada en estándares abiertos
- Contrato de servicio con WSDL, descubrimiento con UDDI, intercambio de mensajes con SOAP



Servicios web (SOAP)

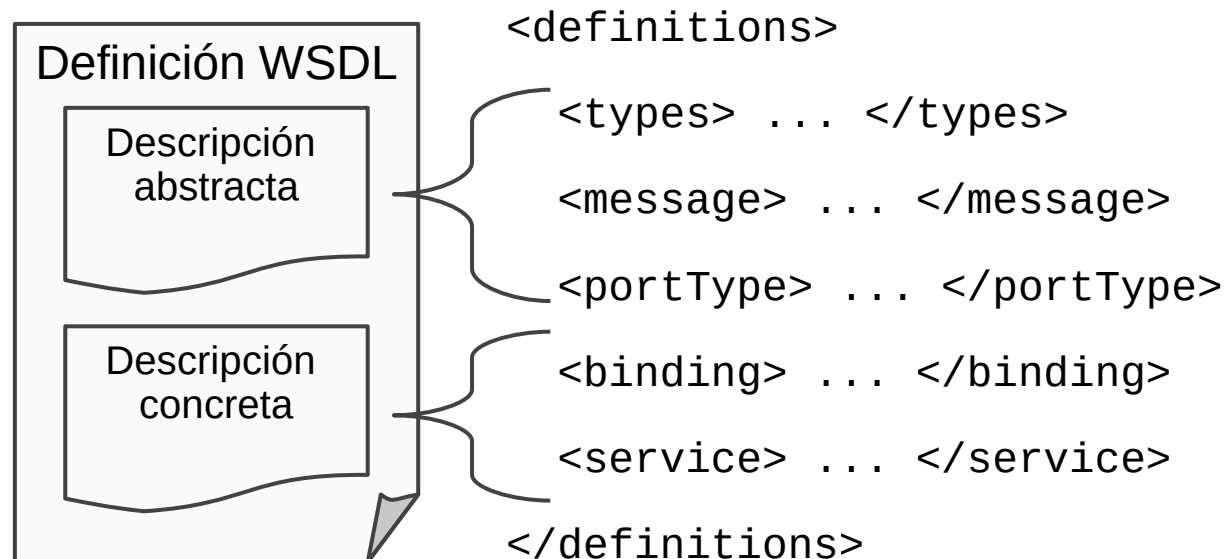
Hoja de ruta

- El contrato de servicio: WSDL
- Implementación con Node.js
- Descubrimiento de servicios
- Consumo de servicios
- Extensiones WS-*

Servicios web (SOAP)

El contrato de servicio: WSDL

- Descripción abstracta (lógica): interfaz (operaciones + datos)
- Descripción concreta (física): tecnologías de comunicación + dirección



Servicios web (SOAP)

El contrato de servicio: WSDL

<types>

- Contiene todos los tipos de datos usados, utilizando [XML Schema](#)

```
<definitions>
  <types>
    <xsd:element name="User">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="name" type="xsd:string"/>
          <xsd:element name="surname" type="xsd:string"/>
          <xsd:element name="address" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </types>
  ...
</definitions>
```


Servicios web (SOAP)

El contrato de servicio: WSDL

`<message>`

- Define los mensajes intercambiados entre el consumidor y el proveedor
- El contenido de los mensajes son datos definidos en `<types>`

`<types>`

`<definitions>`

```
  <types> ... </types>
```

```
  <message name="getBMIRequest">
```

```
    <part name="parameters" element="tns:bmi"/>
```

```
  </message>
```

```
  <message name="getBMIResponse">
```

```
    <part name="parameters" element="tns:bmiResponse"/>
```

```
  </message>
```

```
  ...
```

```
</definitions>
```

Servicios web (SOAP)

El contrato de servicio: WSDL

<portType>

- Define la interfaz del servicio: incluye las operaciones
- Cada operación determina los mensajes intercambiados

```
<definitions>
  <types>...</types>
  <message ...name="echoRequest">...</message>
  <message name="echoResponse">...</message>
  <portType name="EchoPortType">
    <operation name="echo">
      <input message="tns:echoRequest"/>
      <output message="tns:echoResponse"/>
      <fault message="tns:echoFault"/>
    </operation>
  </portType>
  ...
</definitions>
```

Servicios web (SOAP)

El contrato de servicio: WSDL

<binding>

- Determina el protocolo que se usa para codificar y transferir los mensajes (SOAP + HTTP)

```
<definitions>
  ...
  <binding name="EchoPort" type="tns:EchoPortType">
    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="echo">
      <soap:operation soapAction="http://www.example.org/test/echo"/>
      <input><soap:body use="literal"/></input>
      <output><soap:body use="literal"/></output>
    </operation>
  </binding>
  ...
</definitions>
```

Servicios web (SOAP)

El contrato de servicio: WSDL

`<service>`

- Publica el servicio bajo uno o más endpoints (`<port>`)

```
<definitions>
```

```
...
```

```
<binding name="EchoPort" type="tns:EchoPortType">...</binding>
```

```
<service name="EchoService">
```

```
  <port binding="tns:EchoPort" name="EchoPort">
```

```
    <soap:address location="http://localhost:8000/echo/">
```

```
  </port>
```

```
</service>
```

```
...
```

```
</definitions>
```

Servicios web (SOAP)

Implementación con Node.js

- Utilizaremos el paquete [soap](#)
> `npm install soap`
- Se siguen los siguientes pasos:
 1. Implementación del contrato
 2. Creación de un servidor HTTP
 3. Redirección de peticiones HTTP al servicio

Servicios web (SOAP)

Implementación con Node.js

1. Implementación del contrato

- Se implementan las operaciones definidas en el .wsdl, siguiendo las **convenciones**, y **tipos de datos**

```
<definitions>
  <portType name="EchoPortType">
    <operation name="echo">
      <input message="echoRequest" />
      <output message="echoResponse" />
    </operation>
  </portType>

  <service name="EchoService">
    <port name="EchoPort">
      <soap:address location="http://localhost:8000/echo/" />
    </port>
  </service>
</definition>
```

```
var myService = {
  EchoService: {
    EchoPort: {
      echo: function(args) {
        return {out: args.in};
      }
    }
  }
}
```

Servicios web (SOAP)

Implementación con Node.js

2. Creación de un servidor HTTP

- Se usa el paquete core [http](#), el servidor escucha por el puerto adecuado

```
var http = require('http');  
// create http server  
var server = http.createServer(function(request, response) {  
    response.end('404: Not Found: ' + request.url);  
});  
server.listen(8000);
```

Servicios web (SOAP)

Implementación con Node.js

3. Redirección de peticiones HTTP al servicio

- Con [soap.listen\(server,path,service,wsdl,cb\)](#)
- Redirigir peticiones encaminadas a la URL adecuada

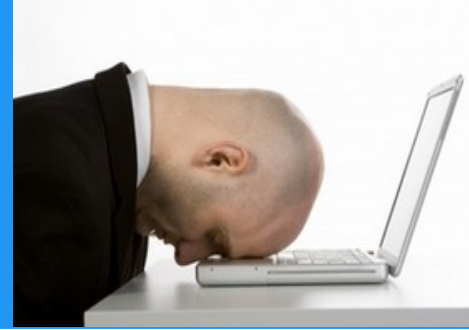
```
var myService = {...}
var server = ...

var soap = require('soap');
var fs = require('fs');

var wsdl= fs.readFileSync('echo.wsdl', 'utf8');

soap.listen(server, '/echo', myService, wsdl, function(){
  console.log('server initialized');
});
```

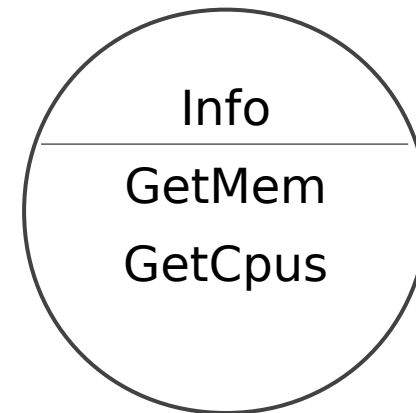

Servicios web (SOAP)



Ejercicio 1

Implementar un servicio web para recuperar info de la máquina/sistema operativo

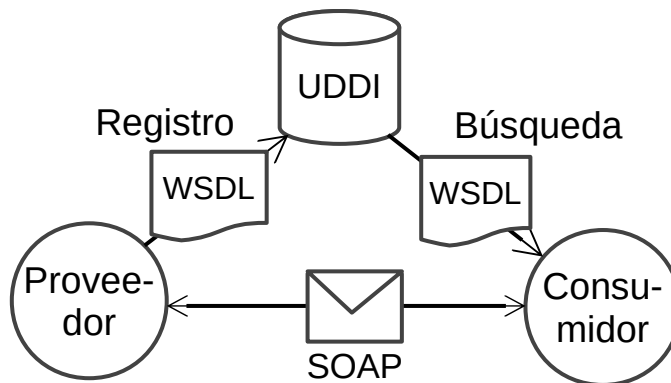
- Definir WSDL
- Implementar proveedor



Servicios web (SOAP)

Descubrimiento de servicios

- Distintas alternativas para obtener el contrato:
 - Ubicación fija
 - Transferencia al consumidor
 - Repositorio de contratos público
 - Permite el descubrimiento dinámico
 - UDDI nació con este fin



Servicios web (SOAP)

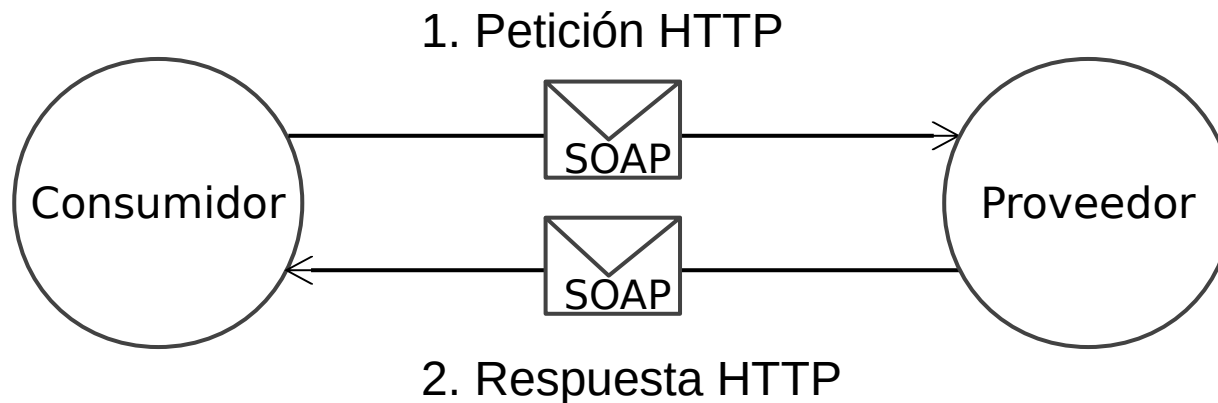
Descubrimiento de servicios

- UDDI
 - Inicialmente registros públicos impulsados por grandes compañías (e.g. Microsoft, IBM, ...)
 - El protocolo era complejo, el sistema de catalogación rígido y dependía exclusivamente de WSDL/SOAP
 - En la actualidad sólo se usa internamente
- ¿Alternativas? Google, soluciones propietarias

Servicios web (SOAP)

Consumo de servicios

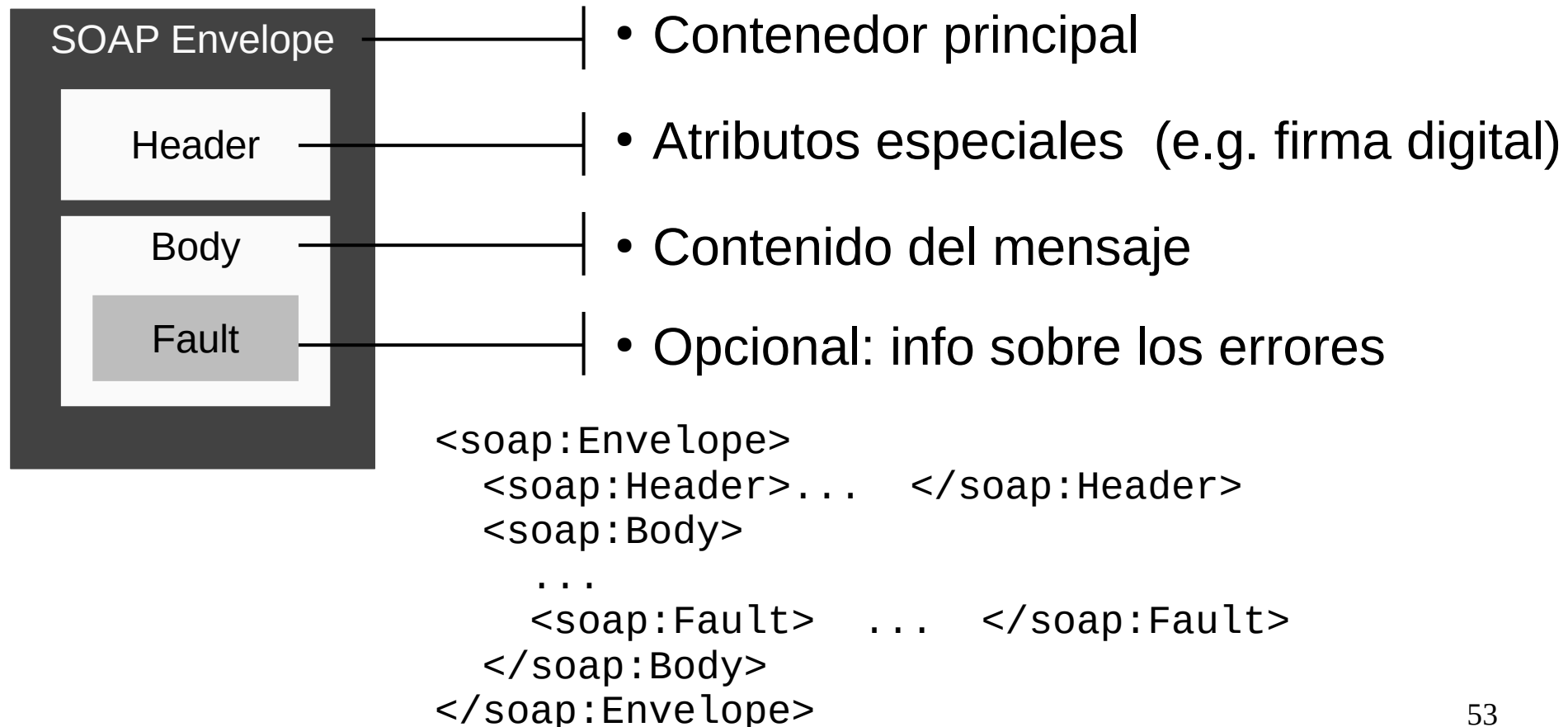
- Comunicación por intercambio de mensajes
- Codificación con SOAP y transferencia con HTTP



Servicios web (SOAP)

Consumo de servicios > SOAP

- Estructura de un mensaje SOAP



Servicios web (SOAP)

Consumo de servicios > SOAP

- Petición SOAP

```
<soap:Envelope>  
  <soap:Body>  
    <m:echo>  
      <m:in>Hola</m:in>  
    </m:echo>  
  </soap:Body>  
</soap:Envelope>
```

- Respuesta SOAP

```
<soap:Envelope>  
  <soap:Body>  
    <m:echoResponse>  
      <m:out>Hola</m:out>  
    </m:echoResponse>  
  </soap:Body>  
</soap:Envelope>
```

Servicios web (SOAP)

Consumo de servicios > Node.js

- Usamos el paquete [soap](#)
> `npm install soap`
- Se siguen los siguientes pasos:
 1. Recuperar el contrato .wsdl
 2. Creación de un proxy
 3. Invocar las operaciones del servicio

Servicios web (SOAP)

Consumo de servicios > Node.js

1. Recuperar el contrato .wsdl

- Es necesario facilitar una URL (localo o remota)

```
var url = 'http://localhost:8000/echo?wsdl';
```

2. Creación de un proxy

- Con [soap.createClient\(url, opts, cb\)](#)

```
var soap = require('soap');  
soap.createClient(url, function (err, client) {  
  // show info about service  
  console.log(JSON.stringify(client.describe()));  
});
```


Servicios web (SOAP)

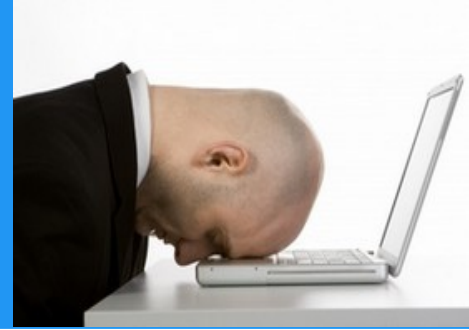
Consumo de servicios > Node.js

3. Invocar las operaciones del servicio

- A través del path totalmente cualificado o abreviado

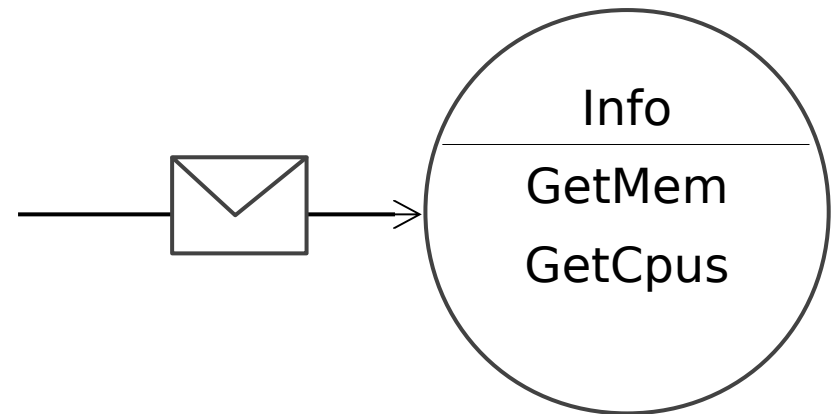
```
soap.createClient(url, function (err, client) {  
  // invoke operation using full path  
  client.EchoService.EchoPort.echo({in:'hello'}, function(err, res){  
    console.log(JSON.stringify(res));  
  });  
  
  // invoke operation using short path  
  client.echo({ in: 'hello' }, function (err, res) {  
    console.log(JSON.stringify(res));  
  });  
});
```

Servicios web (SOAP)



Ejercicio 2

Crear un CLI que consuma el servicio Info del ejercicio 1



Servicios web (SOAP)

Extensiones WS-*

- Nuevas especificaciones que extienden la primera generación de servicios web WSDL-SOAP-UDDI
 - WS-Coordination, WS-BPEL, WS-CDL
 - WS-AtomicTransaction
 - WS-Addressing
 - WS-Notification, WS-Eventing, WS-ReliableMessaging
 - WS-Security
 - WS-Policy
 - WS-I, ...

Servicios RESTful

Introducción

- Los servicios web son muy potentes, pero muy complejos (WS-* difíciles de manejar)
- SOAP no es eficiente (XML ??)
- Otra alternativa es posible:
 - Gestionar colección recursos remotos
 - Simple, eficiente y con tecnologías estándar
 - Son los objetivos de las arquitecturas REST

Servicios RESTful

Hoja de ruta

- Principios de arquitectura
- El protocolo HTTP
- El contrato de servicio
- Implementación con Node.js
- Consumo de servicios

Servicios RESTful

Principios de arquitectura

- REST: Representational State Transfer
- Aparece por primera vez en la [tesis doctoral](#) de Roy Fielding
- Originalmente se refería a un conjunto de [principios de arquitectura](#), aprendidos de WWW
- Actualmente se utiliza para describir cualquier interfaz remota que use HTTP como protocolo de comunicación

Servicios RESTful

Principios de arquitectura

- Cliente-servidor
- Sin estado
- Cache
- Interfaz uniforme
- Sistema a capas
- Código bajo demanda

Servicios RESTful

Principios de arquitectura

- Un servicio RESTful verifica los principio REST, y sigue las siguientes reglas:
 - Envuelve una colección de recursos. Cada recurso posee un identificador-URI único
 - Recursos se manipulan a través de representaciones: los datos son documentos
 - Múltiples representaciones por recurso (XML,JSON,...)
 - Recursos accedidos por operaciones CRUD
 - Proveedor sin estado

Servicios RESTful

Principios de arquitectura

- Propiedades de las arquitecturas REST:
 - Rendimiento
 - Escalabilidad
 - Simplicidad
 - Modificabilidad
 - Visibilidad
 - Portabilidad
 - Fiabilidad

Servicios RESTful

HTTP

- Protocolo de transporte habitual en RESTful
- Protocolo de comunicación sin estado que permite la transferencia/manipulación de recursos en Internet
- Los recursos se representan por medio de URLs (Uniform Resource Locator)

esquema://máquina:puerto/path?query#fragmento

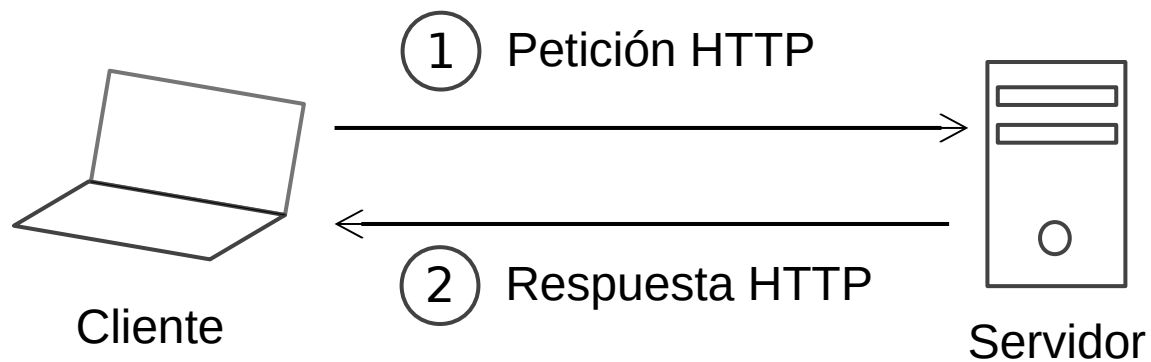
<http://ejemplo.com/data/api/user/username?kind=alumno>

- Soporta distintos tipos de operaciones sobre recursos: GET, POST, PUT, DELETE, etc.

Servicios RESTful

HTTP

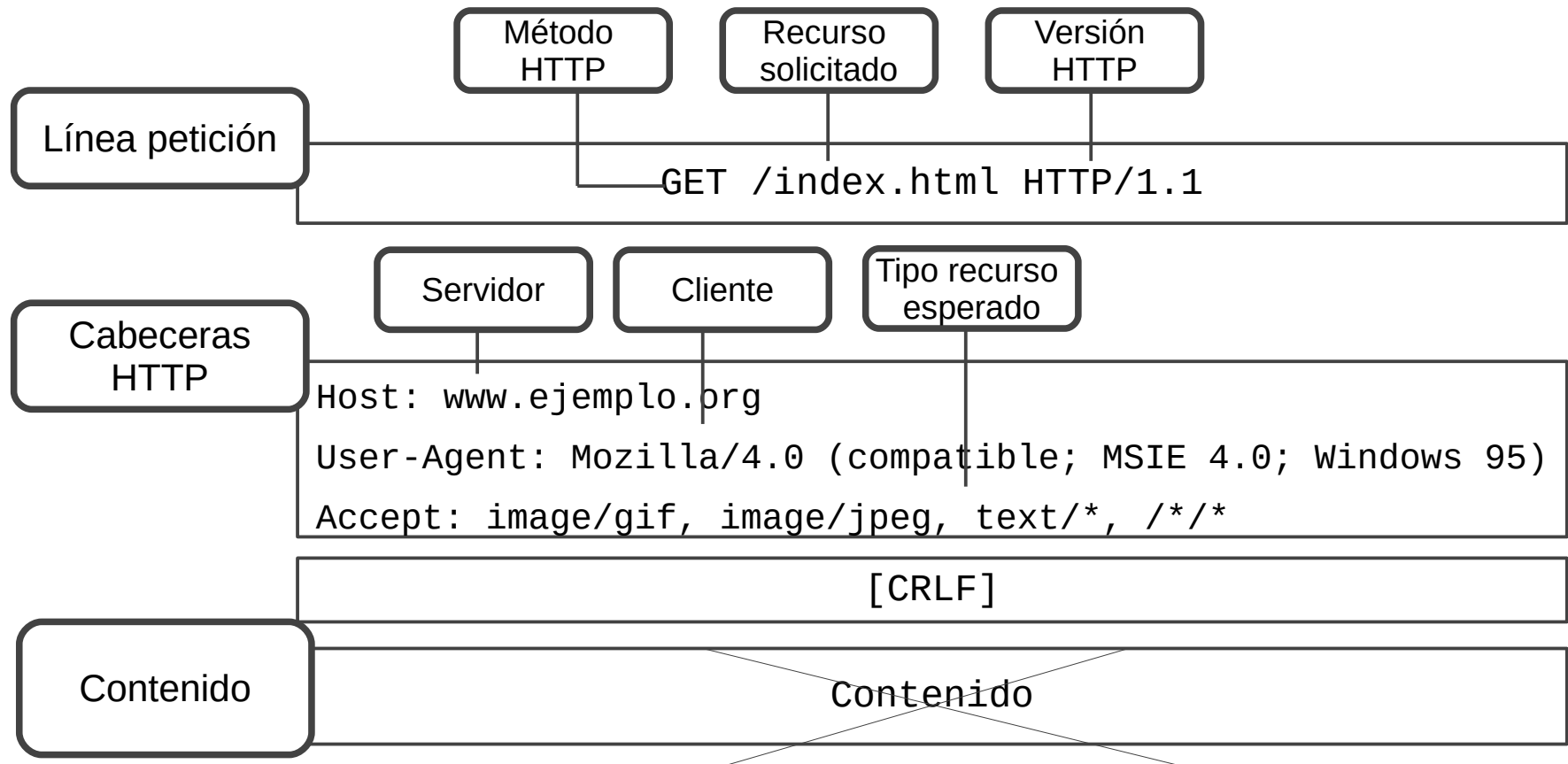
- Petición/respuesta HTTP



Servicios RESTful

HTTP

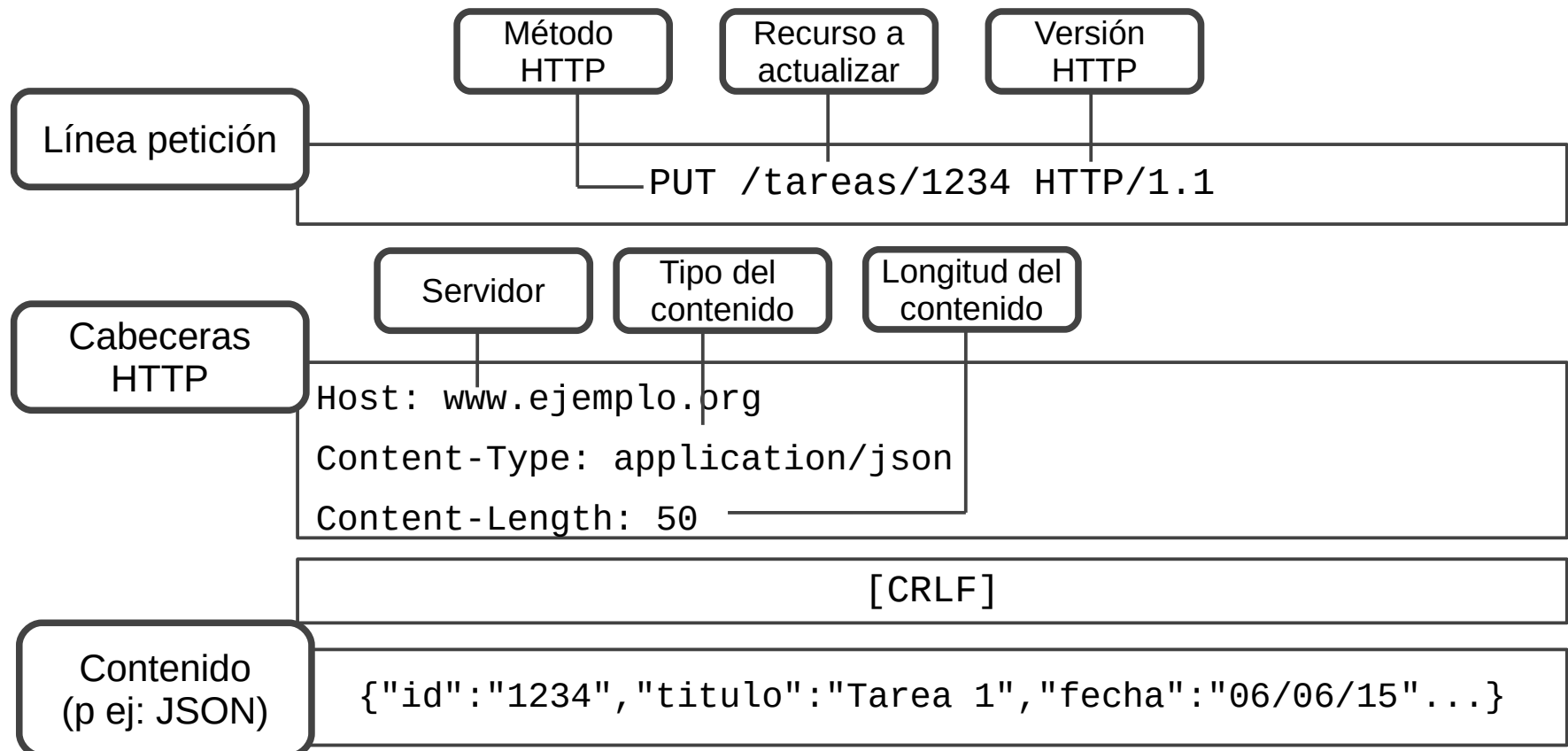
- Petición HTTP de recuperación



Servicios RESTful

HTTP

- Petición HTTP de actualización



Servicios RESTful

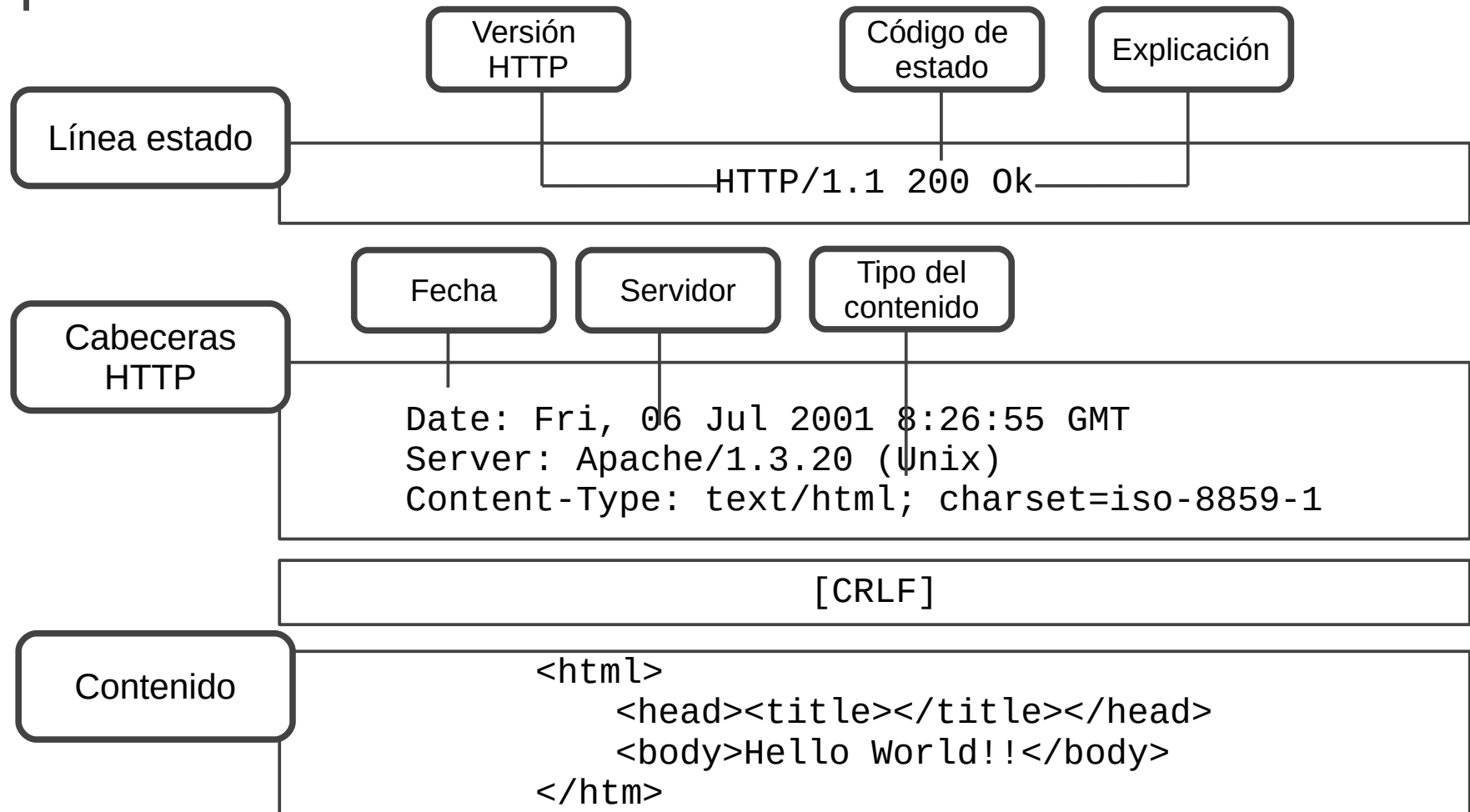
HTTP

- Cabeceras soportadas en una petición HTTP
 - Accept, Accept-Charset, Accept-Encoding, Accept-Language, Accept-Datetime, Authorization, Connection, Cookie, Content-Length, Content-MD5, Content-Type, Date, Expect, From, Host, If-Match, If-Modified-Since, If-None-Match, If-Range, If-Unmodified-Since, Max-Forwards, Origin, Proxy-Authorization, User-Agent, ...

Servicios RESTful


HTTP

- Respuesta HTTP



Servicios RESTful

HTTP

- Códigos de error HTTP:
 - 1xx (información)
 - 2xx (éxito)
 - 3xx (redirección)
 - 4xx (error causado por cliente)
 - 5xx (error causado por servidor)

Servicios RESTful

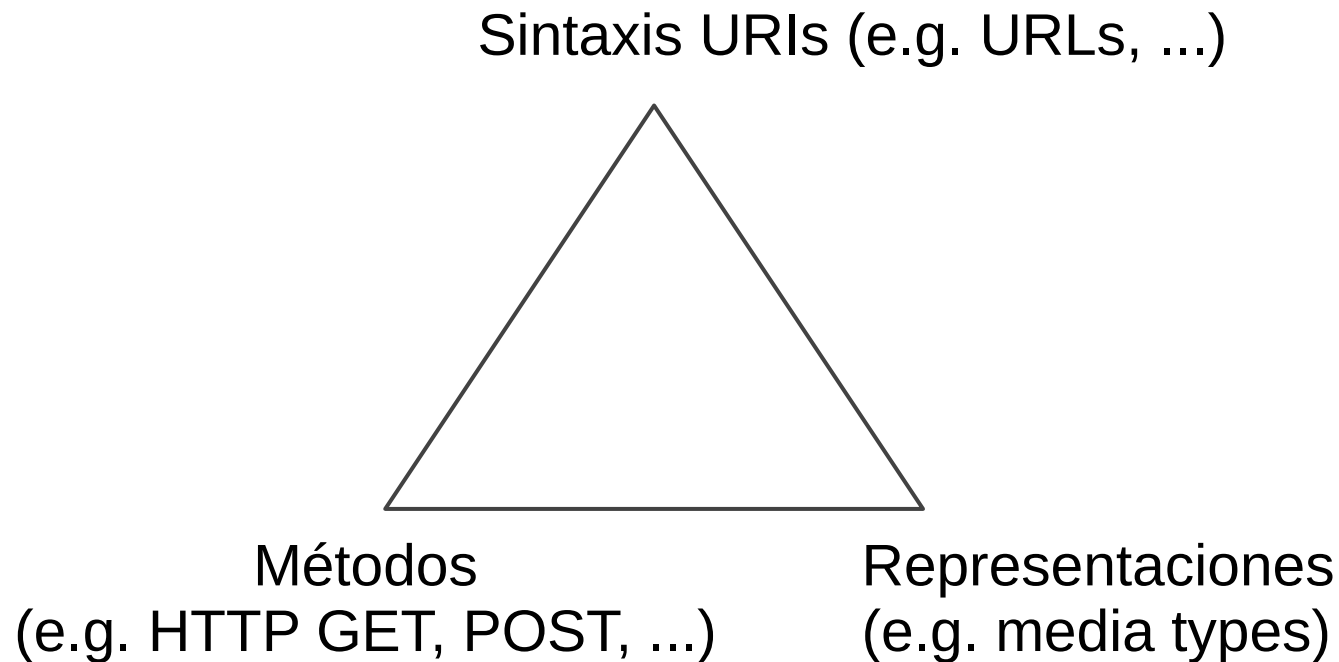
HTTP

- Cabeceras soportadas en una respuesta HTTP
 - Access-Control-Allow-Origin, Accept-Patch, Accept-Ranges, Age, Allow, **Cache-Control**, Connection, Content-Disposition, Content-Encoding, Content-Language, **Content-Length**, Content-Location, Content-MD5, Content-Range, **Content-Type**, **Date**, **Expires**, **Last-Modified**, Link, Location, Proxy-Authenticate, Retry-After, **Server**, Set-Cookie, Status, WWW-Authenticate, ...

Servicios RESTful

Contrato de servicio

- Queda determinado por 3 elementos



Servicios RESTful

Contrato de servicio > Identificadores

`{scheme}://{authority}{path}?{query}`

- Los IDs se construyen a partir de una URI base
- Nombres, no acciones
- Minúsculas, '-' en lugar de '_'
- Subcolecciones con '/'
- Filtrado con '?'
- Versiones

`http://www.example.org/app/users/{userId}`

`http://www.example.org/app/users/{userId}/accounts/{accountId}`

`http://www.example.org/app/users?name=Pepe`

`http://www.example.org/app/v1/users`

Servicios RESTful

Contrato de servicio > Métodos

- Determinan las operaciones sobre los recursos
- Genéricas CRUD (Create-Read-Update-Delete)
- Se reaprovechan los métodos HTTP
 - GET: recupera un recurso o colección
 - POST: crea un nuevo recurso en colección
 - PUT: actualiza un recurso en colección
 - DELETE: elimina un recurso de colección

Servicios RESTful

Contrato de servicio > Representaciones

- Determina los tipos de datos (*media types*) que utilizan las operaciones del contrato

type/subtype [; parameter]

- En HTTP se usan las cabeceras
 - Accept: en peticiones, determina el tipo esperado
 - Content-Type: en peticiones/respuestas, determina el tipo de contenido

Accept: text/html; charset=UTF-8

Content-Type: application/json

Servicios RESTful

Contrato de servicio > Representaciones

- JSON (JavaScript Object Notation)
 - Formato de datos basado en texto, eficiente, para almacenamiento/trasferencia de datos
 - Soporta los tipos de datos: number, boolean, string, null, object, array

```
var conversation = {id: 1, title: 'family', owner: true, date:  
null, messages: [{id: 1, content: 'hello'}, {id: 2, content:  
'bye'}]};
```



JSON.stringify(conversation)

```
'{"id": 1, "title": "family", "owner": true, "date": null,  
"messages": [{"id": 1, "content": "hello"}, {"id": 2, "content":  
"bye"}]}'
```

Servicios RESTful

Contrato de servicio > Ejemplo

Servicio que sirva tareas

`/mistareas/tareas`

Servicios RESTful

Contrato de servicio > Ejemplo

Servicio que sirva tareas

/mistareas/tareas

- HTTP POST: se crea una nueva tarea con el contenido especificado en la petición HTTP

Req: HTTP POST /mistareas/tareas

```
{"titulo": "Tarea 1", "fecha": "25/03/2015"}
```

Resp: 200/201 – Ok/Created

```
{"id": "1234", "titulo": "Tarea 1", "fecha": "25/03/2015"}
```


Servicios RESTful

Contrato de servicio > Ejemplo

Servicio que sirva tareas

`/mistareas/tareas`

- HTTP POST: se crea una nueva tarea con el contenido especificado en la petición HTTP
- HTTP DELETE: elimina una tarea con la url especificada en la petición HTTP

Req: HTTP DELETE `/mistareas/tareas/1234`

Resp: 204 – No Content

Servicios RESTful

Contrato de servicio > Ejemplo

Servicio que sirva tareas

/mistareas/tareas

- HTTP POST: se crea una nueva tarea con el contenido especificado en la petición HTTP
- HTTP DELETE: elimina una tarea con la url especificada en la petición HTTP
- HTTP PUT: modifica una tarea. La tarea y los datos se especifican en la petición HTTP

Req: HTTP PUT /mistareas/tareas/1234

 {"titulo":"Tarea 1.1","fecha":"25/03/2015"}

Resp: 200 - Ok

 {"id":"1234","titulo":"Tarea 1.1","fecha":"25/03/2015"}

Servicios RESTful

Contrato de servicio > Ejemplo

Servicio que sirva tareas

/mistareas/tareas

- HTTP POST: se crea una nueva tarea con el contenido especificado en la petición HTTP
- HTTP DELETE: elimina una tarea con la url especificada en la petición HTTP
- HTTP PUT: modifica una tarea. La tarea y los datos se especifican en la petición HTTP
- HTTP GET: recupera la/s tarea/s que corresponde/n con la url especificada en la petición HTTP

Req: HTTP GET /mistareas/tareas

Resp: 200 - Ok

```
[{"id": "1", "titulo": "Tarea 1"},  
 {"id": "2", "titulo": "Tarea 2"}]
```

Servicios RESTful

Contrato de servicio > Ejemplo

Servicio que sirva tareas

/mistareas/tareas

- HTTP POST: se crea una nueva tarea con el contenido especificado en la petición HTTP
- HTTP DELETE: elimina una tarea con la url especificada en la petición HTTP
- HTTP PUT: modifica una tarea. La tarea y los datos se especifican en la petición HTTP
- HTTP GET: recupera la/s tarea/s que corresponde/n con la url especificada en la petición HTTP

```
Req: HTTP GET /mistareas/tareas/123    Resp: 200 - 0k
{"id":"1","titulo":"Tarea 1"}
```

Servicios RESTful

Contrato de servicio > Ejemplo

Servicio que sirva tareas

/mistareas/tareas

- HTTP POST: se crea una nueva tarea con el contenido especificado en la petición HTTP
- HTTP DELETE: elimina una tarea con la url especificada en la petición HTTP
- HTTP PUT: modifica una tarea. La tarea y los datos se especifican en la petición HTTP
- HTTP GET: recupera la/s tarea/s que corresponde/n con la url especificada en la petición HTTP

Req: HTTP GET /mistareas/tareas?
titulo=xxx&fecha=xxx

Resp: 200 - Ok
[{"id":"1","titulo":"Tarea 1"},
{"id":"2","titulo":"Tarea 2"}]

Servicios RESTful

Implementación con Node.js

[http.Server](#)

- Servidor HTTP básico en módulo '[http](#)':

```
var http = require('http');  
var server = http.createServer(callback);  
server.listen(port);
```

- `callback(req, res)` se invoca para cada petición de entrada
 - `req` ([http.IncomingMessage](#)) contiene toda la información sobre la petición HTTP
 - `res` ([http.ServerResponse](#)) proporciona utilidades para generar la respuesta HTTP

Servicios RESTful

Implementación con Node.js

[http.Server](#)

- Ejemplo

```
var http = require('http');
var server = http.createServer(function(req, res) {
  console.log('nueva petición HTTP');
  res.end('Hola mundo!');
});
server.listen(8080);
```

- http.Server es un EventEmitter: 'request', 'connection', 'close', 'clientError', ...

```
var http = require('http');
var server = http.createServer();
server.on('request', function(req, res) {
  console.log('nueva petición HTTP');
  res.end('Hola mundo!');
});
server.listen(8080);
```

Servicios RESTful

Implementación con Node.js

[http.IncomingMessage](#)

- Stream [Readable](#): puede leerse como un stream
- .httpVersion, .method ('GET', 'PUT', ...)
- .url: URL completa; puede analizarse con módulo '[url](#)'
- .headers (diccionario en minúsculas)

```
require('http').createServer(function(req, res) {  
  res.write('HTTP version: ' + req.httpVersion + '\n');  
  res.write('HTTP method: ' + req.method + '\n');  
  res.write('url: ' + req.url + '\n');  
  for (key in req.headers) {  
    res.write('- ' + key + ':' + req.headers[key] + '\n');  
  }  
  res.end();  
}).listen(8080);
```


Servicios RESTful

Implementación con Node.js

[http.ServerResponse](#)

- Stream [Writable](#) (.write(), .end())
- .statusCode, .statusMessage, .setHeader(name, value)
- .writeHead(statusCode[, statusMsg] [, headers])

```
require('http').createServer(function(req, res) {  
  var body = 'hello world';  
  res.writeHead(200, {      // debe ser el primer método  
    'Content-Length': body.length,  
    'Content-Type': 'text/plain' });  
  res.write(body);  
  res.end();  
}).listen(8080);
```

Servicios RESTful

Implementación con Node.js

Express

- Framework web minimalista para Node.js
- Extiende las capacidades básicas de [http.Server](http://nodejs.org/api/http.html) y proporciona un framework rico para desarrollar aplicaciones web muy eficientes
- <http://expressjs.com/>
- Instalación

```
> npm install express
```

Servicios RESTful

Implementación con Node.js

Express > La Aplicación

- Para inicializar el framework es necesario crear una aplicación (`express()`)

```
var express = require('express');  
var app = express();
```

- Después se añaden varias rutas (`app.get()/post()/put() /delete()/...`)

```
app.get('/', function(req, res) {  
  res.send('Hello world!');  
});
```

- Finalmente, se arranca la aplicación (`app.listen()`)

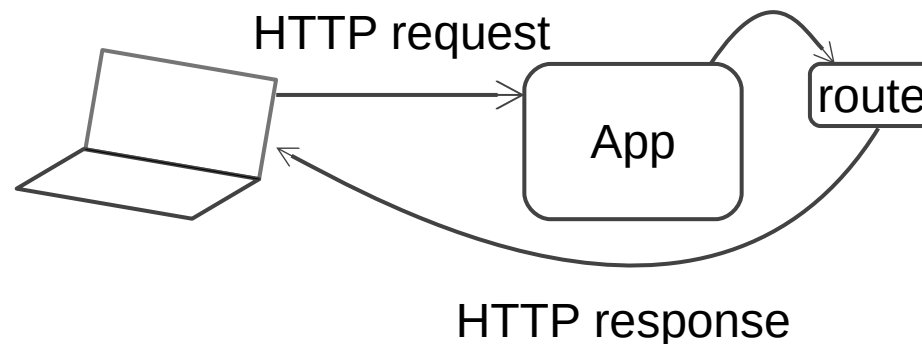
```
app.listen(8080);
```

Servicios RESTful

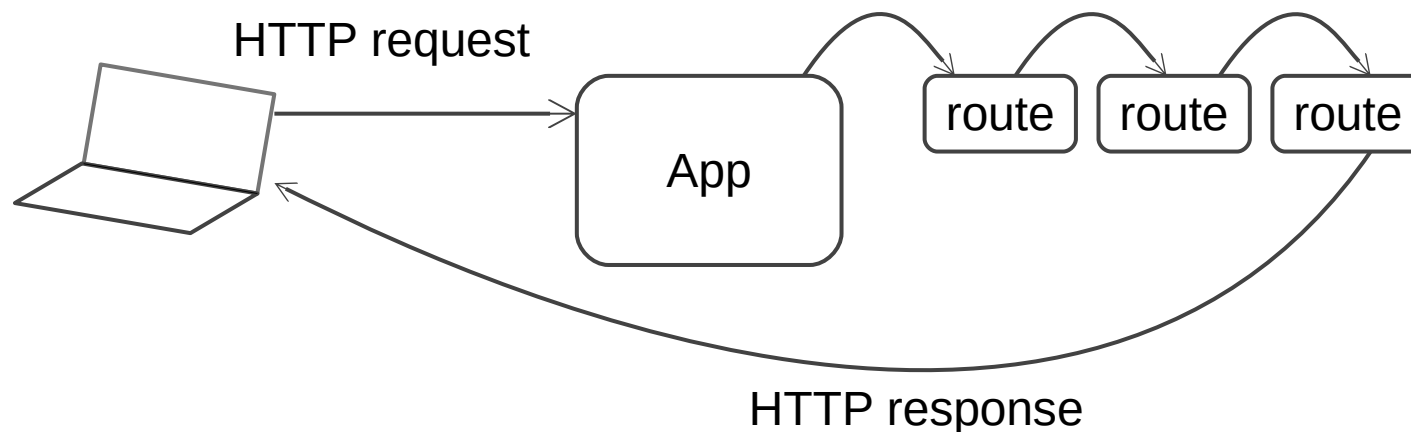
Implementación con Node.js

Express > La Aplicación

- Gráficamente



- Las rutas se pueden encadenar



Servicios RESTful

Implementación con Node.js

Express > Rutas

- Determinan quién recibe las peticiones HTTP
- La sintaxis a usar: `app.METHOD(path, callback)`
- METHOD: `get`, `post`, `put`, `delete`, ... **all**
- `callback(req, res [, next])`
 - req: Objeto [Request](#) de Express
 - res: Objeto [Response](#) de Express
 - next: siguiente callback a invocar (si hay rutas encadenadas)

Servicios RESTful

Implementación con Node.js

Express > Rutas

- Ejemplo de rutas encadenadas

```
var express = require('express');
var app = express();
app.get('/', function(req, res, next) {
    console.log('Primera ruta no hace nada');
    next();
});
app.get('/', function(req, res) {
    res.send('Finalizar la petición.');
```

```
});
app.get('/another', function(req, res) {
    res.send('Otra petición.');
```

```
});
app.listen(8080);
```

Servicios RESTful

Implementación con Node.js

Express > Rutas

Objeto [Request](#)

- Extiende [http.IncomingMessage](#): soporta todas sus operaciones y algunas más avanzadas
- .hostname, .ip, .originalUrl, .path, .protocol
- .query: query ya analizada

```
// GET /search?a=something&b=otherthing
req.originalUrl // => '/search?a=something&b=otherthing'
req.path        // => '/search'
req.query.a      // => 'something'
req.query.b      // => 'otherthing'
```

Servicios RESTful

Implementación con Node.js

Express > Rutas

Objeto Request

```
require('express')()
  .get('/', function(req, res) {
    // old properties
    res.write('HTTP version: ' + req.httpVersion + '\n');
    res.write('HTTP method: ' + req.method + '\n');
    res.write('url: ' + req.url + '\n');
    for (var key in req.headers) {
      res.write('- ' + key + ': ' + req.headers[key] + '\n');
    }
    // new properties
    res.write('hostname: ' + req.hostname + '\n');
    res.write('ip: ' + req.ip + '\n');
    res.write('path: ' + req.path + '\n');
    for (var key in req.query)
      res.write('- query.' + key + ': ' + req.query[key] + '\n');
    res.end();
  })
  .listen(8080);
```


Servicios RESTful

Implementación con Node.js

Express > Rutas

Objeto Request

- `params`: mapa de parámetros especificados en la ruta

```
app.get('/users/:name', function(req, res) {  
    console.log(req.params.name);  
});
```

- `.accepts/Charsets/Encodings/Languages(xxx)`
- `.get(field)`: para acceder a las cabeceras HTTP
- `.is(type)`: para acceder a la cabecera Content-Type

```
req.get('Content-Type')  
req.is('text/html')
```

Servicios RESTful

Implementación con Node.js

Express > Rutas

Objeto [Response](#)

- Extiende a [http.ServerResponse](#): soporta todas sus operaciones y algunas más avanzadas
- `.locals`: contiene información procesada en la petición actual (para compartir en una cadena de rutas)

```
app.get('/test', function(req, res, next) {  
    res.locals.info = 'This is a message'; // share info with next  
    next();  
});  
app.get('/test', function(req, res) {  
    res.send(res.locals.info); // use shared info  
});
```

- `.redirect([status], path)`: para redirigir al cliente a otra URL

Servicios RESTful

Implementación con Node.js

Express > Rutas

Objeto Response

- `.render(view, [, locals] [, callback])`: renderizar plantillas
- `.json(data)`: para enviar datos JSON
- `.send(data)`: envío sencillo de datos (fija automáticamente Content-Length/Type); data puede ser String (text/html), Buffer (application/octet-stream) o un objeto (application/json)
- `.download(path [,filename]) .sendFile(path [, options])`
- `.setStatus(statusCode), .status(code)`
- `.set(field [, val]), .type(type)`

Servicios RESTful

Implementación con Node.js

Express > Middleware

- Fragmento de software que se monta bajo una URL específica y que efectúa cierto proceso sobre las peticiones
- Un middleware se monta con `app.use([url,] middleware)`
- `middleware(req, res, next)`

```
var express = require('express');
var app = express();
// mount middleware
app.use(function(req, res, next) {
  console.log('Logging request: ' + req.url);
  next();
});
// define routes
app.get('/hello', function(req, res, next) {
  res.send('Hello world!');
});
app.listen(8080);
```

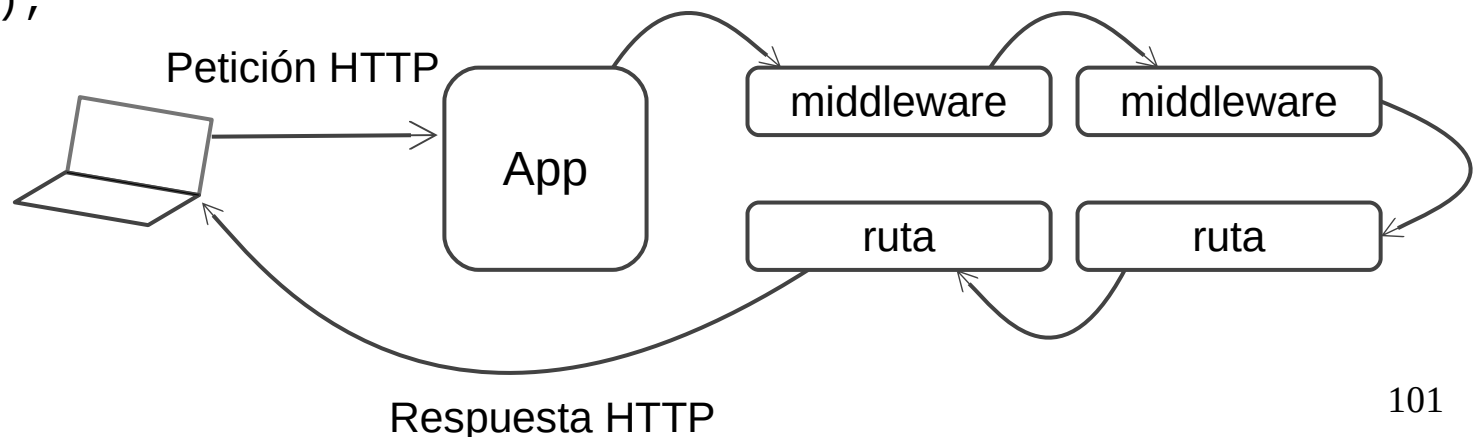
Servicios RESTful

Implementación con Node.js

Express > Middleware

- Los middleware deberían de montarse primero, y después deberían definirse el resto de rutas

```
var express = require('express');  
var app = express();  
// mount middlewares  
app.use(...); app.use(...); app.use(...);  
// define routes  
app.get(...); app.get(...); app.get(...);  
app.listen(8080);
```



Servicios RESTful

Implementación con Node.js

Express > Middleware

- En Express podemos encontrar gran variedad de [middlewares](#) que nos permiten construir aplicaciones más complejas (compresión, encriptación, cookies, sesiones, log, etc.)
- Nosotros veremos sólo unos pocos

Servicios RESTful

Implementación con Node.js

Express > Servir contenido estático

- Podemos implementarlo manualmente con rutas/middleware
- Express ya incorpora un middleware que lo hace por nosotros en [express.static\(root \[, options\]\)](#)

```
var express = require('express');  
var app = express();  
// mount middleware  
app.use\(express.static\('./public'\)\); //contenido estático en './public'  
app.listen(8080);
```

- En general los middleware son librerías externas. Éste middleware es una excepción

Servicios RESTful

Implementación con Node.js

Express > Directrices para implementar RESTful

- Conjunto de rutas que implementan una API REST

```
var express = require('express');  
var app = express();  
app.get('/mistareas/tareas', function(req, res) { ... });  
app.get('/mistareas/tareas/:id', function(req, res) { ... });  
app.post('/mistareas/tareas', function(req, res) { ... });  
app.put('/mistareas/tareas/:id', function(req, res) { ... });  
app.delete('/mistareas/tareas/:id', function(req, res) { ... });  
app.listen(8080);
```

- Cada ruta debe:
 - Procesar la información de entrada
 - Generar la información de salida

Servicios RESTful

Implementación con Node.js

Express > Procesar información de entrada

- Si es HTTP GET los datos están en la query de la URL: se recogen en [req.query](#)

```
app.get('/login', function(req, res) {  
  var user = req.query.user;  
  var passwd = req.query.password;  
});
```

- Si NO es HTTP GET los datos están en el cuerpo de la petición HTTP. Se pueden leer del stream req ([Readable](#)) y parsear (JSON, XML, ...)
- Es un poco costoso, mejor que lo haga alguien por nosotros: middleware [body-parser](#)

Servicios RESTful

Implementación con Node.js

Express > Procesar información de entrada

- Middleware [body-parser](#)
 - > npm install body-parser
- Proporciona varios parsers: JSON, raw, text, URL-encoded
- Cuando una petición HTTP lleva contenido, el middleware lo detecta, lo analiza y rellena el objeto [req.body](#)

Servicios RESTful

Implementación con Node.js

Express > Procesar información de entrada

HTTP POST 'http://localhost:8080/mistareas/tareas'
{ "id": "001", "titulo": "Tarea 1" }

Petición

Servidor

```
var express = require('express');
var bodyParser = require('body-parser');
var app = express();

app.use(bodyParser.json());
app.post('/mistareas/tareas', function(req, res) {
  console.log('id:' + req.body.id);
  console.log('titulo:' + req.body.titulo);
  res.send('ok');
});

app.listen(8080);
```

Servicios RESTful

Implementación con Node.js

Express > Procesar información de entrada

- En un API REST es muy común usar parámetros en las URLs (`req.params`)

```
app.get('/mistareas/tareas/:id', function(req, res) {  
  console.log(req.params.id);  
});
```

Servicios RESTful

Implementación con Node.js

Express > Generar la información de salida

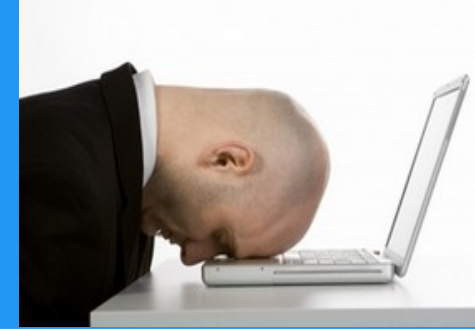
- Con los datos de entrada se procesa la petición
- Si hay que devolver contenido, se generan de alguna manera, típicamente consultando un almacén de datos
- La estrategia más sencilla para devolverlos es [res.send\(\)](#)

```
app.get('/mistareas/tareas/:id', function(req, res) {  
  var tarea = buscarPorId(req.params.id);  
  res.send(tarea);  
});
```

- Si no hay datos, se devuelve un código de estado

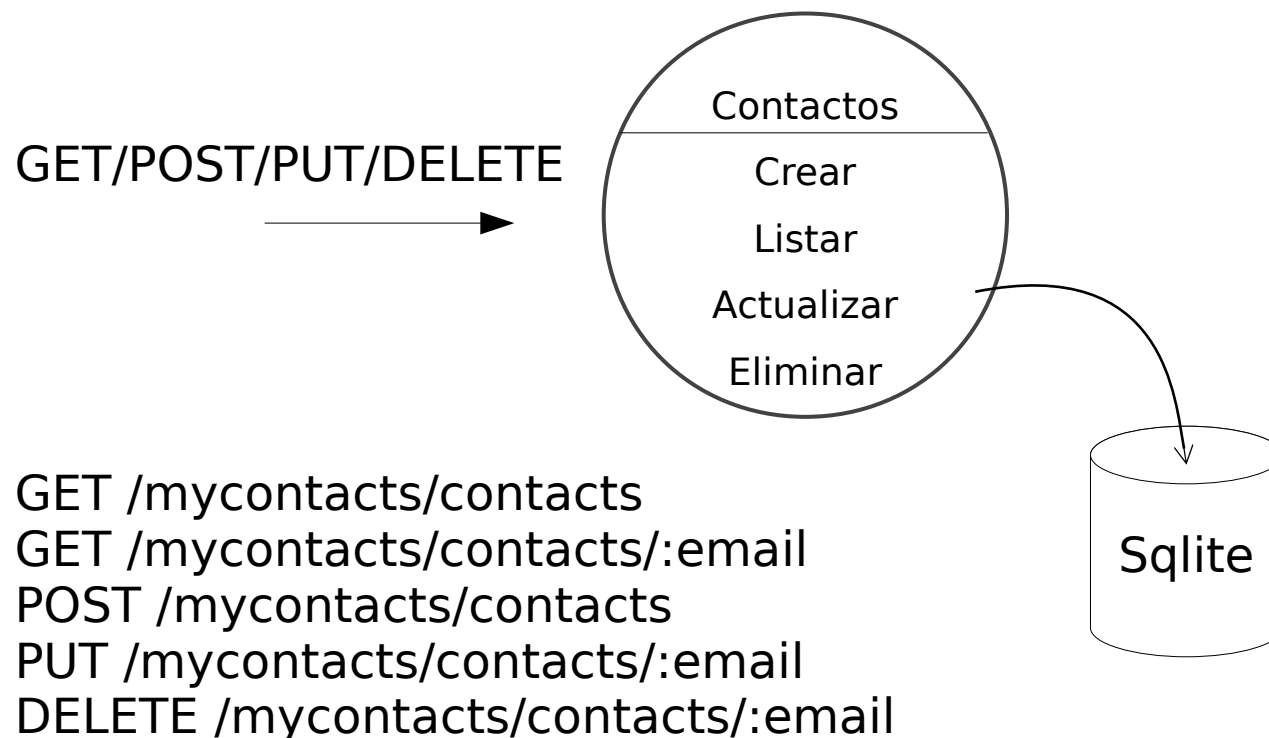
```
app.delete('/mistareas/tareas/:id', function(req, res) {  
  eliminarTarea(req.params.id);  
  res.status(204).send();  
});
```

Servicios RESTful



Ejercicio 3

Implementar un servicio RESTful para gestionar contactos



Servicios RESTful

Consumo de servicios

- Únicamente es necesario contar con un cliente HTTP
- Peticiones HTTP GET: navegador web
- Otras peticiones: [Postman](#), [Insomnia](#), ...
- En JavaScript:
 - (Node.js) Módulos core [http](#), [https](#)
 - (Node.js) Paquete [request](#)
 - Librería [axios](#)
 - (Web) API [Fetch](#)

Servicios RESTful

Consumo de servicios > Axios

- Se puede utilizar tanto en Node.js como en web
- Basado en promesas

```
> npm install axios
```

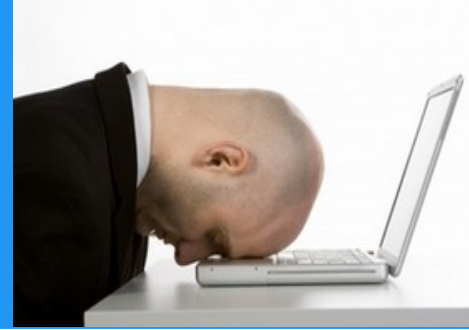
```
const axios = require('axios');
```

```
axios.get('http://www.example.org')  
  .then(response => {  
    console.log(response.data);  })  
  .catch(error => {  
    console.log(error);  
  });
```

```
<script src="https://unpkg.com/axios/  
dist/axios.min.js"></script>
```

```
<script>  
  axios.get('http://www.example.org')  
    .then(response => {  
      console.log(response.data);  })  
    .catch(error => {  
      console.log(error);  
    });  
</script>
```


Servicios RESTful



Ejercicio 4

Implementar una herramienta CLI que consuma el servicio RESTful de contactos

