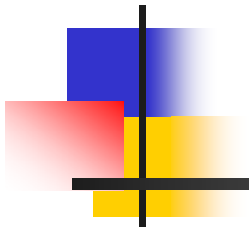
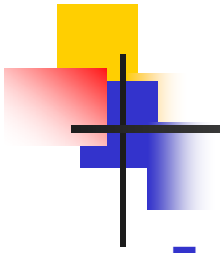
Tema 8: Diseño e Implementación de Servicios Web REST

8.1. Introducción a los Servicios Web REST
8.2. Caso de Estudio: Diseño e Implementación de un Servicio Web REST

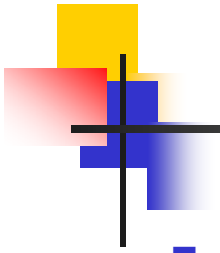


Tema 8.1: Introducción a los Servicios Web REST



Índice

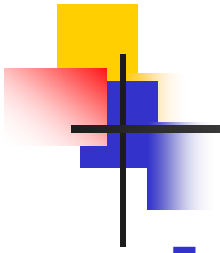
- Introducción
- Recordatorio de HTTP
- Características Servicios REST
  - Servicios “stateless”
  - Recursos y representaciones
  - Interfaz uniforme
  - Intermediarios
  - Hipermedia
  - Representaciones autodescriptivas
- REST en la práctica
- Ventajas e inconvenientes de REST



Introducción

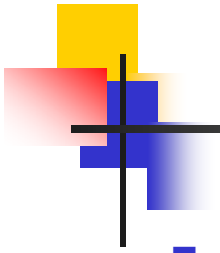
- REpresentational State Transfer. Estilo arquitectónico propuesto por Roy Fielding en 2000.
- La Web es, sin duda, la aplicación distribuida más exitosa de la historia.
- REST: Estilo arquitectónico para construir aplicaciones distribuidas inspirado en las características de la Web.

Introducción (y 2)



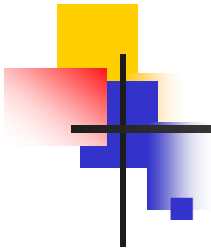
- Aunque es independiente de la tecnología, se suele implementar usando directamente HTTP y algún lenguaje de intercambio (como XML o JSON), sin tecnologías adicionales como SOAP o WSDL
  - A menudo los servicios REST reales no siguen estrictamente todos los principios del estilo arquitectónico REST
  - A los servicios que sí siguen fielmente todos los principios REST, se les denomina a veces servicios web RESTful

Recordatorio de HTTP



- HTTP: HyperText Transfer Protocol. Estandarizado por el W3C y la IETF.
- La versión actual es HTTP 1.1.
- Protocolo cliente/servidor utilizado en la Web.
- Inicialmente construido para transferir páginas HTML, pero puede transferir cualquier contenido textual
  - Usando MIME: contenidos binarios
- Esquema petición/respuesta
- Utiliza TCP para comunicar cliente y servidor (puerto reservado: 80)
- Desde HTTP 1.1, una conexión HTTP puede utilizarse para varias peticiones

Peticiones HTTP (1)

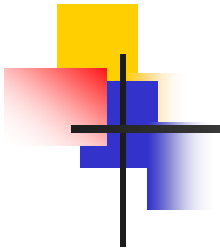


- Concepto clave: URL como identificador **global** de recursos
- Una petición HTTP consta de:
  - Una URL, que identifica al recurso sobre el que actúa la petición
  - Un método de acceso (GET, POST, PUT,...), que especifica la acción a realizar sobre el recurso
  - Cabeceras. Metainformación de la petición que indican información adicional que puede ser necesaria para procesar la petición (e.g. información de autenticación)
  - Cuerpo del mensaje (opcional). Sólo con algunos métodos de acceso

Peticiones HTTP (2)

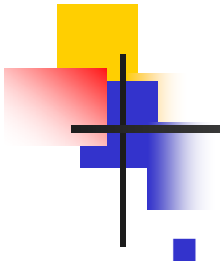
- Métodos de acceso:
  - GET:
    - Solicita una representación del recurso especificado (e.g. página HTML)
    - No debe causar “efectos secundarios” (modificaciones en el recurso)
    - Pueden especificarse parámetros en la URL (consultas). Se especifican como pares campo=valor, separados por el carácter ‘&’

      http://www.bookshop.com/search?tit=Java&author=John+Smith
  - PUT:
    - Carga en el servidor una representación de un recurso.
    - La nueva representación del recurso va en el cuerpo de la petición
  - DELETE:
    - Elimina el recurso especificado



Peticiones HTTP (3)

- POST
  - Envía datos al recurso indicado para que los procese
  - Los datos van en el cuerpo de la petición
  - Pueden causar efectos secundarios:
    - Crear un nuevo recurso
    - Modificar un recurso existente
- Peticiones GET, PUT, DELETE deben ser idempotentes
  - Múltiples peticiones iguales deben tener el mismo efecto que una sola
- GET no debe tener efectos secundarios (sólo lectura, sin coste,...)
- Las peticiones POST no tienen porque ser idempotentes y pueden tener efectos secundarios
- PUT y DELETE tienen efectos secundarios



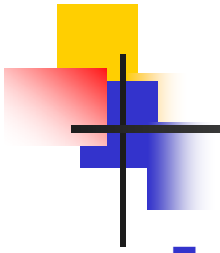
Peticiones HTTP (y 4)

- Cabeceras de la petición:
  - Tipo MIME de datos esperados (e.g. HTML, XML, JSON,...).
    - El cliente puede especificar qué formatos entiende y en que formato prefiere recibir la representación
  - Encoding esperado
  - Lenguaje esperado
  - Peticiones condicionales (If-Modified-Since, Etags):
    - Solicita una representación sólo si ha cambiado desde un momento determinado
    - Caches en el cliente
  - Credenciales de autenticación /autorización.
  - Información para proxies
  - Agente del usuario (e.g. navegador utilizado)
  - ...

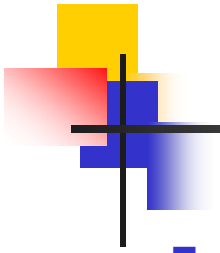
Respuestas HTTP (1)

- Una respuesta HTTP contiene:
  - Código de Status:
    - 200 OK
    - 201 Created
    - 400 Bad Request
    - 404 Not Found
    - 500 Internal Error
    - 403 Forbidden
  - Lista completa: http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html
  - Cabeceras. Metainformación de la respuesta.
  - Cuerpo del mensaje (opcional):
    - GET: representación del recurso invocado
    - POST: vacío o con el resultado del procesamiento realizado por el servidor
    - PUT: normalmente, el cuerpo viene vacío
    - DELETE: normalmente, el cuerpo viene vacío
    - En caso de error, puede incluir información con más detalle

Respuestas HTTP (y 2)

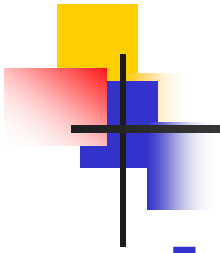


- Las cabeceras especifican metainformación adicional sobre las respuestas. Entre otras:
  - Tipo MIME de datos devueltos
  - Encoding de la respuesta
  - Lenguaje de la respuesta
  - Antigüedad de la respuesta
  - Longitud de la respuesta
  - Control de cache: si el recurso se puede cachear o no, tiempo de expiración (si se puede cachear), momento de última modificación,...
  - Control de reintentos
  - Identificación del servidor
  - ...



Características Servicios REST (1)

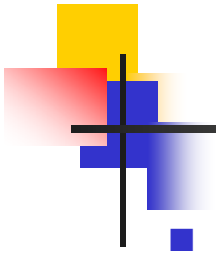
- Los servicios web REST se estructuran de forma similar a un sitio de la WWW
- Sin embargo, en lugar de páginas HTML, normalmente accederemos a información estructurada (e.g. películas, clientes, etc.)
- Al igual que la WWW está compuesta de muchos sitios web autónomos, los servicios web REST se consideran autónomos entre sí
- Al igual que en la WWW, un servicio REST puede referenciar a otro (links) y habrá una serie de servicios generales (e.g. caches) que pueden funcionar sobre cualquier servicio



Características Servicios REST(y 2)

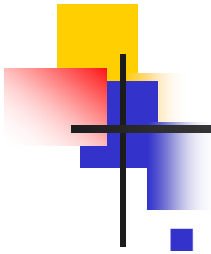
- Cliente – Servidor
- Servicios sin estado (“stateless”)
- Interfaz uniforme
- Sistema en capas: Intermediarios
- Hipermedia
- Representaciones autodescriptivas

Servicios “Stateless”(1)

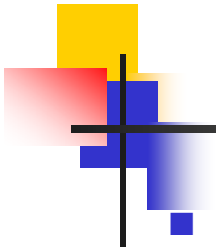


- Cliente- Servidor
  - Clientes invocan directamente URLs para acceder a recursos
- El servidor no guarda información de estado para cada cliente
- Ejemplo de servicio con estado: FTP
  - El servidor guarda el directorio en el que está cada cliente
  - El resultado de la ejecución de un comando (e.g. get, para obtener un fichero) depende de ese estado
  - Si el fichero `file.pdf` está en la ruta `/docs` y el cliente está en ese directorio el comando `get file.pdf` transferirá el fichero. En otro caso, dará error
- Ejemplo de servicio sin estado: HTTP
  - Cada petición de un cliente debe contener toda la información necesaria para que el servidor la responda
  - Ejemplo: `http://www.fileserver.com/docs/file.pdf`

Servicios “Stateless”(y 2)

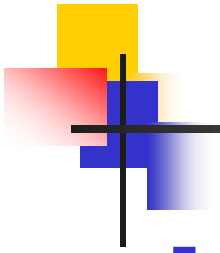


- Ventajas de los servicios sin estado:
  - Replicación del servicio en múltiples máquinas es muy sencilla: basta usar un balanceador de carga que dirige cada petición a una instancia del servicio
    - En un servicio con estado, es necesario o bien garantizar que todas la peticiones de la misma “sesión” van al mismo servidor o bien usar un servidor de sesiones
  - El servidor no necesita reservar recursos para cada sesión
    - Disminuye los recursos que necesita el servicio
    - Si el cliente falla o se cae en el medio de la sesión, el servidor no tiene que preocuparse de detectarlo ni de liberar recursos
- En general: mejora la escalabilidad de los servicios
- Inconveniente: puede ser necesario enviar información adicional con cada petición



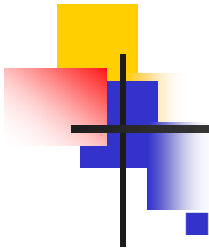
Recursos y Representaciones (1)

- Una aplicación REST se compone de recursos
- Suelen corresponderse con las entidades persistentes
- Hay dos tipos de recursos:
  - Colección: identifican una serie de recursos del mismo tipo (películas en nuestro ejemplo, clientes de una empresa, ...)
  - Individuales: identifican un elemento concreto (una película, un cliente,...)
- Cada recurso es identificado mediante un identificador único y global (típicamente URLs):
  - http://www.movieprovider.com/movies/
    - Recurso colección que representa todas las películas
  - http://www.movieprovider.com/movies/3
    - Una película de movieprovider.com
  - http://www.acme.com/customers/01235
    - Un cliente de acme.com



Recursos y Representaciones (2)

- Los identificadores (URLs) son globales.
  - Todo recurso tiene un nombre único a nivel inter-aplicación
  - No existen espacios de nombres restringidos a un servicio/aplicación: no puede haber dos recursos en servicios distintos con el mismo identificador
  - Por tanto, cualquier servicio puede referenciar y acceder a un recurso de cualquier otro servicio:
    - Eso no quiere decir que todos los recursos sean accesibles para todos (mecanismos de autorización)



Recursos y Representaciones (3)

- Al invocar la URL (usando GET), el cliente obtiene una **representación** del recurso
- La representación debe contener información útil sobre el recurso:
  - Recursos colección: normalmente lista de los elementos de la colección con información resumen y un enlace para obtener la información completa
  - Recursos individuales: datos del elemento individual (e.g. datos de la película, datos del cliente,...)
- La representación de un recurso puede variar en el tiempo.
  - El identificador está ligado al recurso, no a la representación
  - Si cambian los datos de un cliente de Acme, cambiará la representación. La URL apuntará siempre a la representación actual del recurso

Recursos y Representaciones (4)

- Ejemplo representación recurso colección:

```
<?xml version="1.0" encoding="UTF-8"?>
<customers xmlns="http://www.acme.com/restws/customers"
   xmlns:atom="http://www.w3.org/2005/Atom" >

   <customer>
       <cid>1234</cid>
       <cname>Roadrunner</cname>
       <atom:link rel="self“
   href="http://www.acme.com/restws/customers/1234"/>
   </customer>
   <customer>
       <cid>1235</cid>
       <cname>Coyote</cname>
       <atom:link rel="self"
   href="http://www.acme.com/restws/customers/1235"/>
   </customer>
   …
</customers>
```

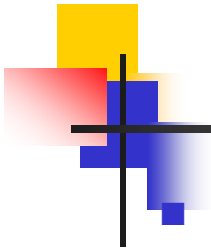
NOTA: Para indicar los links, usamos el tag `link` del estándar ATOM (comentaremos algo más adelante sobre él)

Recursos y Representaciones (y 5)

- Ejemplo representación recurso individual:

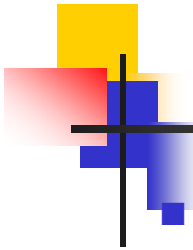
```
<?xml version="1.0" encoding="UTF-8"?>
<customer xmlns="http://www.acme.com/restws/customers">
    <cid>1234</cid>
    <cname>Roadrunner</cname>
    <address> Monument Valley, 5 </address>
    <description> He is fast</description>
    <rating> Good </rating>
</customer>
```

Interfaz Uniforme (1)



- Las operaciones disponibles sobre los recursos son siempre las mismas
- Las normas REST no dicen cuáles deben ser esas operaciones, sólo que deben ser siempre las mismas y funcionar igual en todos los servicios
- Los tipos de respuesta (tanto de éxito como de error) que pueden devolver estas operaciones deben estar también estandarizados
- Nuevamente, REST no impone un conjunto de códigos de respuesta concretos, sólo que deben ser los mismos para todos los servicios

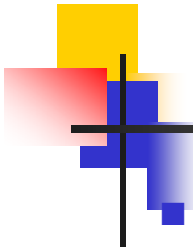
Interfaz Uniforme (2)



- En la práctica, se utiliza HTTP:
  - GET:
    - Acceso a representaciones
    - Consultas sobre recursos colección

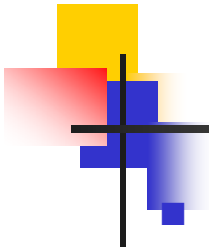
      http://www.movieprovider.com/movies?keywords=Dark+Knight
    - Sin efectos secundarios
  - PUT:
    - Reemplaza la representación de un recurso
    - Si la URL no existe, y el servicio lo permite, crea el recurso
    - No suele permitirse sobre recursos colección
    - Idempotente

Interfaz Uniforme (3)



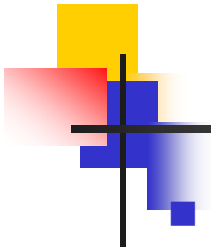
- En la práctica, se utiliza HTTP (cont.):
  - DELETE:
    - Borra el recurso
    - No suele permitirse sobre recursos colección
    - Idempotente
  - POST
    - Sobre recursos colección, suele usarse para crear un nuevo recurso en la colección (e.g. añadir una película)
      - El servidor devuelve la URL del nuevo elemento usando una cabecera estándar HTTP
    - Sobre recursos individuales, puede usarse para modelar operaciones con efectos secundarios que no encajen con los otros métodos
    - Puede no ser idempotente

Interfaz Uniforme (y 4)



- Códigos de respuesta estándar HTTP
  - 200 – OK
  - 201 - Created
  - 400 – Bad Request
  - 404 – Not Found
  - 500 – Internal Error
  - 403 – Forbidden
  - ...
- Cabeceras HTTP estándar para enviar información adicional en la petición y en las respuestas:
  - Autenticación
  - Formatos aceptados /enviados
  - Manejo de caches: tiempos de expiración, soporte para peticiones condicionales,...
  - ...

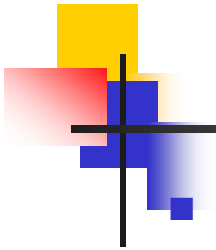
Intermediarios (1)



- El uso de una interfaz uniforme permite que haya intermediarios entre el cliente y el servicio que proporcionan funcionalidad adicional
- Los intermediarios son transparentes para el cliente y el servicio
- Los intermediarios pueden proporcionar funcionalidad adicional a cualquier servicio y a cualquier cliente sin necesidad de saber nada adicional sobre ellos:
  - Esto es posible porque todos los servicios soportan una interfaz uniforme (operaciones, códigos de respuesta, cabeceras)

Intermediarios (2)

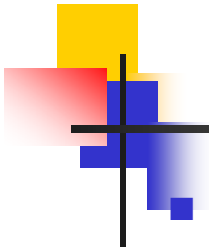
- Ejemplo 1: Servidores de Cache intermedios
- En la WWW y en los servicios REST pueden existir múltiples servidores de cache entre cliente y servicio. Ejemplos:
  - Los grandes servicios de Internet (e.g. Google) pueden usar DNS para redirigir a los clientes a servidores cache de acuerdo a su zona geográfica
  - El administrador de la red de una organización (e.g. UDC) puede instalar un proxy que proporcione servicio de caching para optimizar tiempos y ancho de banda
  - Los servidores de cache sirven una copia del recurso solicitado si la tienen y, si no, invocan al sitio web/servicio real



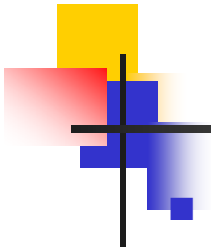
Intermediarios (3)

- Las caches funcionan con cualquier servicio y cliente sin necesidad de ninguna pre-configuración ni en el servicio ni en el sistema cache
- La WWW y los servicios REST tienen esta propiedad porque la interfaz uniforme proporciona la información necesaria para el intermediario:
  - Petición GET de un recurso no invalida copia en cache
  - Peticiones POST, PUT, DELETE sí que invalidan
  - En general, no se cachean peticiones POST ni PUT, DELETE
  - No se deben cachear respuestas que indiquen un código de error temporal (e.g. 500 Internal Error) pero sí los que indiquen errores permanentes (e.g. 400 Bad Request)
  - Cabeceras de cache proporcionan soporte genérico para indicar tiempo de expiración, recursos que no deben cachearse, peticiones condicionales,...

Intermediarios (4)



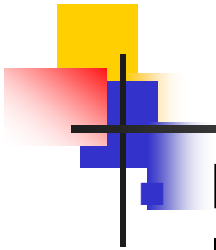
- Otros ejemplos de intermediarios soportados por la WWW y por los servicios REST:
  - Proxies. Pueden reintentar transparentemente peticiones para proporcionar tolerancia a fallos
    - Necesitan saber si una petición es idempotente o no
  - Traducción de formatos. Traducción transparente a un formato de representación no soportado por el servicio.
    - Ejemplos: cliente espera XML y servicio proporciona JSON
      - Cliente indica en la petición que acepta sólo XML
      - Como el intermediario sabe que es capaz de traducir de JSON a XML, modifica la petición para añadir JSON como formato aceptado
      - El servicio responde indicando que el formato es JSON
      - El intermediario transforma de JSON a XML y se lo envía al cliente
    - Necesitan una manera estándar de especificar los distintos formatos (tipos MIME), saber qué formatos acepta el cliente (cabeceras Accept) y en que formato viene la respuesta del servidor (cabecera Content-type)



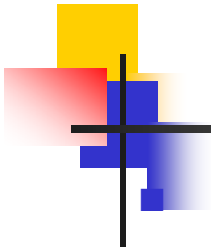
Intermediarios (5)

- Otros ejemplos de intermediarios soportados por la WWW y por los servicios REST (cont.):
  - Seguridad
    - Supongamos que tenemos un servicio ya hecho y difícil de modificar, que no tiene control de acceso
    - Deseamos exponer su funcionalidad a otras aplicaciones pero aplicando políticas de control de acceso
    - Es posible añadir un intermediario que permita el acceso sólo a ciertos clientes
    - Necesita que el formato en el que se envía información de autenticación / autorización sea estándar (cabeceras HTTP)

Intermediarios (6)



- La idea de interfaz uniforme / intermediarios puede llevarse más allá del estándar HTTP
- Si imponemos restricciones adicionales (e.g. dentro de los servicios de una empresa) podemos tener nuevos tipos de intermediarios
- La clave siempre es que la semántica de las peticiones y respuestas sea estándar, de forma que los intermediarios pueden entenderla

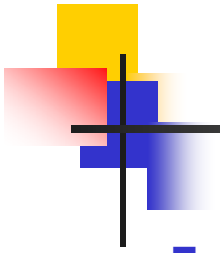


Intermediarios (7)

Ejemplo: Intermediarios más allá de HTTP

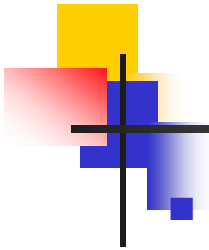
- Restricción adicional: en nuestra empresa las direcciones postales que van en las representaciones de los recursos (e.g. clientes, sucursales, proveedores,...) se representan siempre igual (e.g. con un cierto tag XML definido en un espacio de nombres determinado)
- Supongamos que nos gustaría que todas las direcciones ahora incluyan información de geolocalización (e.g. usando el API de Google Maps)
- Opción 1: modifico todos los servicios que dan información de direcciones para que accedan a Google Maps
- Opción 2: los clientes que quieran esta funcionalidad acceden a los recursos a través de un intermediario. Este intermediario cada vez que reconoce una dirección en la representación, invoca a Google Maps y añade la información de geolocalización a la representación del recurso

Intermediarios (y 8)

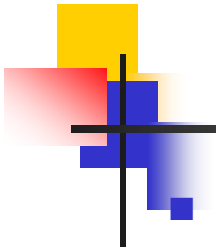


- En el esquema RPC (y en servicios web SOAP) la semántica de las peticiones y respuestas es opaca (específica de cada servicio)
- Ejemplo: un intermediario de cache no tiene forma de saber si la operación `findMovies` es de lectura (y por tanto cacheable) ni de saber si las excepciones que lanza se corresponden con errores permanentes (cacheables) o temporales (no cacheables)
- Habría que configurar cada servidor cache indicando si cada operación de cada servicio es de lectura o no, si debe cachearse o no, si cada tipo de respuesta debe cachearse o no,...

Hipermedia: Cambio de Estado (1)



- La representación recibida por el cliente cambia (**transfer**) su estado (**state**).
- La representación puede contener nuevas URLs hacia otros recursos
  - Ejemplo: La representación de un cliente puede incluir un enlace que permite acceder a sus pedidos
- O controles (similar a formularios) para actuar sobre el recurso
  - Ejemplo: La representación de un producto puede incluir un control que permite modificarlo



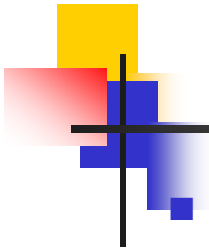
Hipermedia: Cambio de Estado (2)

- Una aplicación REST (web) puede verse como el grafo de transición de estados de un autómata.
  - Las representaciones de recursos (páginas) son estados del autómata.
  - Las URLs (hiperenlaces) y controles son transiciones entre estados.
  - El estado en el que estamos determina qué otros estados tenemos accesibles.
- Esta idea suele resumirse con el acrónimo HATEOAS (Hypermedia As The Engine Of Application State)

Hipermedia (3)

- Ejemplo representación recurso con enlaces a otros recursos:

```
<?xml version="1.0" encoding="UTF-8"?>
<customer xmlns="http://www.acme.com/restws/customers">
    <cid>1234</cid>
    <cname>Roadrunner</cname>
    <address> Monument Valley, 5 </address>
    <description> He is fast</description>
    <rating> Good </rating>
    <atom:link rel=“stockprice"
           title=“Stock price of this customer"

   href=“http://www.financeinformation.com/stockprices?symbol=RRN"/>
    <atom:link rel=“orders"
           title="Orders of this customer" href="1234/custOrders"/>
</customer>
```



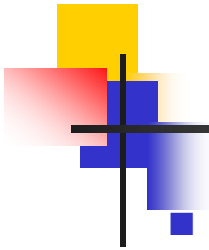
Hipermedia (y 4)

- Usar hiperenlaces oculta detalles de la implementación del servicio. En nuestro ejemplo:
  - El cliente debe entender los valores especificados en el atributo `rel` de los links (semántica del link), pero es independiente de la url del link
  - Si pasamos a obtener la información de cotizaciones de otro servicio, el cliente no se ve afectado (siempre que también entienda el formato del nuevo servicio)
  - Es independiente de cambios en el formato de la URL
- El uso de hiperenlaces también permite que los clientes descubran nueva información. Ejemplo:
  - Crawler (tipo Google) que construye un buscador sobre todos nuestros servicios, atravesando enlaces automáticamente
- No mostramos el uso de controles para operaciones con efectos secundarios. Las ventajas son similares

Uso de Representaciones Autodescriptivas

- Las representaciones devueltas deben intentar expresarse en formatos conocidos a priori por los clientes posibles del servicio:
  - Si los clientes posibles están dentro de una organización intentar usar estándares de la organización
  - Si es posible, utilizar formatos estándar (e.g. ATOM)
    - ATOM es un formato para representar colecciones de recursos. Sobre cada recurso ofrece sólo información básica (nombre, descripción, url, fecha,...) pero se puede extender con campos adicionales específicos de cada tipo de recurso
    - Así cualquier cliente que entienda ATOM podrá entender algunos campos de cualquier recurso:
      - Ejemplo: un crawler puede obtener la descripción de cualquier recurso, aunque no sepa interpretar los campos específicos
    - Los clientes que estén preprogramados para entender nuestro formato podrán beneficiarse igualmente de la información detallada
- La ventaja es aumentar la visibilidad para clientes e intermediarios (como en el ejemplo de geolocalización)

REST en la práctica (1)

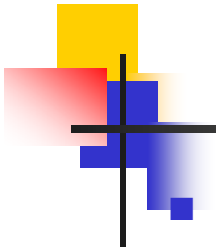


- La mayoría de servicios reales no respetan todos los principios de REST
- Servicios REST “antiguos”:
  - Uso consistente de las operaciones GET y POST
  - Uso consistente de los códigos de respuesta HTTP y de algunas cabeceras estándar HTTP
  - Suficiente para beneficiarse de algunas ventajas (e.g. caches)
  - No se usan URLs para identificar recursos individuales
  - No se usa hipermedia ni representaciones autodescriptivas

REST en la práctica (y 2)

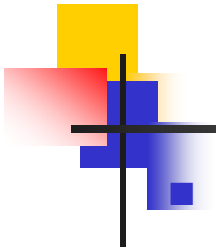
- Servicios REST “recientes”:
  - URLs únicas y globales para cada recurso
  - Uso consistente de GET, PUT, POST y DELETE
  - Uso consistente de los códigos de respuesta HTTP y de muchas cabeceras estándar HTTP
  - Uso limitado o nulo de hipermedia
    - Si cambia la URL muchas veces cambia también el formato de representación, por lo que la ventaja puede diluirse
    - No hay estándares claros:
      - Se suele usar el elemento link de ATOM para los enlaces pero apenas hay opciones para especificar controles para acciones con efectos secundarios o consultas que requieran parámetros
    - Más ineficiente: requiere más peticiones HTTP
  - Uso limitado o nulo de representaciones autodescriptivas
  - Nosotros usaremos un enfoque similar a este en el ejemplo y la práctica

REST: Ventajas (1)

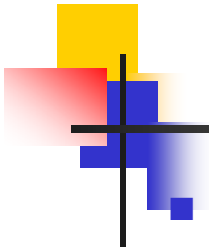


- Stateless: facilita escalabilidad y tolerancia a fallos
- Identificadores únicos y globales para cada recurso:
  - Proporcionan una manera de expresar relaciones entre servicios creados independientemente
  - Cada elemento de datos útil para la empresa tiene un localizador global, independiente de la aplicación que cualquiera puede usar para acceder a información útil
- Intermediarios pueden proporcionar valor añadido a cualquier servicio:
  - Escalabilidad y tolerancia a fallos: caches, proxies,...
  - Seguridad
  - Traducción de formatos
  - Con representaciones autodescriptivas, filtros o enriquecedores de contenidos
  - ...

REST: Ventajas (y 2)



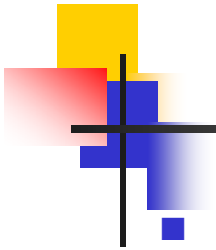
- Se maximiza el desacoplamiento entre servicios y clientes:
  - HATEOAS: Cliente no necesita conocer formatos de URL ni depende de la ruta a los servicios concretos referenciados
  - Uso de operaciones estándar, códigos de error estándar y de formatos estándar hace que cualquier cliente pueda procesar (hasta cierto punto) la salida de cualquier servicio.
    - E.g. crawler / buscador a lo Google para los servicios de nuestra empresa
- En general las ventajas de REST son más evidentes cuanto más se parezca nuestro escenario a la Web:
  - Cientos o miles de servicios y clientes autónomos, sin control centralizado (e.g. una gran multinacional con múltiples departamentos que crean servicios y clientes)
  - Aún así, queremos ser capaces de proporcionar ciertos servicios generales de apoyo para todos esos servicios y clientes (caches, buscadores, etc.)



REST: Inconvenientes

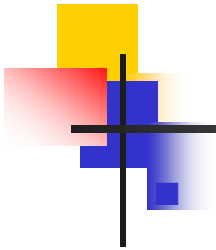
- La interfaz uniforme puede ser restrictiva.
  - No siempre es fácil modelar las operaciones de un servicio en términos de las operaciones de HTTP
  - Hay guías de buenas prácticas para representar operaciones que no encajan directamente con las de HTTP
- HATEOAS puede ser ineficiente ya que requiere de peticiones HTTP adicionales (y sin formatos estándar, la ventaja se atenúa)
- El uso de representaciones autodescriptivas (o cualquier extensión de la interfaz uniforme) en un ámbito determinado (e.g. gran empresa) puede ser difícil de implantar

REST vs SOAP (1)



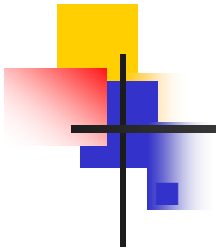
- Los servicios SOAP también suelen ser stateless
- Sin embargo:
  - No existe el concepto de identificador global. Normalmente los identificadores son locales a cada servicio
  - No se soportan intermediaros transparentes, ya que la semántica de las operaciones es opaca (específica de cada servicio)
    - Aunque se use HTTP como transporte, se usa POST para todo, sea cuál sea la semántica de la operación
  - No existe el concepto de hipermedia

REST vs SOAP (2)

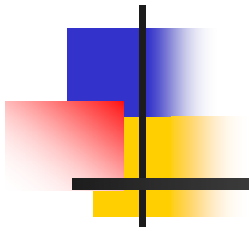


- En SOAP (y tecnologías RPC en general), el acoplamiento del cliente con el servicio es muy elevado:
  - Casi cualquier cambio en la interfaz del servicio (incluso si se elimina o se añade un campo que nuestro cliente no usa), suele obligar a regenerar los stubs y recompilar
  - Con una librería local, está bajo nuestro control instalar o no una nueva versión, pero los cambios en una aplicación remota y autónoma no están bajo nuestro control
  - Como veremos, con las tecnologías usadas habitualmente con REST, puede conseguirse fácilmente que ciertos cambios no nos afecten

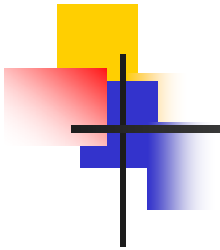
REST vs SOAP (y 3)



- En cuanto a las tecnologías, sin tener en cuenta las diferencias arquitectónicas:
  - El uso de stubs/skeletons es más amigable al programador:
    - Como veremos, con REST suele usarse un enfoque de desarrollo de más bajo nivel
  - Pero el uso de stubs/skeletons complica el entorno de desarrollo y no es fácil utilizarlo en ciertos entornos (e.g. código javascript que se ejecuta dentro de un navegador)
    - REST es fácil de usar desde cualquier sitio que pueda emitir peticiones HTTP (prácticamente cualquier entorno, y cualquier herramienta)

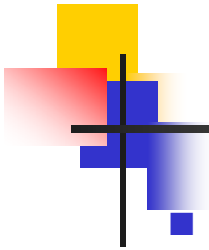


Tema 8.2. Caso de Estudio: Diseño e Implementación de un Servicio REST



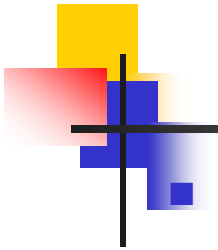
Índice

- Protocolo REST
- Implementación REST de la capa Acceso al Servicio
- Implementación REST de la capa Implementación del Servicio
- Comentarios finales



Protocolo REST (1)

- Seguiremos la misma aproximación que la mayoría de servicios web REST “recientes”:
  - URLs únicas y globales para cada recurso
  - Uso consistente de GET, PUT, POST y DELETE
  - Uso consistente de los códigos de respuesta HTTP
  - Uso de algunas cabeceras estándar HTTP
  - No usaremos hipermedia ni representaciones autodescriptivas
  - En la asignatura Integración de Aplicaciones se profundizará sobre el uso de estos conceptos



Protocolo REST (2)

- Recursos:
  - `/movies` Recurso colección.
    - GET
      - Lista todas las películas
      - La información de cada película se representa en XML en el formato del apartado 6.2
      - El parámetro keywords permite filtrarlas por palabras clave en el título: `/movies?keywords=Dark+Knight`
    - POST
      - Añade una nueva película
      - La película se envía en el cuerpo de la petición en el formato XML del apartado 6.2 (sin identificador)
      - Devuelve el código de respuesta HTTP: `201 Created`
      - Devuelve la URL de la nueva película usando la cabecera estándar HTTP `Location`
      - El cuerpo de la respuesta devuelve la nueva película creada

Protocolo REST (3)

Petición GET a http://XXX/ws-movies-service/movies?keywords=Dark+Night


```
HTTP/1.1 200 OK
…

<?xml version="1.0" encoding="UTF-8"?>
<movies xmlns="http://ws.udc.es/movies/xml">
    <movie>
        <movieId>3</movieId>
        <title>Dark Knight Rises Again</title>
        ...
    </movie>
    <movie>
        <movieId>5</movieId>
        <title>Dark Knight Returns</title>
        ...
    </movie>
</movies>
```

Protocolo REST (4)

Añadir la información de una película

Petición POST a http://XXX/ws-movies-service/movies

```
<?xml version="1.0" encoding="UTF-8"?>
<movie xmlns="http://ws.udc.es/movies/xml">
    <title>Dark Knight Rises Again</title>
    <runtime>165</runtime>
    ...
</movie>
```


```
HTTP/1.1 201 Created
…
Location: http://XXX/ws-movies-service/movies/3

<?xml version="1.0" encoding="UTF-8"?>
<movie>
    <movieId>3</movieId>
    <title>Dark Knight Rises Again</title>
    ...
</movie>
```

Protocolo REST (5)

- Recursos:
  - `/movies/{id}` Recurso individual por película
    - GET
      - Obtiene la información de la película
      - La información de cada película se representa en XML en el formato del apartado 6.2
    - PUT
      - Modifica la película
      - La película se envía en el cuerpo de la petición en el formato XML del apartado 6.2
      - El cuerpo de la respuesta va vacío
      - Devuelve el código `204 No Content`
    - DELETE
      - Borra la película
      - El cuerpo de la respuesta va vacío
      - Devuelve el código `204 No Content`

Protocolo REST (6)

Petición GET a http://XXX/ws-movies-service/movies/3


```
HTTP/1.1 200 OK
…

<?xml version="1.0" encoding="UTF-8"?>
<movie xmlns="http://ws.udc.es/movies/xml">
    <movieId>3</movieId>
    <title>Dark Knight Rises Again</title>
     ...
</movie>
```

Protocolo REST (7)

- Actualizar la información de una película

Petición PUT a http://XXX/ws-movies-service/movies/3

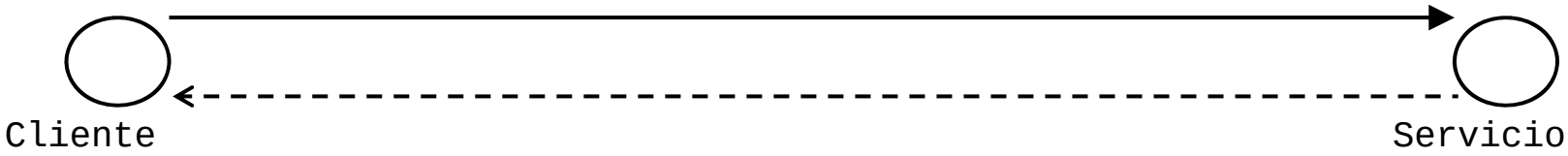
```
<?xml version="1.0" encoding="UTF-8"?>
<movie xmlns="http://ws.udc.es/movies/xml">
    <movieId>3</movieId>
    <title>Dark Knight Rises Again</title>
    <runtime>164</runtime>
    ...
</movie>
```


```
HTTP/1.1 204 No Content
…
```

Protocolo REST (8)

- Eliminar la información de una película

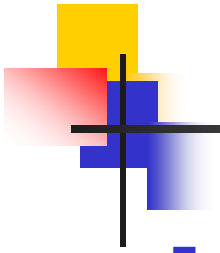
```
Petición DELETE a http://XXX/ws-movies-service/movies/3
```




```
HTTP/1.1 204 No Content
…
```

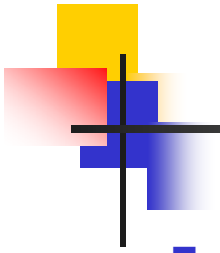
Protocolo REST (9)

- Recursos:
  - `/sales` Recurso colección
    - POST
      - Añade una nueva venta (equivalente a comprar película)
      - Los datos de la venta (`userId`, `movieId`, `creditCardNumber`) se reciben como parámetros
      - Devuelve el código de respuesta HTTP: `201 Created`
      - Devuelve la URL de la nueva película usando la cabecera estándar HTTP `Location`
      - El cuerpo de la respuesta devuelve los datos de la nueva venta creada
  - `/sales/{id}` Recurso individual por venta
    - GET
      - Obtiene la información de la venta
      - La información de cada venta se representa en XML
    - No es posible borrar ni modificar ventas una vez producidas



Protocolo REST (y 10)

- Errores. Se utilizan los códigos HTTP más próximos a la semántica de la respuesta:
  - Parámetros incorrectos: 400 Bad Request
    - Similar a InputValidationException
  - Recurso no existe: 404 Not Found
    - Similar a InstanceNotFoundException
  - Recurso existió pero ya no existe: 410 Gone
    - Similar a SaleExpirationException
  - Error interno de ejecución: 500 Internal Error
- Pero el cuerpo del mensaje puede llevar información adicional:
  - Representación en XML de los datos de la excepción



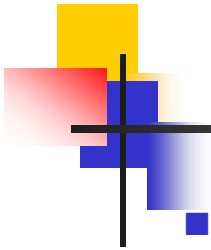
Consideraciones Diseño REST (1)

- Creación de recursos con POST:
  - La URL en la cabecera Location permite al cliente conocer la URL del nuevo recurso creado para referirse a él más tarde
  - Usar una cabecera estándar permite proporcionar semántica para cualquier intermediario y cliente, aunque no conozcan los formatos de nuestro servicio
    - Ejemplo: intermediario que hace transparentemente copia de seguridad de los recursos que se crean
  - Devolver el nuevo recurso creado en el cuerpo es útil si creemos que el cliente va a utilizarlo de inmediato (ahorra al cliente una petición HTTP)
  - ... pero si la representación puede ser grande y no es seguro que el cliente la necesite inmediatamente, puede ser mejor enviar sólo la URL

Consideraciones Diseño REST (2)

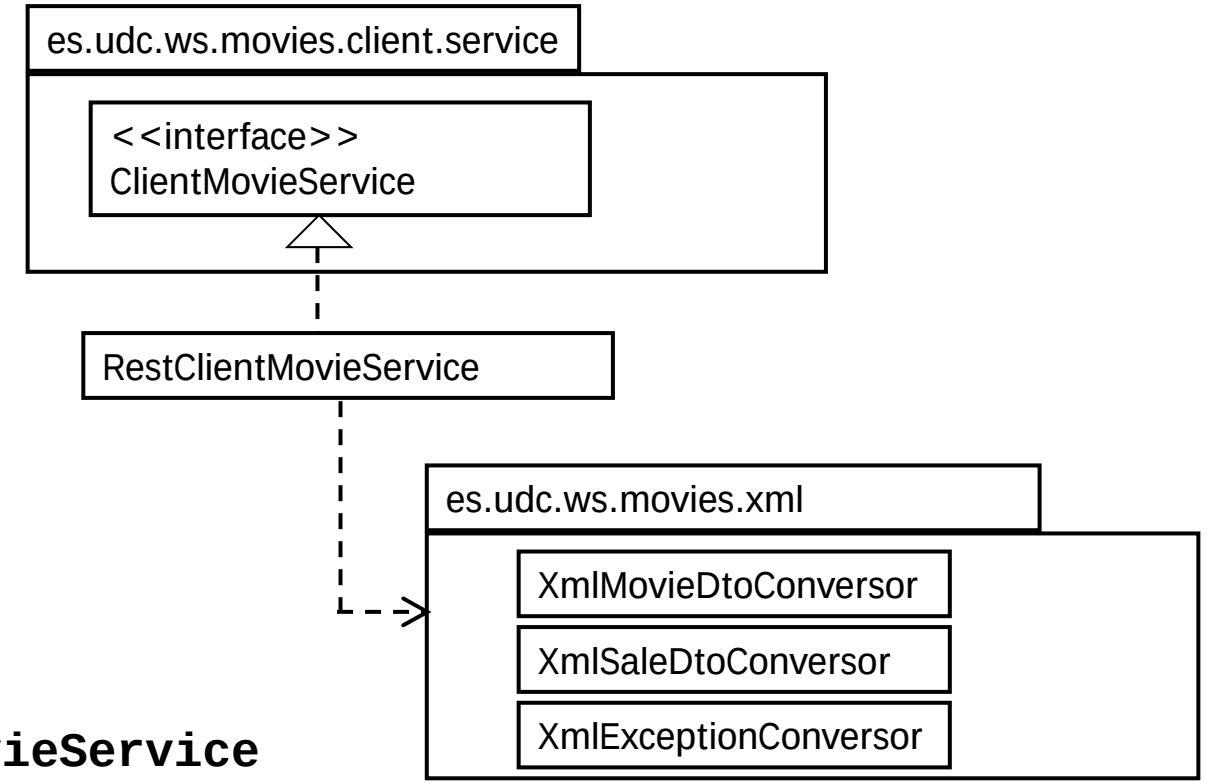
- Posible Alternativa para recursos de ventas:
  - `/movies/{id}/sales` Ventas de una película
    - POST: Crea una nueva venta de esa película
  - `/movies/{id}/sales/{id}`
    - GET: Obtiene los datos de la venta
- Este diseño considera a la venta como dependiente de la película a la que se refiere:
  - Representa de forma natural el que una venta siempre va asociada a una película
  - ... pero no permitiría representar fácilmente consultas sobre todas las ventas (algo muy habitual)
  - Por eso, en este caso hemos considerado más adecuado el diseño anterior

Consideraciones Diseño REST (y 3)



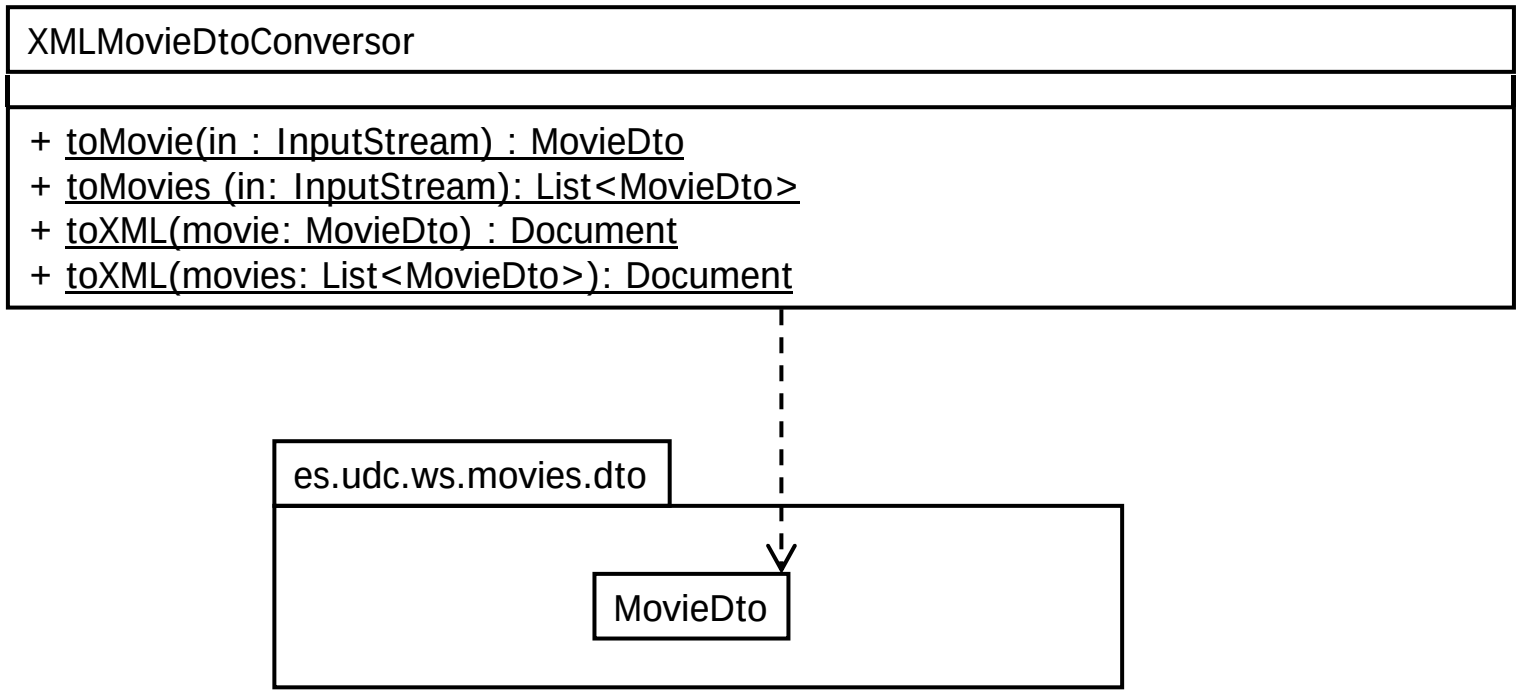
- Códigos de respuesta y de error: la ventaja de usar códigos estándar es que cualquier cliente o intermediario conoce la semántica de la respuesta sin conocer nuestros formatos específicos. Ejemplos:
  - Respuestas 400 o 410 son cacheables, 404 no (aunque a veces se considera), 500 no es cacheable
  - No tiene sentido reintentar una petición que ha devuelto 400 o 410, pero sí una que ha devuelto 500
- ... aunque a veces la diferencia puede no ser muy relevante en nuestro caso. E.g. 204 vs 200
- El cuerpo del mensaje puede llevar información adicional no especificable en HTTP para los clientes que sí conozcan nuestros formatos:
  - Ejemplo: cuánto hace que expiró la venta

es.udc.ws.movies.client.service.rest [Capa de Acceso al Servicio]

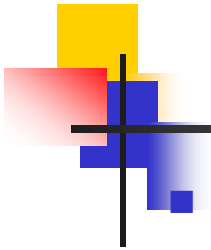


- **`RestClientMovieService`**
  - Utiliza Jakarta Commons HttpClient para realizar las peticiones HTTP
    - Framework Open Source de Apache (http://jakarta.apache.org/commons/httpclient)
    - Forma parte de Apache HttpComponents (http://hc.apache.org)
  - Utiliza `XmlMovieDtoConversor`, `XmlSaleDtoConversor` y `XmlExceptionConversor` para parsear / generar XML

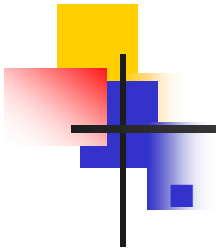
es.udc.ws.movies.xml (1)



es.udc.ws.movies.xml (2)



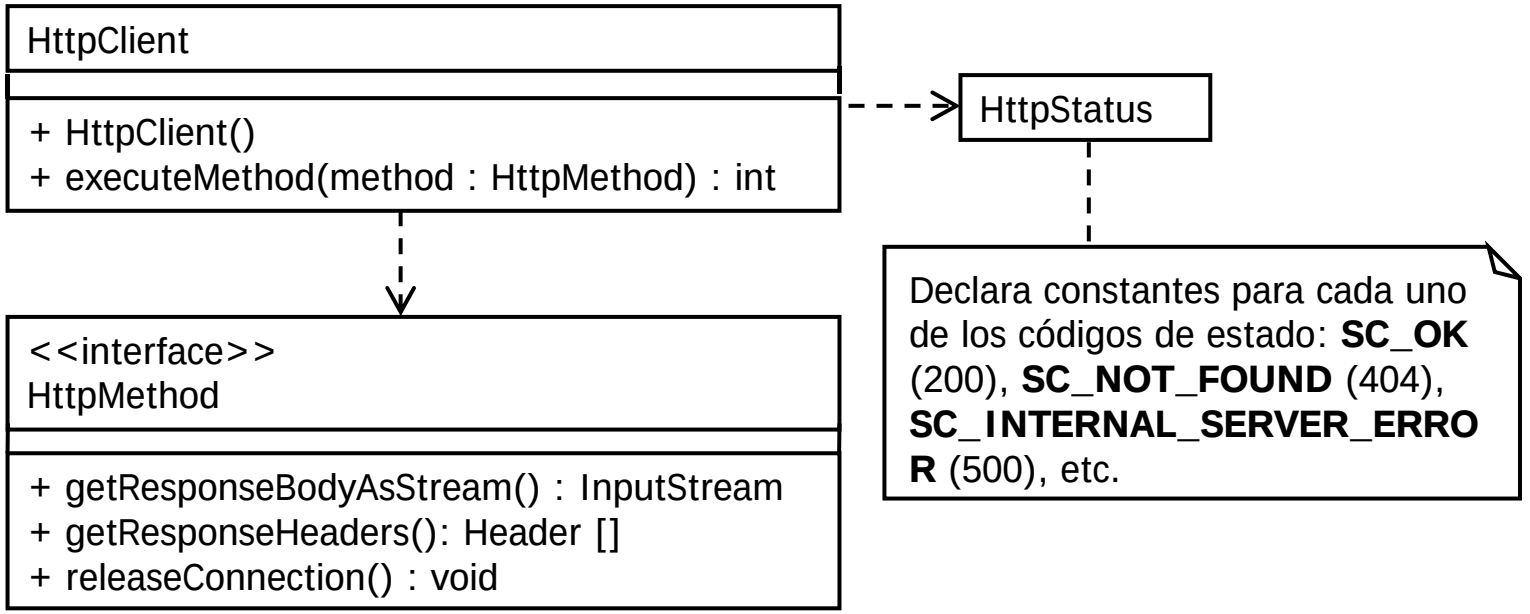
- Se usa en la capa de acceso y en la de implementación del servicio REST para convertir a/desde los objetos DTO desde/a su representación en XML
- **`XmlMovieDtoConversor`**
  - Realiza conversiones de **`MovieDto`** a/desde XML
  - Los métodos **`toMovie`** y el primer método **`toXML`** se estudiaron en el apartado 6.2
  - **`toMovies`**
    - Recibe un stream conteniendo una lista de películas en XML
    - Lo utiliza la capa de Acceso al Servicio para parsear las películas resultado de una consulta
  - Segundo método **`toXML`**
    - Operación inversa a **`toMovies`**
    - Lo utiliza la capa de Implementación del Servicio (para devolver los datos de las películas resultado de una consulta)



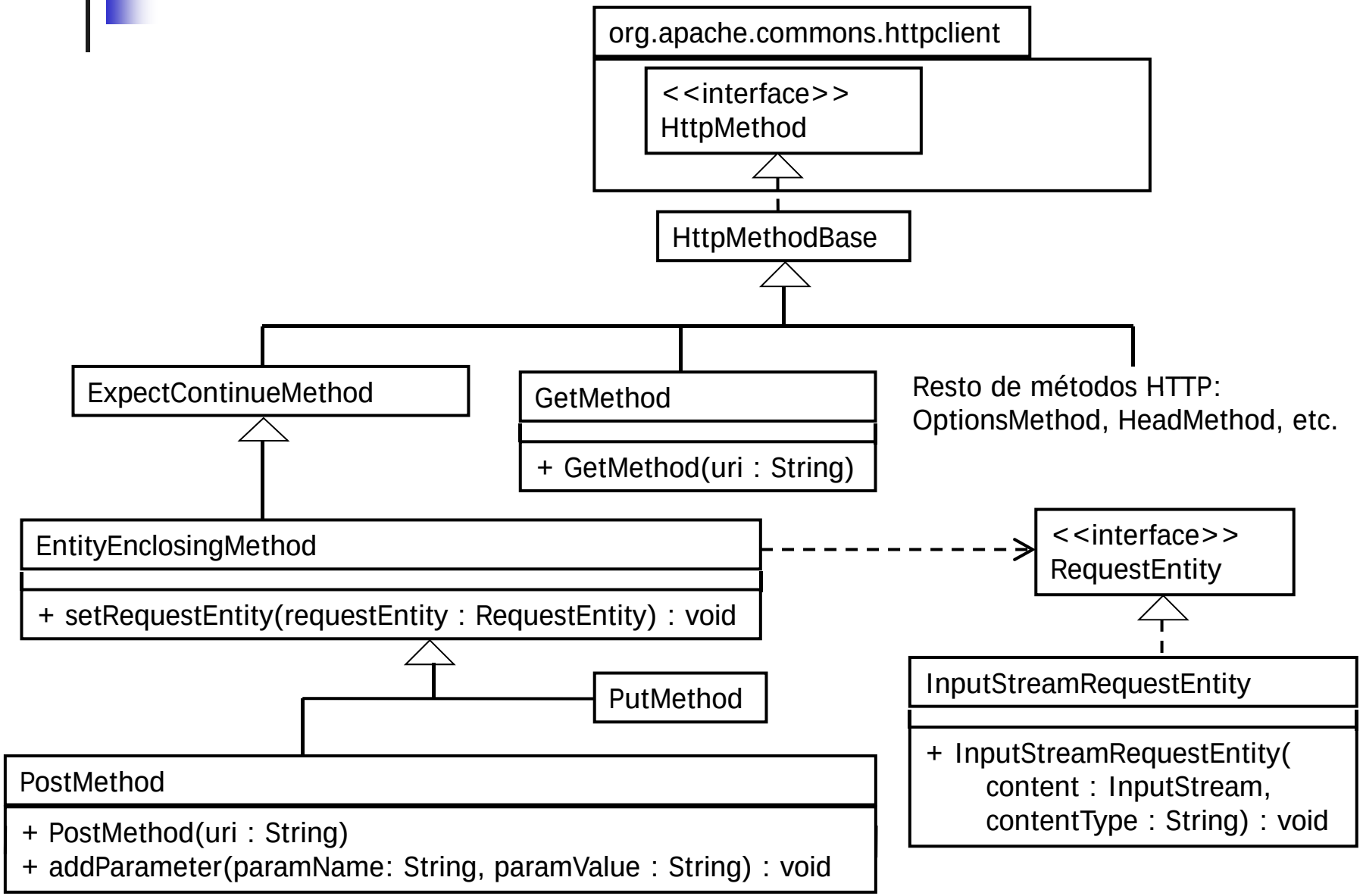
es.udc.ws.movies.xml (y 3)

- **`XmlSaleDtoConversor`**
  - Realiza conversiones de **`SaleDto`** a/desde XML
  - No se necesitan métodos que conviertan listas de ventas a/desde XML
- **`XmlExceptionConversor`**
  - Convierte a/desde las excepciones del modelo a su representación en XML

Visión global de HttpClient [org.apache.commons.httpclient] (1)

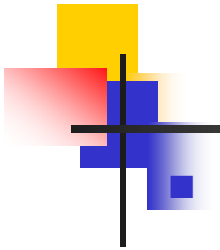


Visión global de HttpClient [org.apache.commons.httpclient.methods] (2)



Visión global de HttpClient (3)

- **`HttpMethod`**
  - Representa una petición HTTP y su correspondiente respuesta
  - Por cada tipo de operación HTTP (GET, POST, etc.) existe una clase **`XXXMethod`** (**`GetMethod`**, **`PostMethod`**, **`PutMethod`**, etc.) que implementa el interfaz **`HttpMethod`**
    - Veremos **`GetMethod`** y **`PostMethod`** . El resto son similares
  - Método **`InputStream getResponseBodyAsStream()`**
    - Permite leer el cuerpo de la respuesta de una operación HTTP
  - Método **`Headers[] getResponseHeaders()`**
    - Permite leer las cabeceras de respuesta de una operación HTTP
      - `String Header.getName ()` Nombre de la cabecera
      - `String Header.getValue ()` Valor de la cabecera
  - Método **`void releaseConnection()`**
    - Libera la conexión HTTP (HttpClient puede reusarla)



Visión global de HttpClient (4)

- **`HttpClient`**
  - **`int executeMethod(HttpMethod method)`**
    - Envía la petición HTTP especificada por parámetro
    - Devuelve un código de estado con el resultado de la petición

Visión global de HttpClient (5)

- **GetMethod**
  - Constructor **GetMethod(String uri)**
    - Permite construir un objeto para realizar una petición GET a la URI especificada por parámetro
    - Los parámetros (si los hay) forman parte de la URI
- Esquema típico para enviar una petición GET y leer su respuesta

```
GetMethod method = new
   GetMethod("http://example.org/resource?param1=val1&param2=val2");
try {
    /* 1: Enviar petición. */
    HttpClient client = new HttpClient();
    int statusCode = client.executeMethod(method);

    /* 2: Procesar respuesta. */
    if (statusCode == HttpStatus.SC_OK) {
        InputStream in = method.getResponseBodyAsStream();
        Headers [] headers = method.getResponseHeaders();
        << Tratar respuesta >>
    } else {
        << Tratar error >>
    }
} catch (<<...>>) {
   << Tratar excepciones >>
} finally {
    method.releaseConnection();
}
```

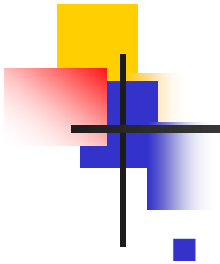
Visión global de HttpClient (6)

- **`GetMethod`** (cont)
  - Sólo se deben usar caracteres ASCII en la URL
  - Si en el **valor de un parámetro** es necesario usar un carácter especial (e.g. “?”, “&”, blanco, etc.) o un carácter no ASCII (e.g. “ñ”, “á”, etc.), se puede usar **`java.net.URLEncoder`**
    - Ejemplo

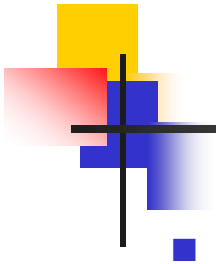
```
String url = ...
url += "parameter=" + URLEncoder.encode(value, "UTF-8");
```

  - En http://wiki.apache.org/HttpComponents/FrequentlyAskedApplicationDesignQuestions se documentan otras opciones posibles

Visión global de HttpClient (7)



- **`PostMethod`**
  - Constructor **`PostMethod(String uri)`**
    - Permite construir un objeto para realizar una petición POST a la URI especificada por parámetro
  - Método **`void addParameter(String paramName, String paramValue)`**
    - Permite añadir parámetros a la petición
    - Los parámetros no forman parte de la URI sino del cuerpo de la petición
  - Método **`void setRequestEntity(RequestEntity requestEntity)`**
    - Permite asociar un objeto **`RequestEntity`**
    - Un objeto **`RequestEntity`** permite enviar texto en el cuerpo de la petición HTTP



Visión global de HttpClient (8)

- **`InputStreamRequestEntity`**
  - Constructor **`InputStreamRequestEntity(InputStream content, String contentType)`**
    - Construye un **`RequestEntity`** que lee el texto desde un **`InputStream`** y cuyo tipo de contenido viene especificado en **`contentType`** (e.g. **`"text/xml; charset=UTF-8"`** si se envía XML en UTF-8)

Visión global de HttpClient (9)

- Esquema típico para enviar una petición POST con texto en el cuerpo y leer su respuesta

```
PostMethod method = new PostMethod("http://example.org/resource");
try {
    /* 1: Preparar petición. */
    InputStream xmlInputStream = ...
    InputStreamRequestEntity requestEntity =
        new InputStreamRequestEntity(xmlInputStream,
            "text/xml; charset=UTF-8");

    /* 2: Enviar petición. */
    HttpClient client = new HttpClient();
    method.setRequestEntity(requestEntity);
    int statusCode = client.executeMethod(method);

    /* 3: Procesar respuesta. */
    if (statusCode == HttpStatus.SC_OK) {
        InputStream in = method.getResponseBodyAsStream();
        Headers [] headers = method.getResponseHeaders();
        << Tratar respuesta >>
    } else {
        << Tratar error >>
    }
} catch (<<...>>) {
    << Tratar excepciones >>
} finally {
    method.releaseConnection();
}
```

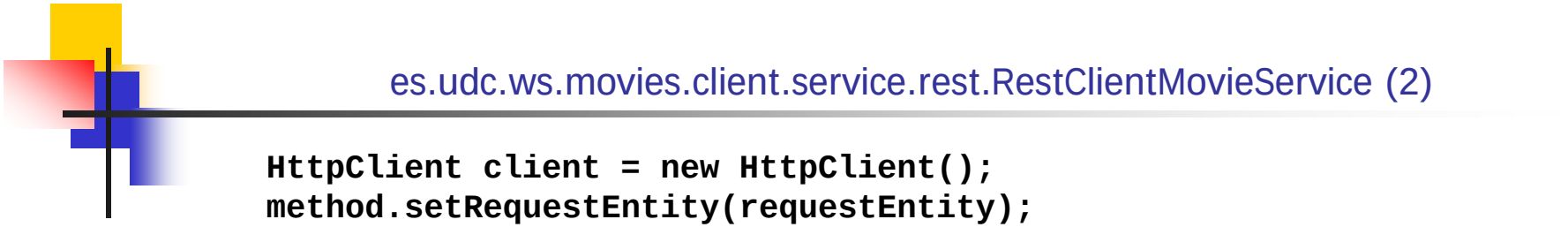
Visión global de HttpClient (y 10)

- Esquema típico para enviar una petición POST con parámetros y sin texto en el cuerpo, y leer su respuesta

```
PostMethod method = new PostMethod("http://example.org/resource");

method.addParameter("param1", "val1");
method.addParameter("param2", "val2");

try {

    /* 1: Enviar petición. */
    HttpClient client = new HttpClient();
    int statusCode = client.executeMethod(method);

    /* 2: Procesar respuesta. */
    if (statusCode == HttpStatus.SC_OK) {
        InputStream in = method.getResponseBodyAsStream();
        << Tratar respuesta >>
    }
} catch (<<...>>) {
    << Tratar excepciones. >>
} finally {
    method.releaseConnection();
}
```

es.udc.ws.movies.client.service.rest.RestClientMovieService (1)

```
@Override
public Long addMovie(MovieDto movie) throwsInputValidationException {

  PostMethod method = new PostMethod(getEndpointAddress() + "movies");

    try {
          ByteArrayOutputStream xmlOutputStream = new
   ByteArrayOutputStream();
           Document document;
           try {
               document = XmlMovieDtoConversor.toXml(movie);
               XMLOutputter outputter = new XMLOutputter(
                       Format.getPrettyFormat());
               outputter.output(document, xmlOutputStream);
           } catch (IOException ex) {
               throw new InputValidationException(ex.getMessage());
           }
     ByteArrayInputStream xmlInputStream =
            new ByteArrayInputStream(xmlOutputStream.toByteArray());

     InputStreamRequestEntity requestEntity =
                  new InputStreamRequestEntity(xmlInputStream,
                  response.getContentType());
```

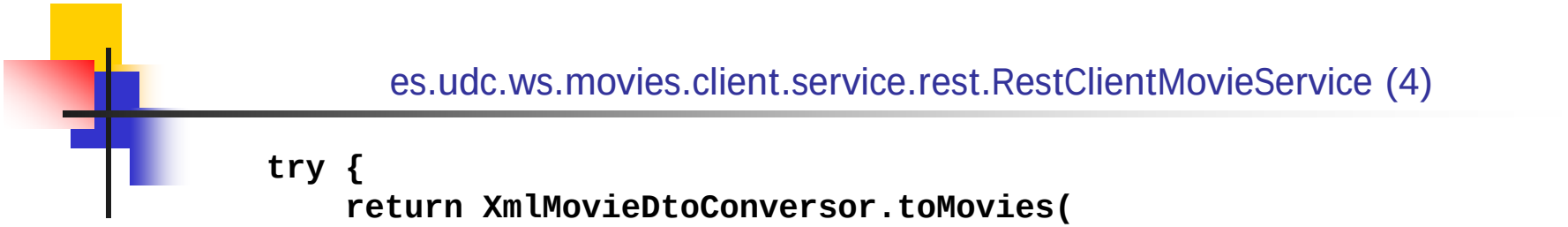


es.udc.ws.movies.client.service.rest.RestClientMovieService (2)

```
        HttpClient client = new HttpClient();
        method.setRequestEntity(requestEntity);

        int statusCode;
        try {
            statusCode = client.executeMethod(method);
        } catch (IOException ex) {
            throw new RuntimeException(ex);
        }

        try {
            validateResponse(statusCode, HttpStatus.SC_CREATED, method);
        } catch (InputValidationException ex) {
            throw ex;
        } catch (Exception ex) {
            throw new RuntimeException(ex);
        }

        return getIdFromHeaders(method);

    } finally {
        if (method != null) {
            method.releaseConnection();
        }
    }
}
```

es.udc.ws.movies.client.service.rest.RestClientMovieService (3)

```
public List<MovieDto> findMovies(String keywords) {
    GetMethod method = null;

    try {
        method = new GetMethod(getEndpointAddress() + "movies?keywords="
                        + URLEncoder.encode(keywords, "UTF-8"));
    } catch (UnsupportedEncodingException ex) {
        throw new RuntimeException(ex);
    }

    try {
        HttpClient client = new HttpClient();
        int statusCode;
        try {
            statusCode = client.executeMethod(method);
        } catch (Exception ex) {
            throw new RuntimeException(ex);
        }

        try {
            validateResponse(statusCode, HttpStatus.SC_OK, method);
        } catch (Exception ex) {
            throw new RuntimeException(ex);
       }
```



es.udc.ws.movies.client.service.rest.RestClientMovieService (4)

```
            try {
                return XmlMovieDtoConversor.toMovies(
                        method.getResponseBodyAsStream());
            } catch (ParsingException | IOException ex) {
                    throw new RuntimeException(ex);
            }

        } finally {
            if (method != null) {
                method.releaseConnection();
            }
        }
    }
```

es.udc.ws.movies.client.service.rest.RestClientMovieService (5)

```
public void removeMovie(Long movieId) throws InstanceNotFoundException{

    DeleteMethod method =
           new DeleteMethod(getEndpointAddress() + "movies/" + movieId);

    try {
        HttpClient client = new HttpClient();
        int statusCode = client.executeMethod(method);
        validateResponse(statusCode, HttpStatus.SC_NO_CONTENT, method);
    } catch (InstanceNotFoundException ex) {
        throw ex;
    } catch (Exception ex) {
        throw new RuntimeException(ex);
    } finally {
        if (method != null) {
            method.releaseConnection();
        }
    }
}
```

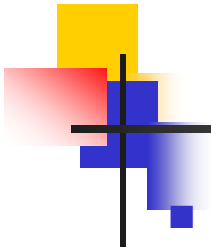
es.udc.ws.movies.client.service.rest.RestClientMovieService (6)

```
@Override
public Long buyMovie(Long movieId, String userId, String creditCardId)
      throws InstanceNotFoundException, InputValidationException {

  PostMethod method = new PostMethod(getEndpointAddress() + "sales");

  try {
      method.addParameter("movieId", Long.toString(movieId));
      method.addParameter("userId", userId);
      method.addParameter("creditCardId", creditCardId);
      HttpClient client = new HttpClient();
      int statusCode;
      try {
          statusCode = client.executeMethod(method);
      } catch (IOException ex) {
          throw new RuntimeException(ex);
      }
      try {
          validateResponse(statusCode, HttpStatus.SC_CREATED, method);
      }catch(InputValidationException | InstanceNotFoundException ex) {
          throw ex;
      } catch (Exception ex) {
          throw new RuntimeException(ex);
      }
      return getIdFromHeaders(method);
```

```
    } finally {
        if (method != null) {
            method.releaseConnection();
        }
    }
}

private static Long getIdFromHeaders(HttpMethod method) {
    String location = getResponseHeader(method, "Location");
    if (location != null) {
        int idx = location.lastIndexOf('/');
        return Long.valueOf(location.substring(idx + 1));
    }
    return null;
}

private static String getResponseHeader(HttpMethod method,
                String headerName) {
    Header[] headers = method.getResponseHeaders();
    for (int i = 0; i < headers.length; i++) {
        Header header = headers[i];
        if (headerName.equalsIgnoreCase(header.getName())) {
            return header.getValue();
        }
    }
    return null;
}
```

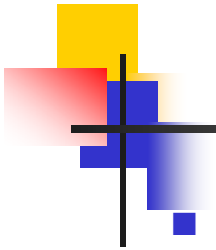
```
private void validateResponse(int statusCode,
                              int expectedStatusCode,
                              HttpMethod method)
            throws InstanceNotFoundException, SaleExpirationException,
                   InputValidationException,ParsingException{

  InputStream in;

  try {
      in = method.getResponseBodyAsStream();
  } catch (IOException ex) {
      throw new RuntimeException(ex);
  }

  String contentType = getResponseHeader(method, "Content-Type");
  boolean isXmlResponse = "application/xml".equalsIgnoreCase(contentType);
  if (!isXmlResponse && statusCode >= 400) {
      throw new RuntimeException("HTTP error; status code = "+statusCode);
  }
```

```
switch (statusCode) {
    case HttpStatus.SC_NOT_FOUND:
        try {
            throw XmlExceptionConversor.fromInstanceNotFoundExceptionXml(in);
        } catch (ParsingException e) {
            throw new RuntimeException(e);
       }
    case HttpStatus.SC_BAD_REQUEST:
        try {
            throw XmlExceptionConversor.fromInputValidationExceptionXml(in);
        } catch (ParsingException e) {
            throw new RuntimeException(e);
        }
    case HttpStatus.SC_GONE:
        try {
            throw XmlExceptionConversor.fromSaleExpirationExceptionXml(in);
        } catch (ParsingException e) {
            throw new RuntimeException(e);
        }
    default:
        if (statusCode != expectedStatusCode) {
            throw new RuntimeException("HTTP error; status code = "
                    + statusCode);
        }
        break;
  }
}
```



Comentarios

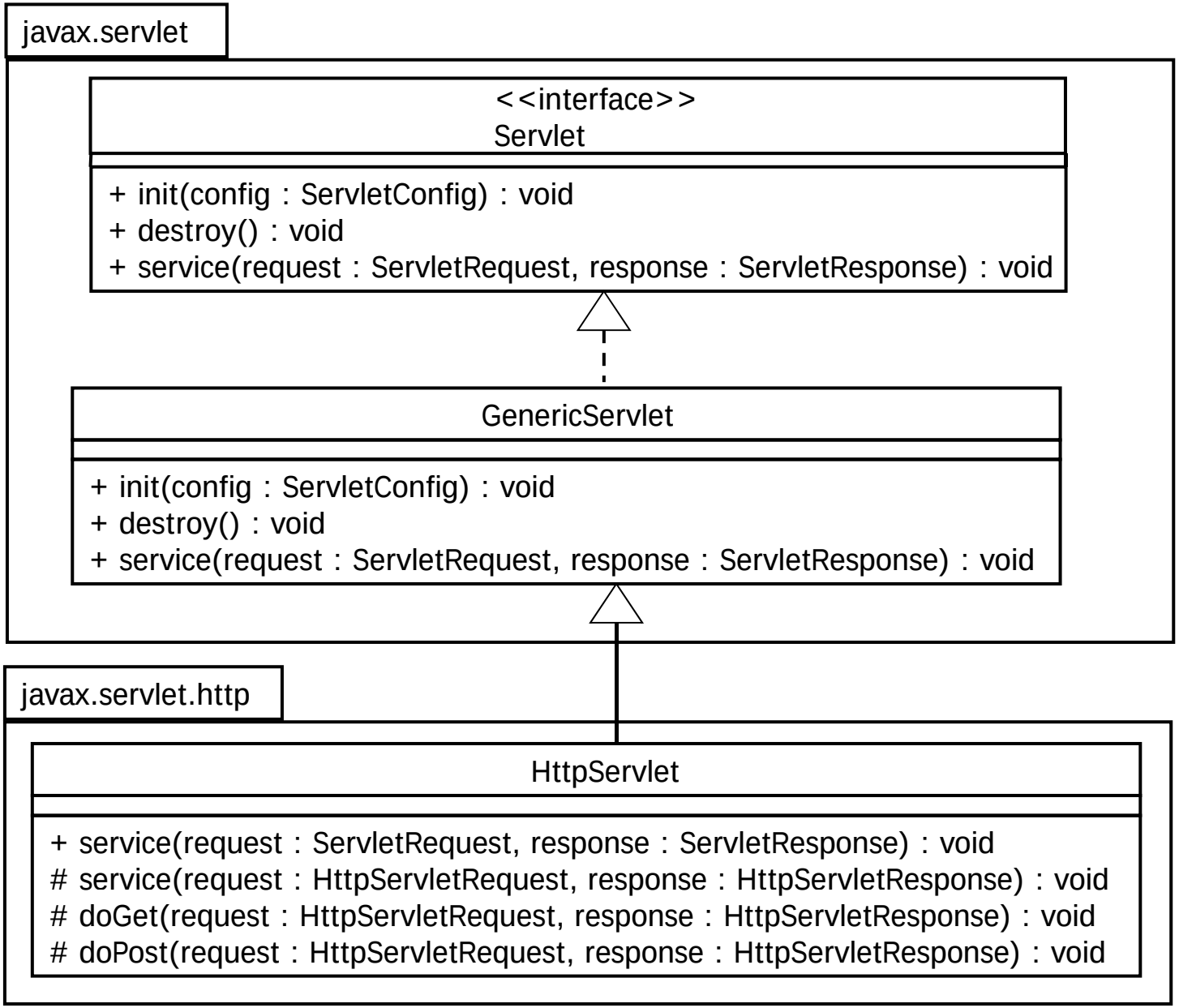
- No se muestran los métodos:
  - `updateMovie`. Similar a addMovie
  - `getMovieUrl`. Invoca por GET el recurso `/sales/{id}` y obtiene la URL de la respuesta
  - `getEndpointAddress ()`. Obtiene la parte fija de la URL desde el fichero de configuración usando la clase `ConfigurationParameters`
- `addMovie` tiene que “parsear” el identificador en la URL devuelta en la cabecera `Location` y el resto de métodos componer URLs en base a identificadores
  - Nuestra interfaz cliente asume identificadores numéricos
  - Un cliente que funciona en un entorno REST normalmente asumiría que los identificadores son URLs y haría esto innecesario



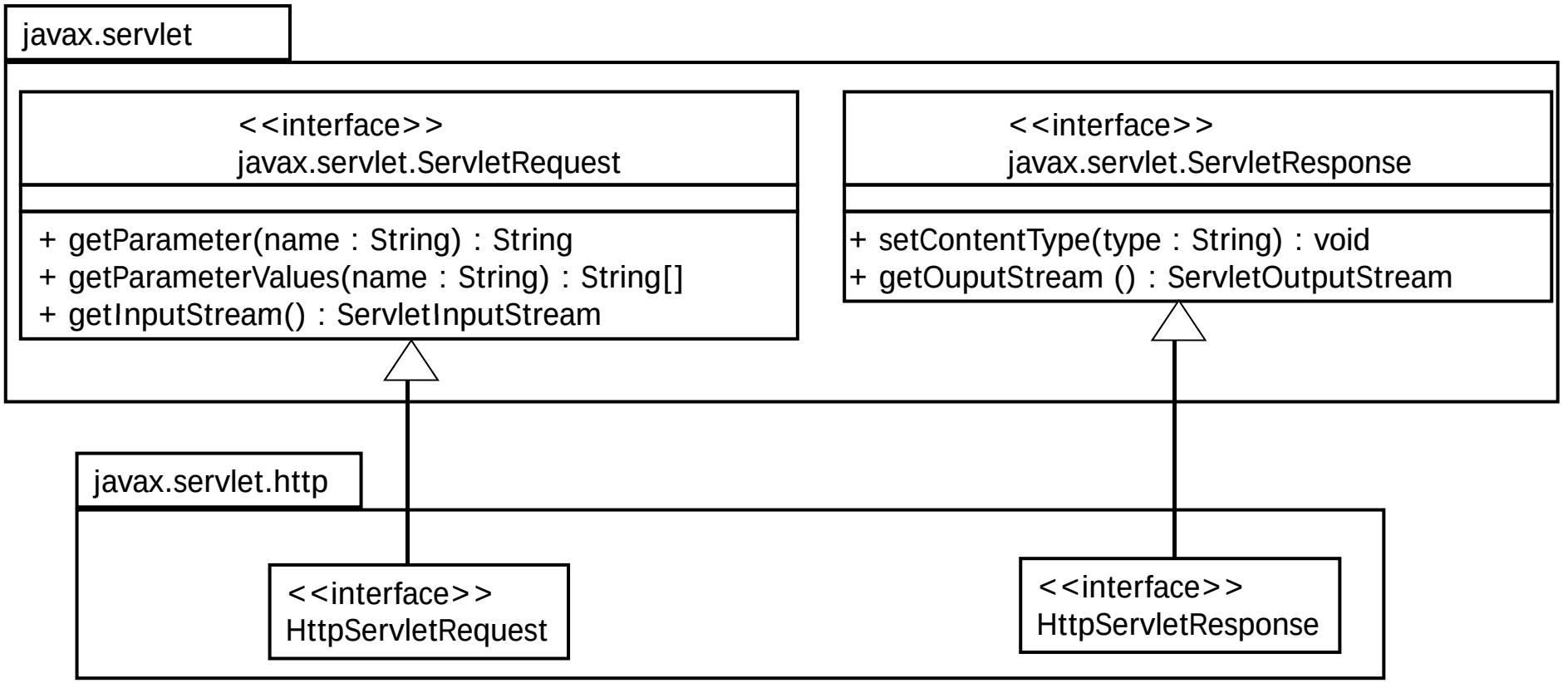
Visión global del framework de servlets (1)

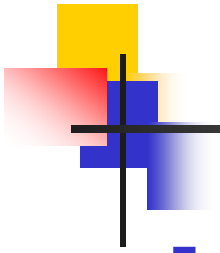
- Para la capa de implementación del servicio usaremos el API de servlets
- Un servlet es una clase que:
  - Se asocia a una o varias URLs
  - Cuando el servidor recibe una petición sobre esa URL, invoca al servlet asociado.
  - El servlet:
    - Accede al contenido de la petición
      - Ejemplo: valores de un form HTML de búsqueda
    - Ejecuta el código deseado para obtener la respuesta
      - Ejemplo web: búsqueda en BD por los parámetros del form
    - Codifica la respuesta en el formato deseado
      - Ejemplo web: página HTML con los resultados de la búsqueda
  - La respuesta devuelta por el servlet será la respuesta enviada por el servidor de aplicaciones al cliente

Visión global del framework de servlets (2)



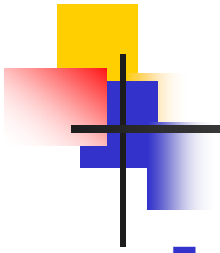
Visión global del framework de servlets (3)





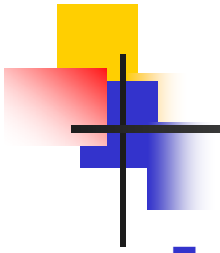
Visión global del framework de servlets (4)

- Normalmente el desarrollador implementa un servlet extendiendo de **`HttpServlet`** y redefine los métodos **`doXXX`** (e.g. **`doGet`**, **`doPost`**, **`doPut`**, **`doDelete`**, etc.) necesarios
- Ciclo de vida de un servlet
  - Cuando el servidor de aplicaciones Web necesita cargar un servlet en memoria (e.g. al arrancar, la primera vez que se accede a él, etc.), crea una instancia de la clase de implementación y llama a la operación **`init`** (interfaz **`Servlet`**)
    - **En una máquina virtual Java, sólo existe una instancia de cada servlet para cada aplicación Web instalada**
  - Cuando el servidor de aplicaciones Web decide eliminar un servlet de memoria (e.g. lleva cierto tiempo sin usarse), llama a la operación **`destroy`** (interfaz **`Servlet`**)



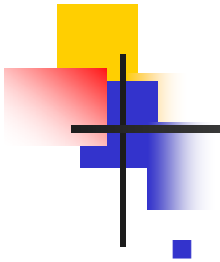
Visión global del framework de servlets (5)

- Ciclo de vida de un servlet (cont)
  - Cuando el servidor de aplicaciones Web recibe una petición dirigida a un servlet, invoca la operación **`service`** (interfaz **`Servlet`**)
  - La operación pública **`service`** de **`HttpServlet`** llama a la operación protegida **`service`**
    - Esta operación es una operación plantilla (Template Method), que llama a **`doGet`**, **`doPost`**, **`doPut`**, **`doDelete`**, etc., según la petición HTTP sea GET, POST, PUT, DELETE, etc.



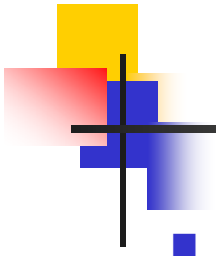
Visión global del framework de servlets (6)

- Modelo multi-thread de procesamiento de peticiones
  - **Cada vez que el servidor de aplicaciones Web recibe una petición dirigida a un servlet, la petición se procesa en un thread independiente (concurrentemente con otras peticiones)**
  - Internamente los servidores de aplicaciones suelen usar un “pool de threads”
    - Existe un conjunto (pool) de threads de tamaño configurable
    - Cada petición se inserta en una cola
    - La petición es servida por uno de los threads libres del pool
    - Si no hay ningún thread libre, la petición permanece en la cola hasta que uno termina su trabajo



Visión global del framework de servlets (7)

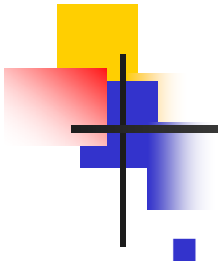
- Modelo multi-thread de procesamiento de peticiones (cont)
  - Dado que sólo existe una instancia de cada servlet para cada aplicación Web instalada, **los métodos `doXXX` tienen que ser thread-safe**
  - **No es necesario utilizar `synchronized` (¡¡¡ni eficiente!!!) si la implementación de los métodos `doXXX` sólo hace uso de variables locales o de variables globales (`static`) de sólo lectura (típicamente caches), que es lo normal**
  - Si los métodos **doXXX** modifican alguna estructura global (un atributo del servlet o alguna variable global), necesitan sincronizar su acceso
    - Sin embargo, en general, eso es mala idea, dado que una aplicación con estas características no funcionará en un entorno cluster
    - En un entorno cluster, el servidor de aplicaciones Web está replicado en varias máquinas para conseguir escalabilidad y tolerancia a fallos
    - En estos casos, es mejor usar una base de datos para las estructuras globales que sean de lectura/escritura



Visión global del framework de servlets (8)

- **`ServletRequest`**
  - **`String getParameter(String name)`**

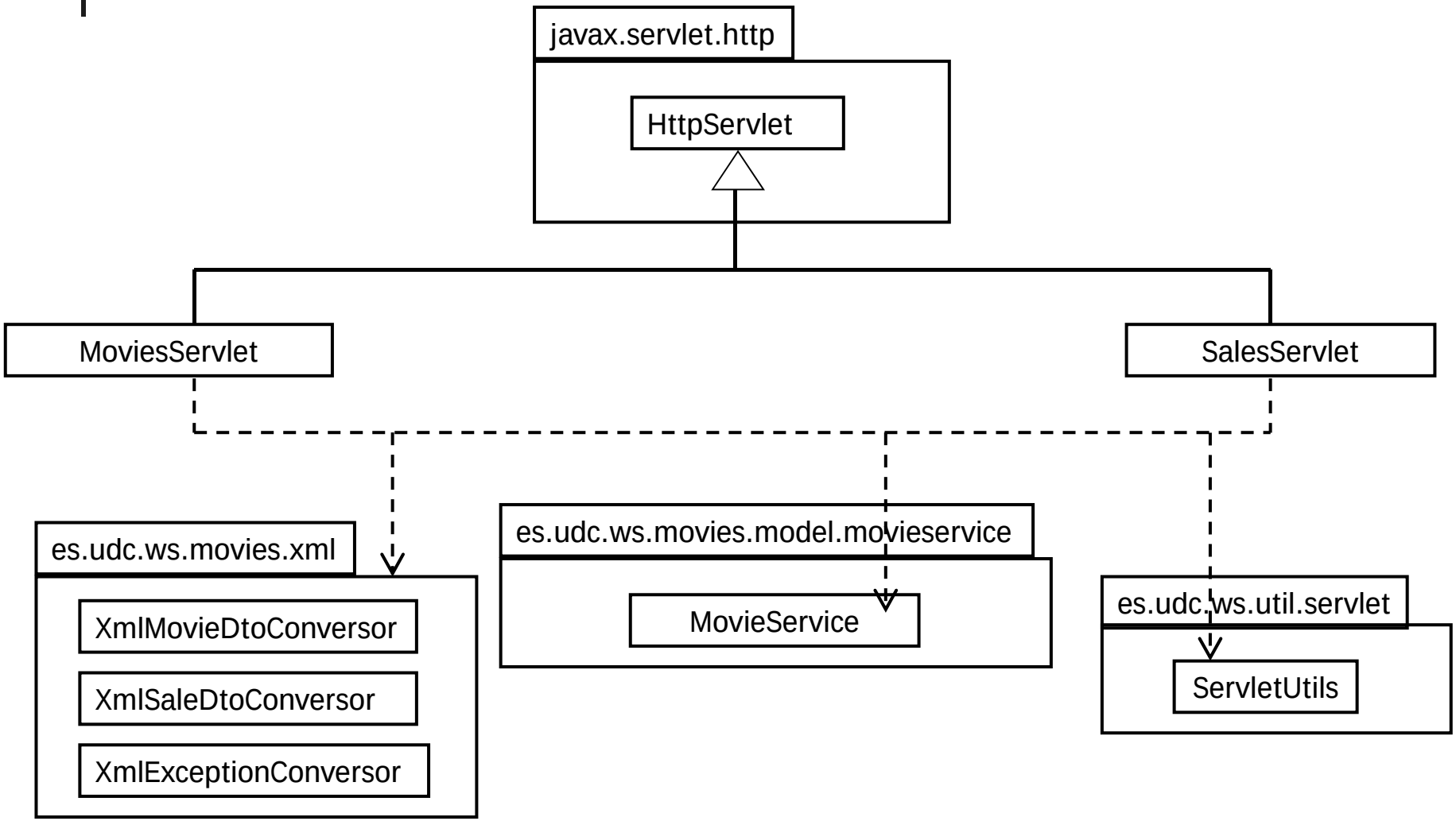
    Permite obtener el valor de un parámetro univaluado
  - **`String[] getParameterValues(String name)`**
    - Permite obtener el valor de un parámetro multivaluado
    - También se puede usar con parámetros univaluados
  - NOTA
    - Un parámetro multivaluado es aquel que tiene (o puede tener) varios valores
    - Al realizar la petición HTTP el parámetro (en la URI o en el cuerpo del mensaje, dependiendo del tipo de petición) se especifica “n” veces, cada una con su valor
      - Ejemplo: http://example.com/weather?city=Coruna&city=Santiago
  - **`ServletInputStream getInputStream()`**
    - Permite leer el cuerpo de la petición
    - **`ServletInputStream`** es una clase abstracta que implementa **`java.io.InputStream`**



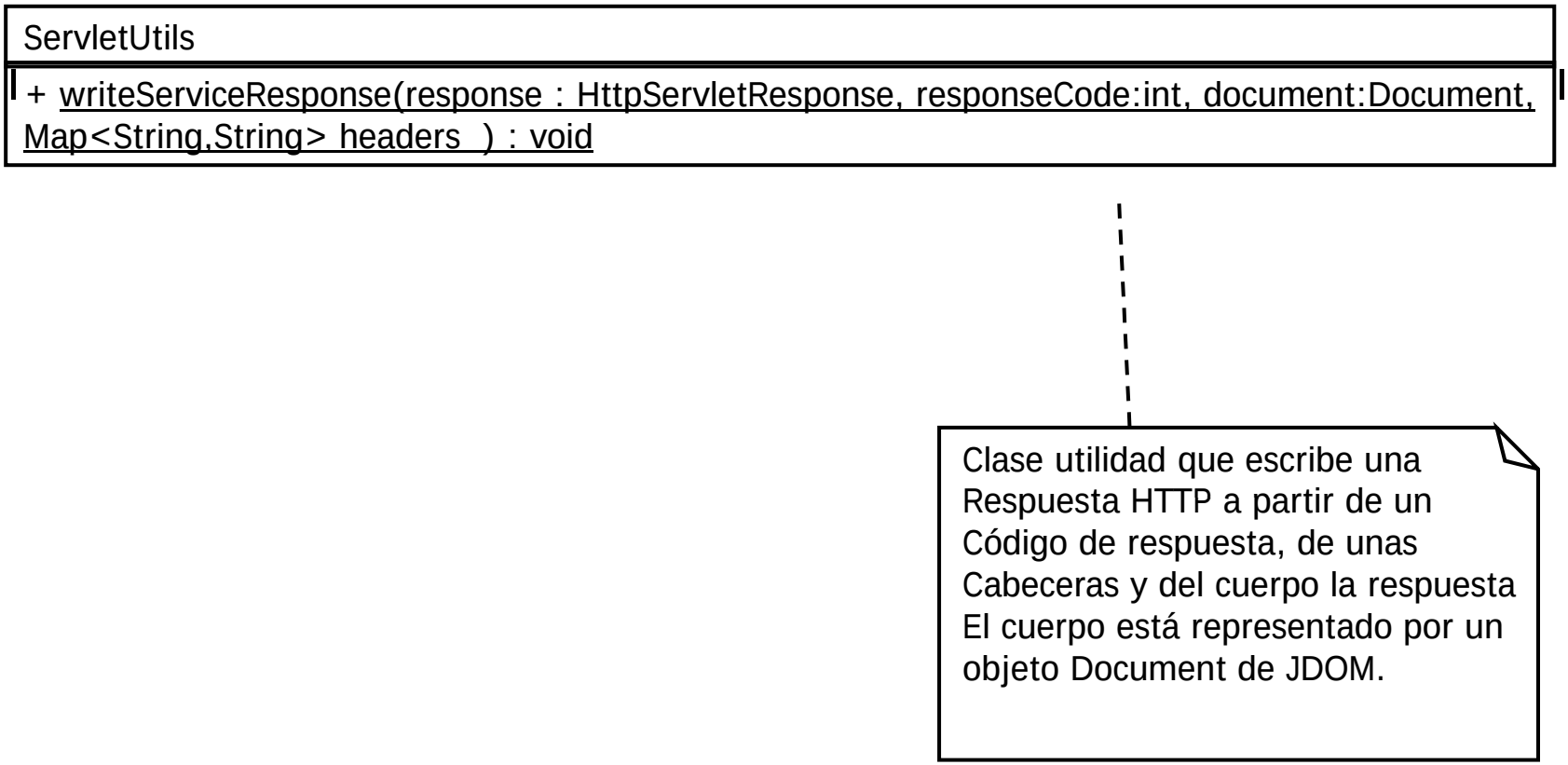
Visión global del framework de servlets (y 9)

- **`ServletResponse`**
  - **`void setContentType(String type)`**
    - Especifica el tipo de contenido de la respuesta (e.g. **`"text/xml; charset=UTF-8"`**)
  - **`ServletOutputStream getOutputStream()`**
    - Permite escribir la respuesta
    - **`ServletOutputStream`** es una clase abstracta que implementa **`java.io.OutputStream`**

es.udc.ws.movies.restservice.servlets [Capa Implementación del Servicio] (1)

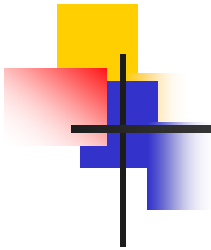


es.udc.ws.util.servlet
[Capa Implementación del Servicio] (y 2)



es.udc.ws.movies.restservice.servlets.MoviesServlet (1)

```
public class MoviesServlet extends HttpServlet {

  @Override
  protected void doGet(HttpServletRequest req, HttpServletResponse
    resp) throws ServletException, IOException {

    String path = req.getPathInfo();

    if(path == null || path.length() == 0 || "/".equals(path)) {

        String keyWords = req.getParameter("keywords");

        List<Movie> movies = MovieServiceFactory.getService()
                .findMovies(keyWords);

        List<MovieDto> movieDtos = MovieToMovieDtoConversor
                .toMovieDtos(movies);

        ServletUtils.writeServiceResponse(resp,
                 HttpServletResponse.SC_OK,
                  XmlMovieDtoConversor.toXml(movieDtos), null);
    }
```

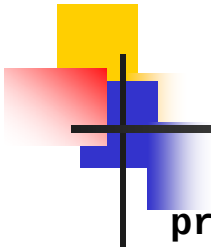


es.udc.ws.movies.restservice.servlets.MoviesServlet (2)

```
else {
    String movieIdAsString = path.endsWith("/") && path.length() > 2?
    path.substring(1, path.length() - 1) : path.substring(1);

    Long movieId;

    try {
        movieId = Long.valueOf(movieIdAsString);

    } catch (NumberFormatException ex) {

        ServletUtils.writeServiceResponse(resp,
          HttpServletResponse.SC_BAD_REQUEST,
          XmlExceptionConversor.toInputValidationExceptionXml(
           new InputValidationException("Invalid Request: " +
          "parameter 'movieId' is invalid '" +movieIdAsString +"'")),
           null);

        return;
    }
```

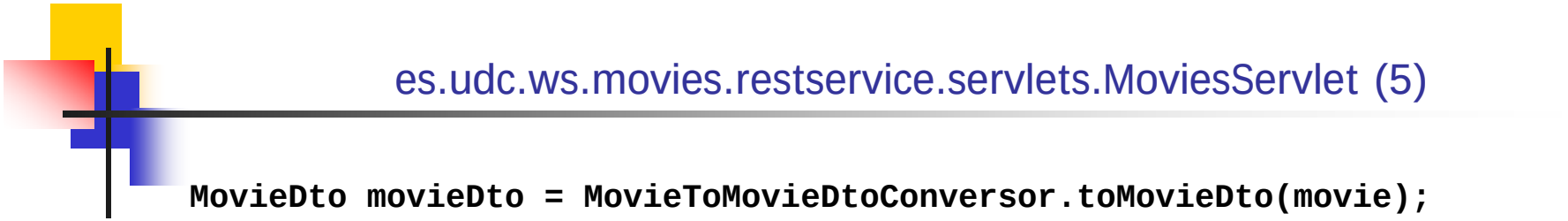
es.udc.ws.movies.restservice.servlets.MoviesServlet (3)

```
        Movie movie;

        try {
             movie = MovieServiceFactory.getService().findMovie(movieId);
        } catch (InstanceNotFoundException ex) {

             ServletUtils.writeServiceResponse(resp,
                HttpServletResponse.SC_NOT_FOUND,
                XmlExceptionConversor.toInstanceNotFoundException(
                 new InstanceNotFoundException(
                    ex.getInstanceId().toString(), ex.getInstanceType())),
                 null);
             return;
        }

        MovieDto movieDto = MovieToMovieDtoConversor.toMovieDto(movie);

        ServletUtils.writeServiceResponse(resp,
                     HttpServletResponse.SC_OK,
                     XmlMovieDtoConversor.toXml(movieDto), null);
        }
    }
```



es.udc.ws.movies.restservice.servlets.MoviesServlet (4)

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {

  MovieDto xmlmovie;

  try {
      xmlmovie = XmlMovieDtoConversor.toMovie(req.getInputStream());
  } catch (ParsingException ex) {
      ServletUtils.writeServiceResponse(resp,
             HttpServletResponse.SC_BAD_REQUEST,
             XmlExceptionConversor.toInputValidationExceptionXml(
                  new InputValidationException(ex.getMessage())),null);
      return;
  }

  Movie movie = MovieToMovieDtoConversor.toMovie(xmlmovie);
  try {
      movie = MovieServiceFactory.getService().addMovie(movie);
  } catch (InputValidationException ex){
      ServletUtils.writeServiceResponse(resp,
             HttpServletResponse.SC_BAD_REQUEST,
             XmlExceptionConversor.toInputValidationExceptionXml(
              new InputValidationException(ex.getMessage())),null);
      return;
  }
```

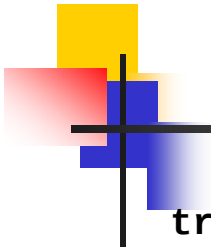


es.udc.ws.movies.restservice.servlets.MoviesServlet (5)

```
        MovieDto movieDto = MovieToMovieDtoConversor.toMovieDto(movie);

        String movieURL = req.getRequestURL().append("/").append(
                        movie.getMovieId()).toString();

        Map<String, String> headers = new HashMap<>(1);
        headers.put("Location", movieURL);

        ServletUtils.writeServiceResponse(resp,
                        HttpServletResponse.SC_CREATED,
                        XmlMovieDtoConversor.toXml(movieDto), headers);
    }
```

es.udc.ws.movies.restservice.servlets.MoviesServlet (6)

```
@Override
protected void doDelete(HttpServletRequest req, HttpServletResponse
   resp) throws ServletException, IOException {

  String requestURI = req.getRequestURI();
  int idx = requestURI.lastIndexOf('/');
  if (idx <= 0) {
      ServletUtils.writeServiceResponse(resp,
             HttpServletResponse.SC_BAD_REQUEST,
             XmlExceptionConversor.toInputValidationExceptionXml(
              new InputValidationException("Invalid Request: " +
                  "unable to find movie id")), null);
      return;
  }
  Long movieId;
  String movieIdAsString = requestURI.substring(idx + 1);
  try {
      movieId = Long.valueOf(movieIdAsString);
  } catch (NumberFormatException ex) {
      ServletUtils.writeServiceResponse(resp,
        HttpServletResponse.SC_BAD_REQUEST,
        XmlExceptionConversor.toInputValidationExceptionXml(
         new InputValidationException("Invalid Request: " +
          "unable to parse movie id '" + movieIdAsString + "'")),null)
      return;
  }
```

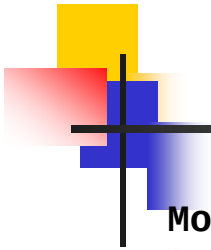


es.udc.ws.movies.restservice.servlets.MoviesServlet (7)

```
try {
      MovieServiceFactory.getService().removeMovie(movieId);
  } catch (InstanceNotFoundException ex) {
     ServletUtils.writeServiceResponse(resp,
             HttpServletResponse.SC_NOT_FOUND,
             XmlExceptionConversor.toInstanceNotFoundException(
              new InstanceNotFoundException(
                     ex.getInstanceId().toString(),
               ex.getInstanceType())), null);
      return;
  }
  ServletUtils.writeServiceResponse(resp,
         HttpServletResponse.SC_NO_CONTENT, null, null);
}
```

es.udc.ws.movies.restservice.servlets.MoviesServlet (8)

```
@Override
protected void doPut(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

    String requestURI = req.getRequestURI();
    int idx = requestURI.lastIndexOf('/');
    if (idx <= 0) {
        ServletUtils.writeServiceResponse(resp,
                HttpServletResponse.SC_BAD_REQUEST,
                XmlExceptionConversor.toInputValidationExceptionXml(
                 new InputValidationException("Invalid Request: " +
                     "unable to find movie id")), null);
        return;
    }
    Long movieId;
    String movieIdAsString = requestURI.substring(idx + 1);
    try {
        movieId = Long.valueOf(movieIdAsString);
    } catch (NumberFormatException ex) {
        ServletUtils.writeServiceResponse(resp,
            HttpServletResponse.SC_BAD_REQUEST,
            XmlExceptionConversor.toInputValidationExceptionXml(
              new InputValidationException("Invalid Request: " +
            "unable to parse movie id '" + movieIdAsString + "'")),null);
        return;
    }
```



es.udc.ws.movies.restservice.servlets.MoviesServlet (9)

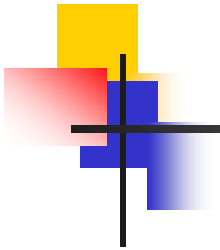
```
MovieDto movieDto;
try {
      movieDto = XmlMovieDtoConversor.toMovie(req.getInputStream());
  } catch (ParsingException ex) {
      ServletUtils.writeServiceResponse(resp,
          HttpServletResponse.SC_BAD_REQUEST,
          XmlExceptionConversor.toInputValidationExceptionXml(
           new InputValidationException(ex.getMessage())), null);
      return;
  }
  Movie movie = MovieToMovieDtoConversor.toMovie(movieDto);
  movie.setMovieId(movieId);
  try {
      MovieServiceFactory.getService().updateMovie(movie);
  } catch (InputValidationException ex) {
      ServletUtils.writeServiceResponse(resp,
          HttpServletResponse.SC_BAD_REQUEST,
          XmlExceptionConversor.toInputValidationExceptionXml(
           new InputValidationException(ex.getMessage())), null);
      return;
```

es.udc.ws.movies.restservice.servlets.MoviesServlet (y 10)

```
    } catch (InstanceNotFoundException ex) {
        ServletUtils.writeServiceResponse(resp,
           HttpServletResponse.SC_NOT_FOUND,
           XmlExceptionConversor.toInstanceNotFoundException(
            new InstanceNotFoundException(
             ex.getInstanceId().toString(), ex.getInstanceType())),null);
        return;
    }

ServletUtils.writeServiceResponse(resp,
            HttpServletResponse.SC_NO_CONTENT, null, null);
}
```

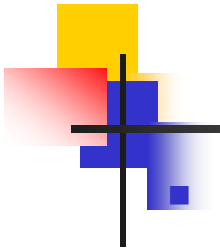
es.udc.ws.movies.service.rest.servlets.ServletUtils (1)

```
public class ServletUtils {

  public static void writeServiceResponse(HttpServletResponse response,
    int responseCode,Document edocument,
    Map<String,String> headers)throws IOException {

      writeResponse(response, responseCode,
              "application/xml", headers);

      if(document != null) {
      XMLOutputter outputter = new XMLOutputter(
                          Format.getPrettyFormat());

      outputter.output(document, response.getOutputStream());
       }
}
```



es.udc.ws.movies.service.rest.servlets.ServletUtils (y 2)

```
private static void writeResponse(HttpServletResponse response,
 int responseCode, String contentType, Map<String, String> headers) {

    if (headers != null && !headers.isEmpty()) {
        for (Map.Entry<String, String> entry : headers.entrySet()) {
            String key = entry.getKey();
            String value = entry.getValue();
            response.setHeader(key, value);
        }
    }
    response.setStatus(responseCode);
    if (contentType != null) {
        response.setContentType(contentType);
    }
}
```

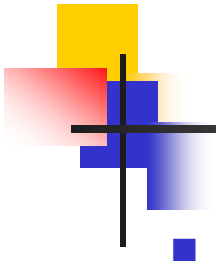


Comentarios

- `SalesServlet`
  - `doPost`. Similar al de MoviesServlet. Invoca a la operación `buyMovie` del modelo.
  - `doGet`. Similar al de MoviesServlet pero obliga a proporcionar un id (e.g. `sales/2`)
- Los servlets no tratan `RuntimeException`
  - Representa errores relativos a la infraestructura usada
  - El servidor de aplicaciones captura las excepciones de runtime, y si se producen, devuelve una respuesta con código de estado 500 (INTERNAL SERVER ERROR), que es lo que deseamos
- `ServletUtils`
  - Clase utilidad para escribir la respuesta a una petición HTTP
  - El cuerpo de la respuesta es un objeto `Document` de JDOM.

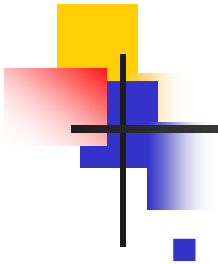
Cambios en web.xml

```
<!-- REST service -->
<servlet>
    <display-name>MoviesServlet</display-name>
    <servlet-name>MoviesServlet</servlet-name>
    <servlet-class>
        es.udc.ws.movies.restservice.servlets.MoviesServlet
    </servlet-class>
</servlet>

<servlet>
    <display-name>SalesServlet</display-name>
    <servlet-name>SalesServlet</servlet-name>
    <servlet-class>
        es.udc.ws.movies.restservice.servlets.SalesServlet
    </servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>MoviesServlet</servlet-name>
    <url-pattern>/movies/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
    <servlet-name>SalesServlet</servlet-name>
    <url-pattern>/sales/*</url-pattern>
</servlet-mapping>
```



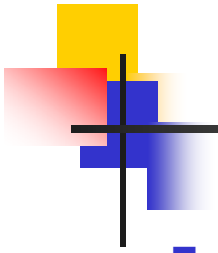
Comentarios (1)

- **`display-name`**
  - Nombre visual que mostrará la aplicación de administración del servidor de aplicaciones para esta aplicación Web
- **`servlet`**
  - Permite definir un servlet, especificando un nombre visual (**`display-name`**), un nombre (**`servlet-name`**) para referirse al servlet desde el resto del fichero **`web.xml`** y el nombre completo de la clase que lo implementa (**`servlet-class`**)



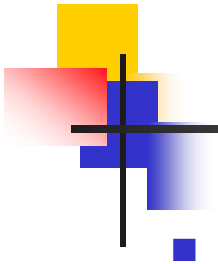
Comentarios (y 2)

- **`servlet-mapping`**
  - Permite especificar las URLs a las que responderá el servlet especificado en **`servlet-name`**
  - **`url-pattern`** permite especificar una URL concreta o un patrón, como en el ejemplo (e.g. **`/movies/*`**)
  - NOTA: las URLs especificadas no incluyen la parte inicial (**`http://direcciónServidor[:puerto]/nombreAplicaciónWeb`**)
    - Si el servicio cambia de dirección y/o puerto, o la aplicación Web se reinstala con otro nombre, no hay que cambiar el fichero **`web.xml`**



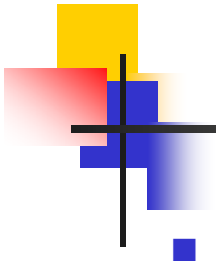
Validación de documentos XML (1)

- Tanto los clientes como el servicio comprueban que el XML está bien formado
  - Lo hace JDOM automáticamente
- Ni los clientes ni el servicio comprueban con JDOM que el documento es válido
  - **Sin embargo, se hacen las comprobaciones necesarias**
  - Ejemplo 1: **`XmlMovieDtoConversor`** comprueba que los elementos obligatorios aparecen o lanza una `ParsingException`
  - Ejemplo 2: La capa modelo comprueba que los parámetros y datos recibidos son válidos (e.g. números de tarjeta de crédito).



Validación de documentos XML (2)

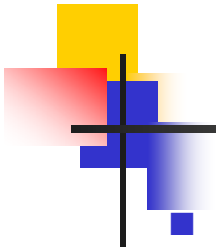
- Ventajas de no realizar una validación estricta
  - Eficiencia
    - Especialmente importante para la implementación de servicios (que potencialmente pueden recibir muchas peticiones concurrentes)
  - Evolución en el tiempo
    - Es posible extender el XML del protocolo sin que dejen de funcionar clientes y/o servicios



Validación de documentos XML (3)

- Ejemplo 1 (evolución en el tiempo)
  - Muchas empresas han construido clientes de búsqueda de películas
  - Más adelante la proveedora del servicio decide añadir el tag **`rating`** (puntuación) a la información de una película
  - Modifica la implementación del servicio y el esquema XML. En este momento, los clientes no están actualizados
  - Sin embargo, no dejan de funcionar, dado que en el XML que reciben sólo preguntan por los tags que conocen (e.g. **`movieId`**, **`title`**, **`runtime`**, etc.)
  - Aún no usando RPC, si los clientes tuviesen una copia local del viejo esquema e intentasen validar contra ella, fallaría la validación hasta que se actualizase el esquema
    - Podría descargarse el esquema cada vez, pero es ineficiente

Validación de documentos XML (y 4)



- Ejemplo 2 (evolución en el tiempo)
  - Los clientes que introdujesen datos de nuevas películas sin el nuevo tag también podrían seguir funcionando
  - No enviarían la puntuación de la película
  - El servicio tendría que tratar el tag **`rating`** como opcional (podría validarse siempre que el esquema especificase este elemento como opcional)
- Ejemplo 3 (evolución en el tiempo)
  - Si hubiese más de una instancia del servicio, y alguna de ellas no estuviese actualizada, los clientes actualizados que trabajasen contra servicios desactualizados tampoco dejarían de funcionar
  - Los servicios desactualizados descartarían el tag **`rating`** (si validasen contra el viejo esquema, fallaría)
  - Los clientes actualizados tendrían que tratar el tag **`rating`** como opcional (podrían validar, siempre que el esquema especificase este atributo como opcional)
- Las implementaciones de RPC (e.g. SOAP), normalmente fallan en estos casos hasta que se regeneran los stubs.