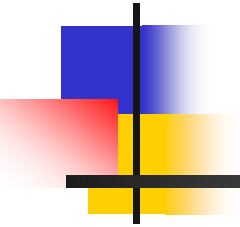




Tema 6: Diseño e Implementación de Servicios Web REST

- 6.1. Descripción del Caso de Estudio
- 6.2. Introducción a los Servicios Web REST
- 6.3. Caso de Estudio: Diseño e Implementación de un Servicio Web REST

Tema 6.1: Descripción del Caso de Estudio

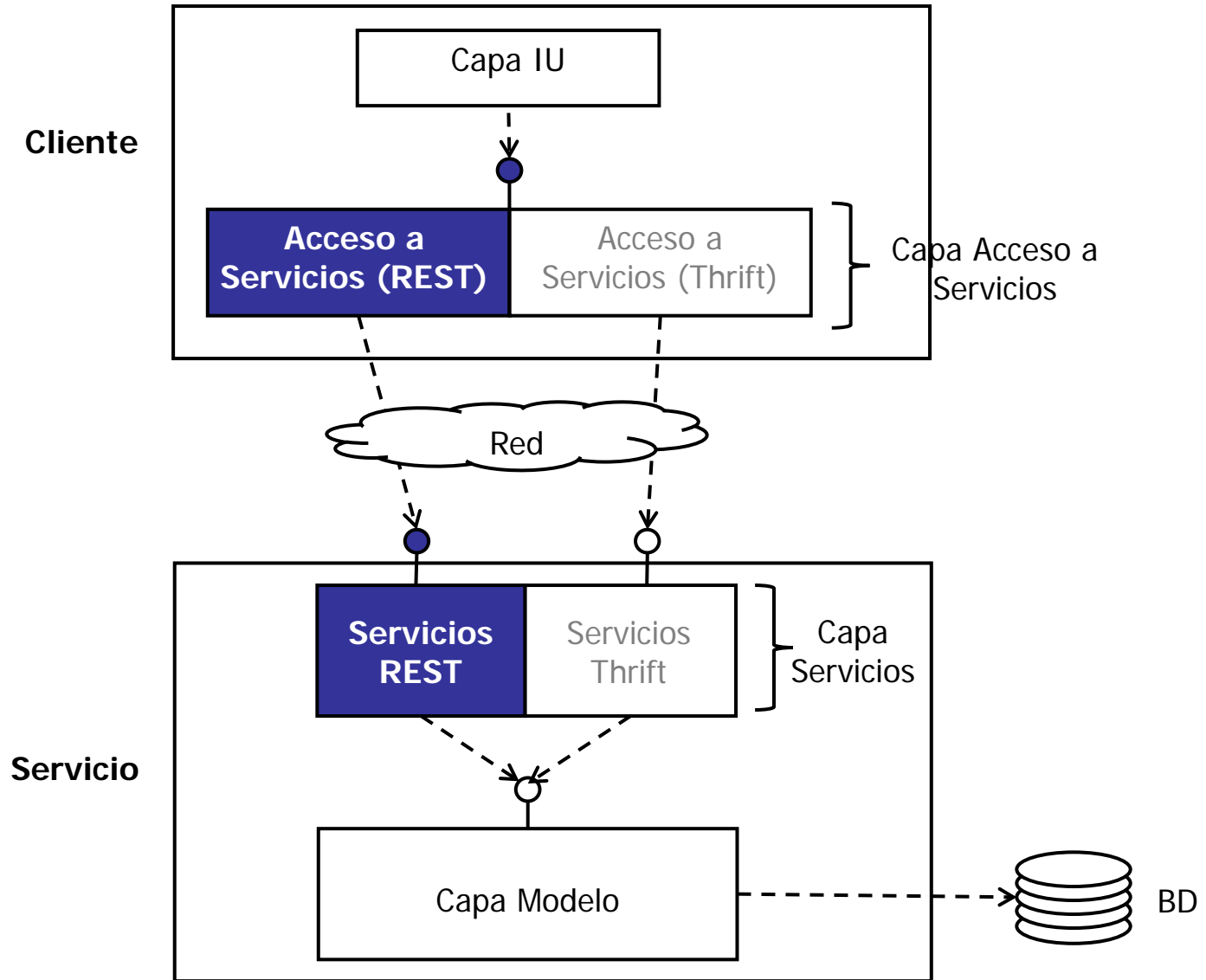




Descripción del caso de estudio

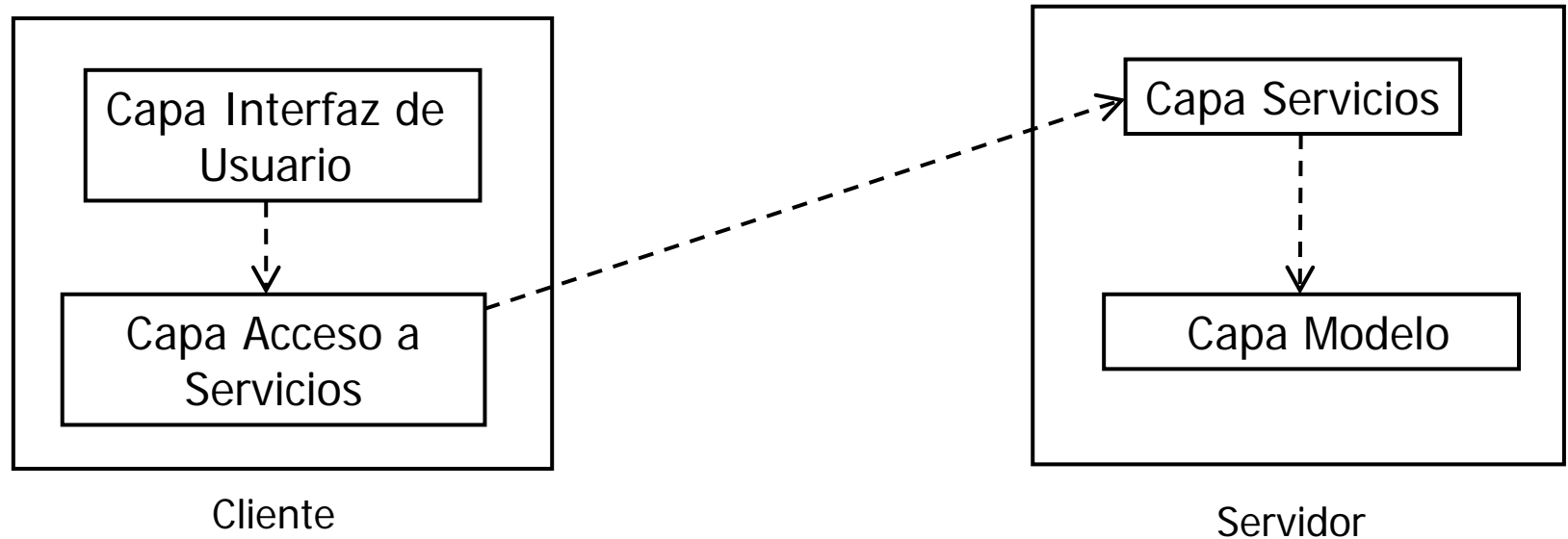
- Retomamos el ejemplo ilustrado en los temas 3 y 5
- Expondremos la capa modelo desarrollada como un servicio web usando REST y como un servicio Apache Thrift
- En nuestro ejemplo, el cliente será una sencilla aplicación de línea de comandos
- El cliente puede configurarse (sin necesidad de recompilación) para usar el servicio web REST o Thrift

Arquitectura general de Movies



Diseño por Capas (1)

- En el caso de estudio usaremos el diseño por capas para ocultar las tecnologías de acceso e implementación de los servicios





Diseño por capas (2)

- Capa **Interfaz de Usuario (o lógica de la aplicación cliente)**
 - Implementa la interfaz de usuario (y/o la lógica de la aplicación cliente)
 - No depende de la tecnología de acceso al servicio
- Capa **Acceso a Servicios**
 - Implementa el acceso a los servicios
 - Ofrece una API que oculta la tecnología usada para acceder a los servicios
- Capa **Servicios**
 - Utiliza una tecnología para implementar los servicios
 - Delega en la capa Modelo
- Capa **Modelo**
 - Implementa la lógica de la aplicación
 - No depende de la tecnología de implementación de los servicios



Diseño por capas (y 3)

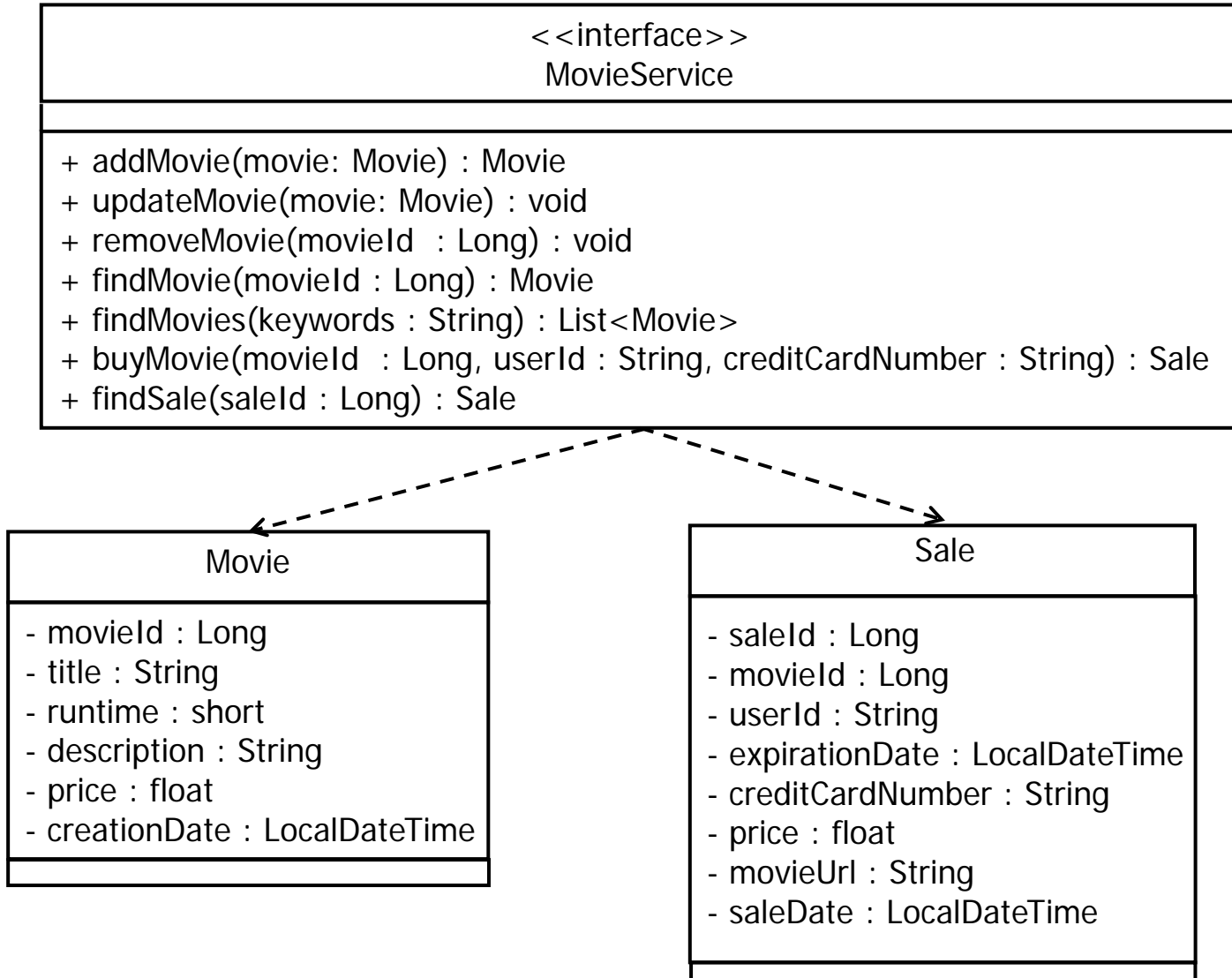
El uso de diseño por capas en este caso tiene las ventajas y riesgos habituales

- La persona (o el grupo de personas) que se encarga del desarrollo de las capas IU y Modelo no tienen porque saber nada sobre las tecnologías de los servicios
- Cada capa puede ser desarrollada en paralelo
- Se facilita el mantenimiento del software
 - Cambios en la implementación de una capa (e.g. uso de nuevas tecnologías) no conllevan cambios en el resto de capas

■ Riesgos

- El software es más complejo
 - Si el software a construir es muy sencillo (e.g. un prototipo), puede no valer la pena
- A veces, el que el modelo conozca los detalles de la tecnología de implementación del servicio podría permitir ciertas optimizaciones

Interfaz de la Capa Modelo





Capa Servicios (1)

- Expone remotamente la capa modelo que creamos en el Tema 3
- Sin embargo, los servicios no expondrán directamente los objetos **Movie** y **Sale** del modelo
- La razón es que en este ejemplo se asume que el acceso remoto lo van a realizar aplicaciones externas
 - La fecha de creación de la película en la BD es un dato interno que no se desea exponer a aplicaciones externas
 - En cuanto a las ventas, los clientes externos sólo necesitan los identificadores de venta y película, la fecha de expiración y la url para ver la película
- Por tanto, los servicios utilizarán dos objetos alternativos: **ServiceMovieDto** y **ServiceSaleDto**



Capa Servicios (2)

- DTO: Data Transfer Object. Objeto para transferir datos entre aplicaciones. Encapsula diferencias con los objetos internos de cada aplicación
 - Porque no son relevantes externamente (nuestro caso)
 - Porque queremos objetos más "gruesos" para minimizar llamadas remotas.
 - Ejemplo
 - Supongamos que tenemos también información de actores
 - Los actores tienen su propia tabla en la BD, su clase y su DAO
 - Entre otros casos de uso para manejar información de actores, el modelo ofrece el método **getActorsByMovie**, que recibe un **movieId** y devuelve una lista de actores
 - Si sabemos que la mayoría de los clientes, al buscar información de películas van a querer los actores, podríamos hacer que el objeto **ServiceMovieDto** incluyese ya la lista
 - La operación **findMovies** del servicio invocaría a **findMovies** del Modelo y, para cada película, invocaría a **getActorsByMovie** para obtener los actores

Capa Servicios (3)

- Visión ofrecida por la capa servicios

<i>ServiceMovieDto</i>
<ul style="list-style-type: none">- movieId : Long- title : String- runtime : short- description : String- price : float
<ul style="list-style-type: none">+ Constructores+ métodos get/set

<i>ServiceSaleDto</i>
<ul style="list-style-type: none">- saleId : Long- movieId : Long- expirationDate : String- movieUrl : String
<ul style="list-style-type: none">+ Constructores+ métodos get/set



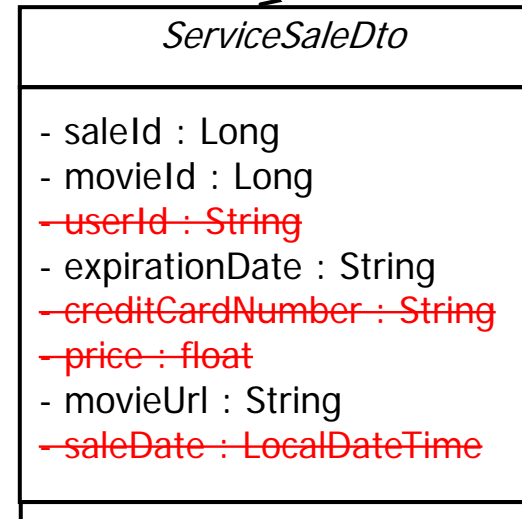
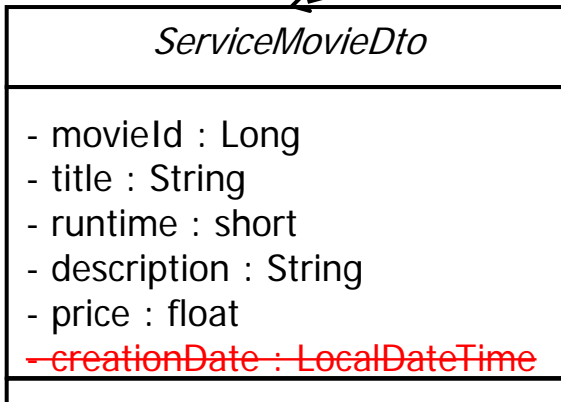
Capa Servicios (y 4)

- Esto ilustra el hecho común de que a las aplicaciones externas se les expone una API diferente que a las internas
 - Diferencias en los objetos expuestos
 - No exponemos la operación **findMovieByID**
 - En casos más complejos podrían ser necesarias validaciones o pasos adicionales en la lógica de negocio
- En nuestro caso, asumimos que las aplicaciones internas que necesiten acceder a más datos de las películas y ventas que los expuestos por la capa Servicios diseñada para las aplicaciones externas, accederían a través de otra capa Servicios con una API menos restringida
- **NOTA:** En **ServiceSaleDto** la propiedad **expirationDate** es tipo **String** en lugar de **LocalDateTime**
 - La fecha se envía al cliente como un String

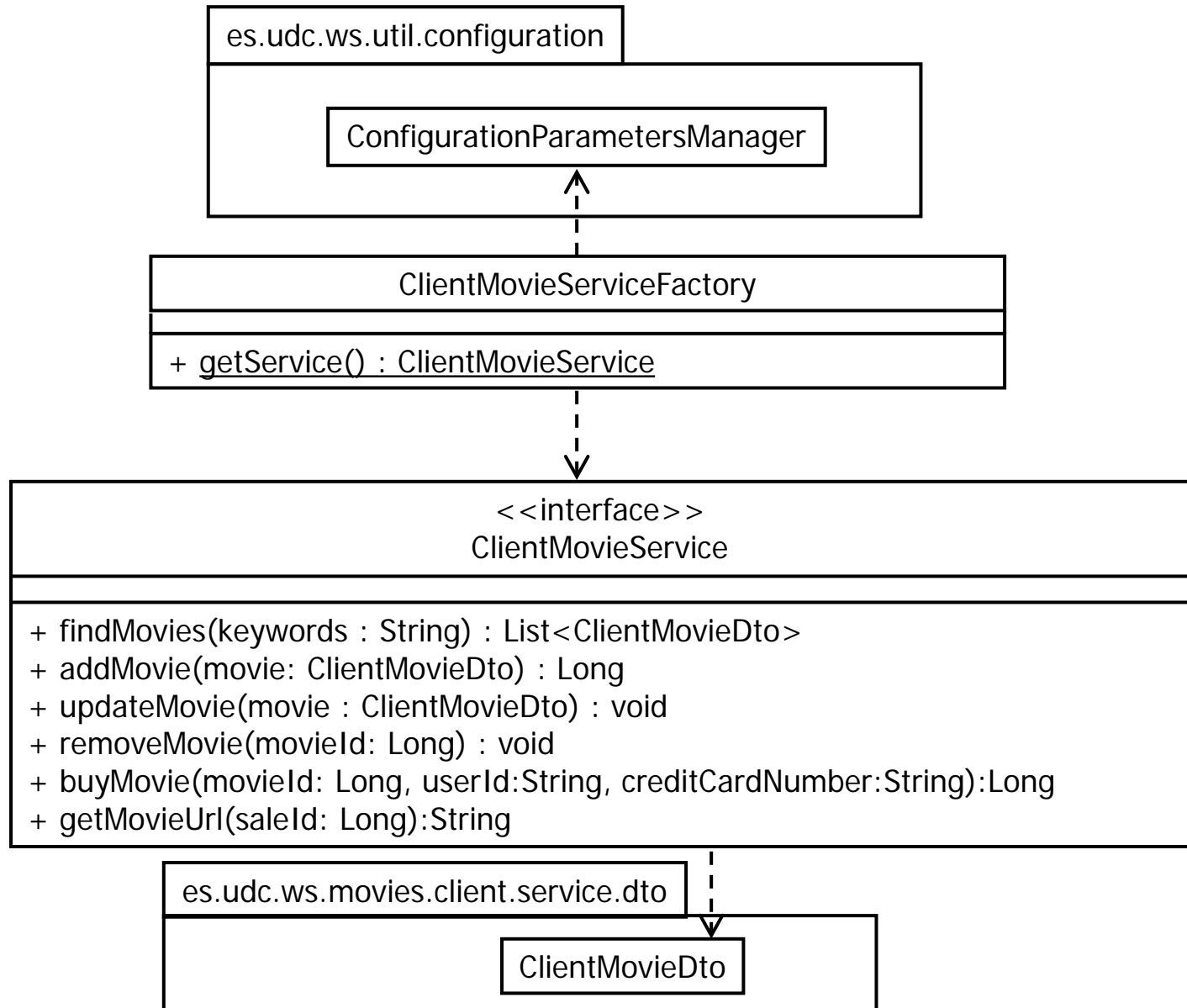
Interfaz de la Capa Servicios

MovieService (Visión ofrecida por Capa Servicios)

- + addMovie(movie: ServiceMovieDto) : ServiceMovieDto
- + updateMovie(movie : ServiceMovieDto) : void
- + removeMovie(movieId: Long) : void
- ~~+ findMovie(movieId : Long) : ServiceMovieDto~~
- + findMovies(keywords : String) : List<ServiceMovieDto>
- + buyMovie(movieId : Long, userId : Long, creditCardNumber : String) : Long
- + findSale(saleId : Long) : ServiceSaleDto



Capa Acceso a Servicios (1)





Capa Acceso a Servicios (2)

- Define una **sencilla API** para acceder al servicio
- **Oculto la tecnología** usada para acceder al servicio
- Paquete **`es.udc.ws.movies.service`**
- **`ClientMovieService`**
 - Patrón Facade
 - Interfaz con un método por cada una de las operaciones del servicio
- **`ClientMovieServiceFactory`**
 - Patrón Factory
 - Permite construir una instancia de **`ClientMovieService`** sin que el llamador necesite conocer la clase que implementa el interfaz
 - **`getService`**
 - Lee de un fichero de configuración el nombre de la clase que implementa la interfaz
 - Construye una instancia de esa clase

Capa Acceso a Servicios (3)

- **ClientMovieService + ClientMovieServiceFactory**
 - Además de las anteriores ventajas (sencillez + ocultación de tecnología), permiten que el cliente pueda acceder a distintas implementaciones del servicio sin necesidad de recompilarlo
 - Basta especificar en la configuración la implementación de **ClientMovieService** que se desea
 - El caso de estudio proporciona dos implementaciones de **ClientMovieService**: una para acceder a un servicio web REST y otra para acceder a un servicio Thrift
 - En ambos casos, los clientes son los mismos (**no se han implementado dos veces**)
- **Ejemplo de uso**

```
ClientMovieService service =  
    ClientMovieServiceFactory.getService();  
List<ClientMovieDto> movies =  
    service.findMovies("batman");
```


Capa Acceso a Servicios (4)

- La aplicación cliente usará el **runtime** dividido en horas y minutos, por lo que usa su propio objeto para representar la información de películas
- Paquete **`es.udc.ws.movies.client.service.dto`**

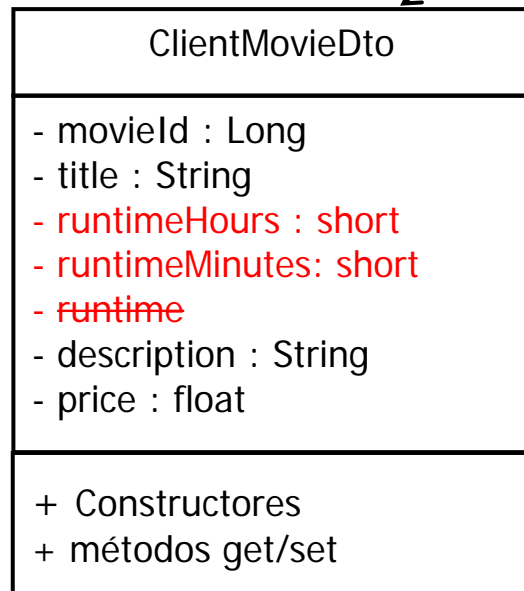
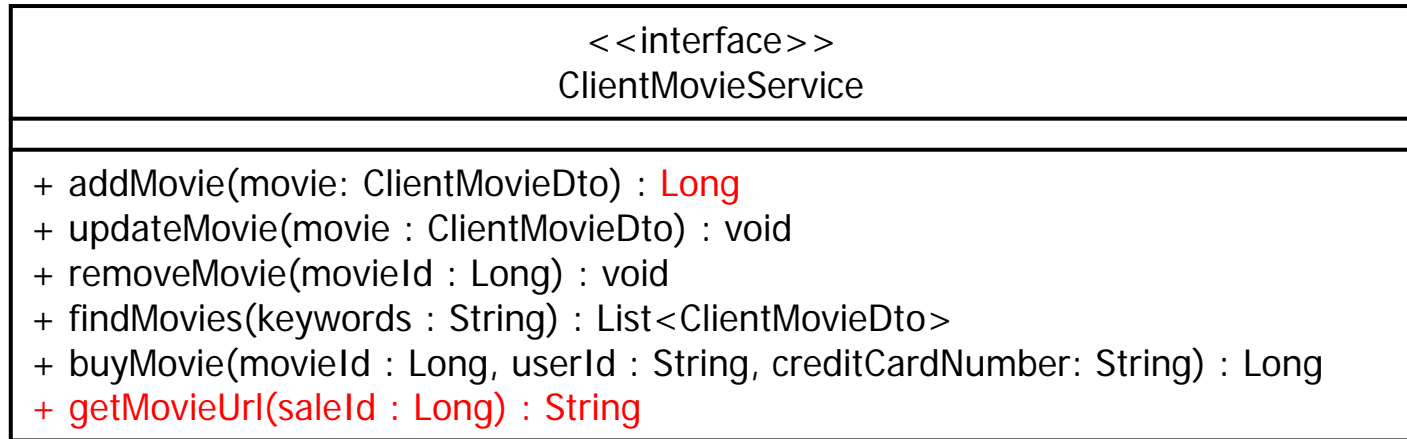
ClientMovieDto
<ul style="list-style-type: none">- movieId : Long- title : String- runtimeHours : short- runtimeMinutes: short- description : String- price : float
<ul style="list-style-type: none">+ Constructores+ métodos get/set



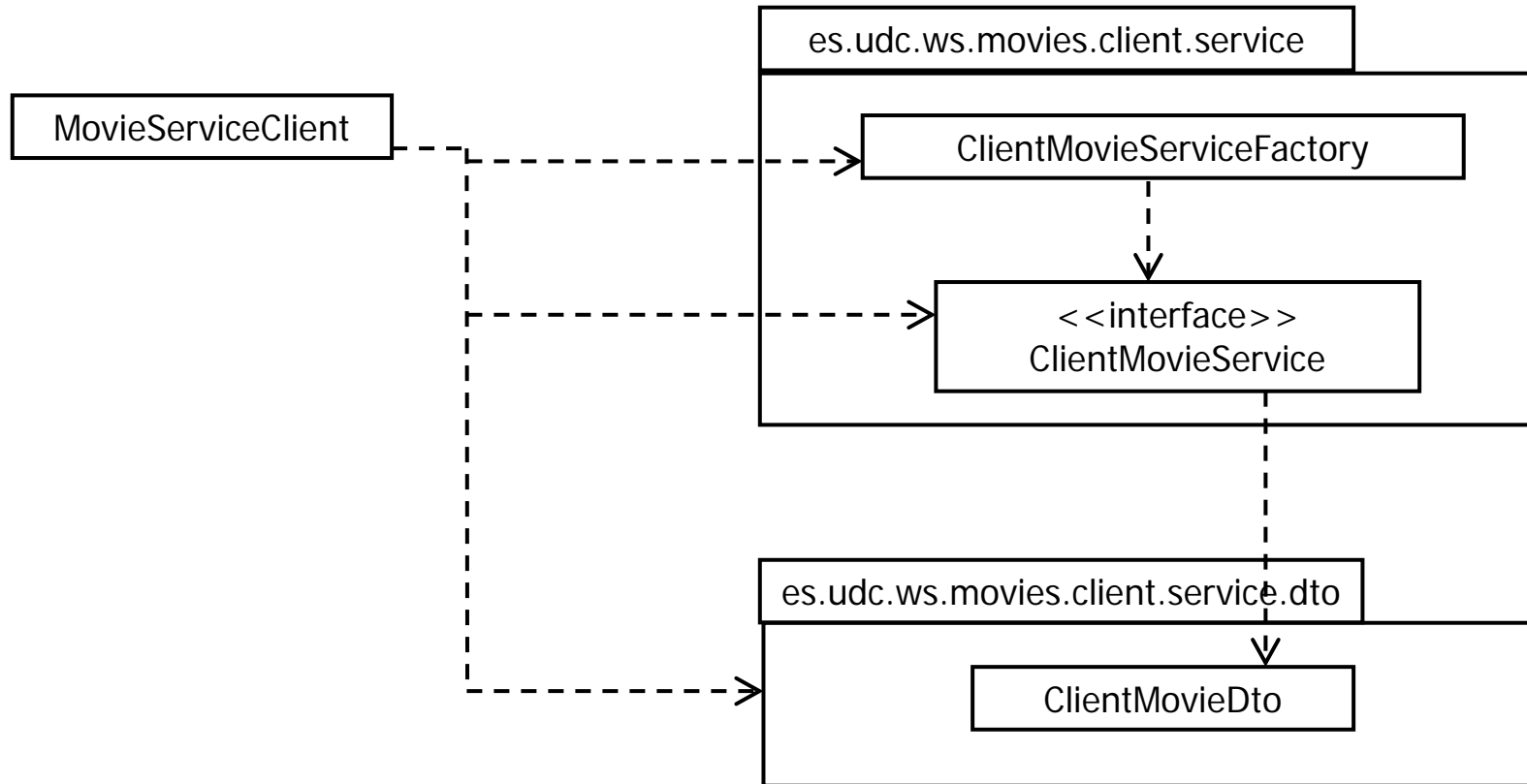
Capa Acceso a Servicios (y 5)

- En los casos reales, lo habitual es que cada cliente use sus propios objetos ya que
 - Permite adaptar los objetos a lo que el cliente necesita
 - Es más natural tener objetos con sólo lo que necesitas
 - Ejemplo
 - Al escribir esta diapositiva, la API de Facebook expone 38 campos y 42 relaciones para un usuario
 - Una aplicación que te avise del cumpleaños de tus amigos sólo necesita dos atributos en su objeto User (name, birthday)

Interfaz de la Capa Acceso a Servicios



Capa Interfaz de Usuario (1)





Capa Interfaz de Usuario (2)

- Cliente MovieServiceClient

- Añadir película

- MovieServiceClient -a <title> <hours> <minutes>
<description> <price>

- Borrar película

- MovieServiceClient -r <movieId>

- Actualizar película

- MovieServiceClient -u <movieId> <title> <hours> <minutes>
<description> <price>

- Buscar películas por keywords en el título

- MovieServiceClient -f <keywords>

- Comprar película

- MovieServiceClient -b <movieId> <userId>
<creditCardNumber>

- Ver película (recuperar venta)

- MovieServiceClient -g <saleId>



Capa Interfaz de Usuario (y 3)

- NOTA

- Tenemos un único cliente que puede ejecutarse desde la intranet del servicio o desde Internet
- Sin embargo, en un caso real, posiblemente habría al menos dos tipos de clientes
 - Uno para las labores de administración que se ejecutaría normalmente desde la intranet del servicio
 - Otro para buscar, comprar y ver películas que se usaría normalmente desde Internet
- El cliente especifica la duración de la película separándola en horas y minutos

Tema 6.2: Introducción a los Servicios Web REST



- Introducción
- Recordatorio de HTTP
- Características Servicios REST
 - Servicios "stateless"
 - Recursos y representaciones
 - Interfaz uniforme
 - Intermediarios
- 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
 - 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 última versión es HTTP 2
- Protocolo cliente/servidor utilizado en la Web
- Inicialmente construido para transferir páginas HTML, pero puede transferir cualquier contenido
 - Las peticiones y respuestas HTTP usan la cabecera Content-Type para indicar el tipo de MIME (e.g. text/html, application/json, image/jpeg, etc.) que indica la naturaleza de la información que se envía en el cuerpo (e.g. Content-Type: application/json)
- Esquema petición/respuesta
- Utiliza TCP para comunicar cliente y servidor (puerto reservado: 80)



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)
 - Solo con algunos métodos de acceso



Peticiones HTTP (2)

- Métodos de acceso

- GET

- Solicita una representación del recurso especificado (e.g. página HTML)
- Es seguro (*safe*)
 - Semántica esencialmente de solo lectura
 - El cliente no pide ni espera que se produzcan cambios en el servidor como resultado de realizar una petición sobre un recurso utilizando este método
 - Un cliente puede utilizarlo sin miedo a causar efectos indeseados
- 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>



Peticiones HTTP (3)

- Métodos de acceso

- PUT

- Carga en el servidor una representación de un recurso
- La nueva representación del recurso va en el cuerpo de la petición
- No es seguro

- DELETE

- Elimina el recurso especificado
- No es seguro

- POST

- Envía datos al recurso indicado para que los procese
 - Los datos van en el cuerpo de la petición
- Puede utilizarse para
 - Crear un nuevo recurso
 - Modificar un recurso existente
- No es seguro



Peticiones HTTP (4)

- Peticiones GET, PUT, DELETE deben ser idempotentes
 - Múltiples peticiones iguales deben tener el mismo efecto en el servidor que una sola
- Las peticiones POST no tienen porque ser idempotentes



Peticiones HTTP (y 5)

- Cabeceras de la petición
 - Tipo MIME de datos enviados en el cuerpo
 - 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
 - 403 Forbidden
 - 404 Not Found
 - 500 Internal Error
- Lista completa: <https://datatracker.ietf.org/doc/html/rfc7231#section-6>
- 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")
- Recursos y representaciones
- Interfaz uniforme
- Sistema en capas: Intermediarios
- Hipermedia (no lo veremos)
- Representaciones autodescriptivas (no lo veremos)



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
 - Por tanto, cualquier servicio puede referenciar y acceder a un recurso de cualquier otro servicio



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">

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



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`

- Es segura y es idempotente

- 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
- No es segura y es idempotente

- DELETE

- Borra el recurso
- No suele permitirse sobre recursos colección
- No es segura y es idempotente



Interfaz Uniforme (3)

En la práctica, se utiliza HTTP (cont.)

- 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 (o colección), puede usarse para modelar operaciones no seguras que no encajen con los otros métodos → **overloaded POST**
 - En este caso la petición debe incluir en algún otro lugar (en la URI, cabeceras o cuerpo) información sobre la operación a invocar sobre el recurso
 - Semejante a estilo RPC (no sigue principios de interfaz uniforme)
- No es segura y puede no ser idempotente



Interfaz Uniforme (4)

Códigos de respuesta estándar HTTP

- 200 – OK
- 201 – Created
- 400 – Bad Request
- 403 – Forbidden
- 404 – Not Found
- 410 – Gone
- 500 – Internal Error
- ...

■ Códigos de error permanente / temporal

- Es importante poder distinguir entre códigos de respuesta que indican que un error es permanente o temporal
- La especificación HTTP no dice explícitamente si un código de error tiene asociada una semántica de error permanente o temporal



Interfaz Uniforme (5)

Códigos de error permanente / temporal (cont.)

- Dentro de una organización siempre es posible establecer convenciones sobre qué códigos representan errores permanentes o temporales
- Si queremos poder utilizar intermediarios transparentemente (como veremos a continuación) también deben utilizar las mismas convenciones
- En la asignatura, atendiendo a su semántica, utilizaremos los siguientes códigos de error
 - Para errores temporales: 404, 500
 - Para errores permanentes: 400, 403, 410
 - NOTA: Para elegir el código concreto a utilizar se tendrá en cuenta la semántica del error concreto, además de si es temporal o permanente



Interfaz Uniforme (y 6)

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: Servidores de Cache intermedios

- En la WWW y en los servicios REST pueden existir múltiples servidores de cache entre cliente y servicio
- Ejemplos de servidores de cache
 - 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)

Ejemplo: Servidores de Cache intermedios (cont.)

- 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, PUT ni 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 y si el código de estado de la respuesta es temporal o permanente
- 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 qué formato viene la respuesta del servidor (cabecera Content-type)



Intermediarios (y 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)



REST en la práctica

- La mayoría de servicios reales no utilizan algunos de los principios de REST
 - Uso limitado de hipermedia (no lo hemos visto)
 - Uso limitado de representaciones autodescriptivas (no lo hemos visto)
 - Nosotros usaremos un enfoque similar a este en el ejemplo y en la práctica



Tema 6.3. Caso de Estudio: Diseño e Implementación de un Servicio REST

- Protocolo REST
- Implementación REST de la capa Servicios
- Implementación REST de la capa Acceso a Servicios
- Comentarios finales



Protocolo REST (1)

■ Seguiremos la misma aproximación que la mayoría de servicios web REST reales

- 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
- 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 JSON en el formato del apartado 5.4
- El parámetro keywords permite filtrarlas por palabras clave en el título: /movies?keywords=Dark+Knight
- Devuelve el código 200 Ok

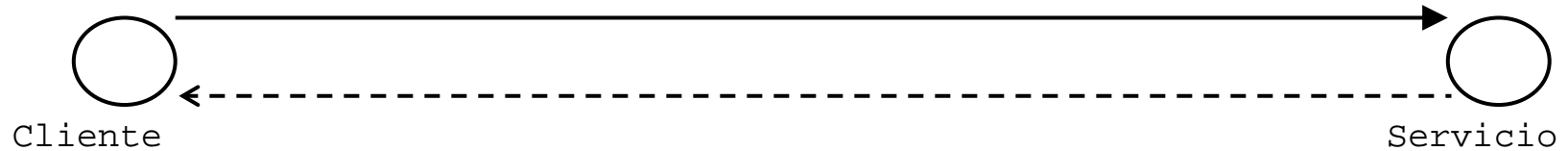
■ POST

- Añade una nueva película
- La película se envía en el cuerpo de la petición en el formato JSON del apartado 5.4 (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)

■ Obtener películas filtradas por palabras clave

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



```
HTTP/1.1 200 OK
...

[
  {
    "movieId": 3,
    "title": "Dark Knight Rises Again",
    "runtime": 165,
    "price": 4.99,
    "description": "Ocho años después de ... "
  },
  {
    "movieId": 5,
    "title": "Dark Knight Returns",
    ...
  }
]
```

Protocolo REST (4)

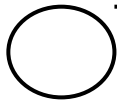
Añadir la información de una película

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

```
{  
  "title": "Dark Knight Rises Again",  
  "runtime": 165,  
  "price": 4.99,  
  "description": "Ocho años después de ... "  
}
```

Cliente

Servicio



HTTP/1.1 201 Created

...

Location: `http://XXX/ws-movies-service/movies/3`

```
{  
  "movieId": 3,  
  "title": "Dark Knight Rises Again",  
  "runtime": 165,  
  "price": 4.99,  
  "description": "Ocho años después de ... "  
}
```

Protocolo REST (5)

Recursos

- `/movies/{id}` Recurso individual por película

- PUT

- Modifica la película
- La película se envía en el cuerpo de la petición en el formato JSON del apartado 5.4
- El cuerpo de la respuesta va vacío
- Devuelve el código 204 No Content

- DELETE

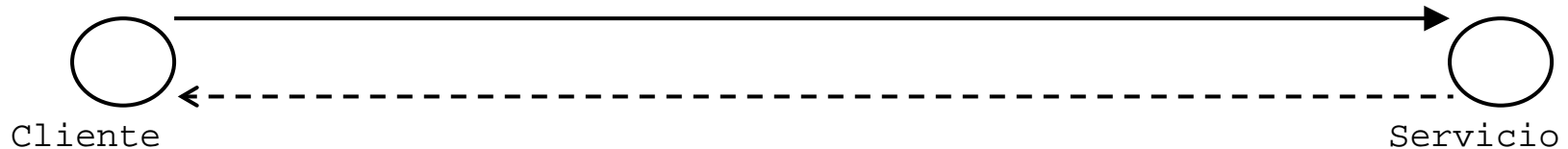
- Borra la película
- El cuerpo de la respuesta va vacío
- Devuelve el código 204 No Content

Protocolo REST (6)

- Actualizar la información de una película

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

```
{  
  "movieId": 3,  
  "title": "Dark Knight Rises Again",  
  "runtime": 165,  
  "price": 3.99,  
  "description": "Ocho años después de ... "  
}
```



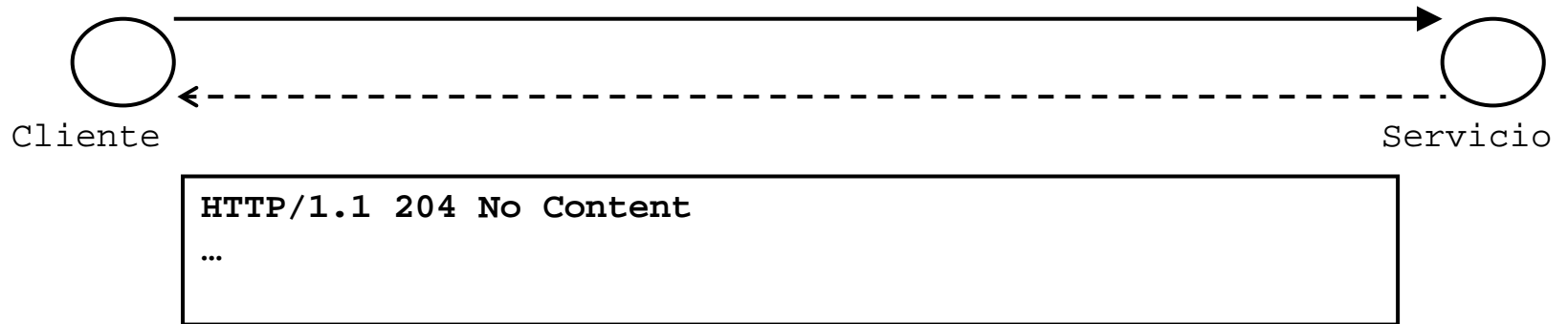
```
HTTP/1.1 204 No Content
```

```
...
```

Protocolo REST (7)

- Eliminar la información de una película

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



Protocolo REST (8)

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 venta 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 JSON
 - No es posible borrar ni modificar ventas una vez producidas

Protocolo REST (9)

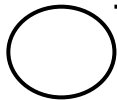
- Añadir la información de una venta

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

```
movieId=7&userId=username&creditCardNumber=1234567890123456
```

Cliente

Servicio



`HTTP/1.1 201 Created`

`...`

`Location: http://XXX/ws-movies-service/sales/5`

```
{  
  "saleId": 5,  
  "movieId": 7,  
  "movieUrl": "http://ws-movies.udc.es/sale/stream/7/...",  
  "expirationDate": "2020-04-17T20:12:47"  
}
```

Protocolo REST (10)

- Obtener la información de una venta

Petición GET a `http://XXX/ws-movies-service/sales/5`



```
HTTP/1.1 200 OK
...
{
  "saleId": 5,
  "movieId": 7,
  "movieUrl": "http://ws-movies.udc.es/sale/stream/7/...",
  "expirationDate": "2020-04-17T20:12:47"
}
```



Protocolo REST (11)

- 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**
 - Se deniega el acceso a la acción solicitada: 403 Forbidden
 - Similar a **MovieNotRemovableException**
 - Error interno de ejecución: 500 Internal Error
- Pero el cuerpo del mensaje puede llevar información adicional (en formato JSON)
 - Normalmente un mismo código de error estará asociado a varias excepciones → Debe indicarse de alguna forma a cuál se refiere en cada caso
 - Representación de los datos de la excepción

Protocolo REST (y 12)

- Crear una película con datos incorrectos

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

```
{  
  "title": "Dark Knight Rises Again",  
  "runtime": -165,  
  "price": 4.99,  
  "description": "Ocho años después de ... "  
}
```

Cliente

Servicio



HTTP/1.1 400 Bad Request

```
...  
{  
  "errorType": "InputValidation",  
  "message": "Invalid runtime value (it must be  
greater than 0 and lower than 1000): -165"  
}
```



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 completo 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 el identificador



Consideraciones Diseño REST (2)

- Códigos de respuesta y de error
 - La ventaja de usar códigos estandarizados es que cualquier cliente o intermediario que utilice ese estándar conoce la semántica de la respuesta
 - Por ejemplo, siguiendo las convenciones explicadas en el apartado 6.2
 - Respuestas 400, 403 y 410 son cacheables, mientras que respuestas 404 y 500 no lo son
 - No tiene sentido reintentar una petición que ha devuelto 400, 403 o 410, pero sí una que ha devuelto 404 o 500 si la operación es idempotente
 - El cuerpo del mensaje puede llevar información adicional para los clientes específicos del servicio
 - Ejemplo: cuánto hace que expiró la venta



Consideraciones Diseño REST (y 3)

- ¿Cómo se modelarían otras operaciones que no encajasen con operaciones CRUD? ➔ **Overloaded POST**

- Ejemplo 1: Marcar una película como "destacada"

Petición POST a `http://XXX/ws-movies-service/movies/{id}/highlight`

- Ejemplo 2: Poner de rebajas una película (e.g. especificando el porcentaje de rebaja)

Petición POST a `http://XXX/ws-movies-service/movies/{id}/reduce`

`percentage=50`

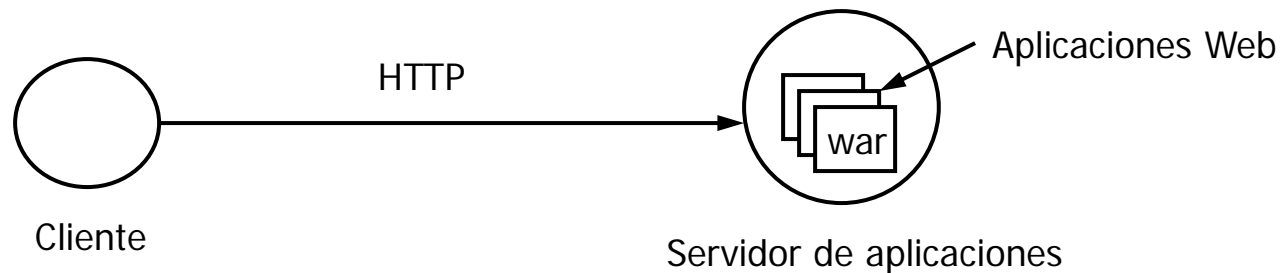
- Ejemplo 3: Rebajar el precio de las películas más antiguas que una cierta fecha (e.g. especificando el porcentaje de rebaja)

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

`date=01-01-2015&percentage=30`

Aplicaciones Web Java EE

- En Java EE / Jakarta EE, las **aplicaciones Web** se instalan en **servidores (contenedores) de aplicaciones**



- **La API de programación Web que ofrece el servidor de aplicaciones está estandarizada**
 - Una aplicación web Java EE puede instalarse en cualquier servidor de aplicaciones Java EE
- **Las aplicaciones Web se distribuyen en ficheros WAR**
 - Un fichero WAR (Web ARchive) es un fichero JAR con una estructura de directorios estandarizada
 - Contiene clases objeto (**.class**), librerías (**.jar**), ficheros de configuración y otras abstracciones propias de una aplicación Web



Aplicaciones Web Java EE (y 2)

- Las APIs de programación Web Java no están pensadas para devolver sólo HTML/XHTML, sino cualquier tipo de información sobre HTTP
- En consecuencia, se pueden utilizar para implementar servicios Web
- Utilizaremos la **API de Servlets** para implementar Servicios Web REST



Visión Global del Framework de Servlets (1)

- Para la implementación de la capa servicios usaremos la 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 web: 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)

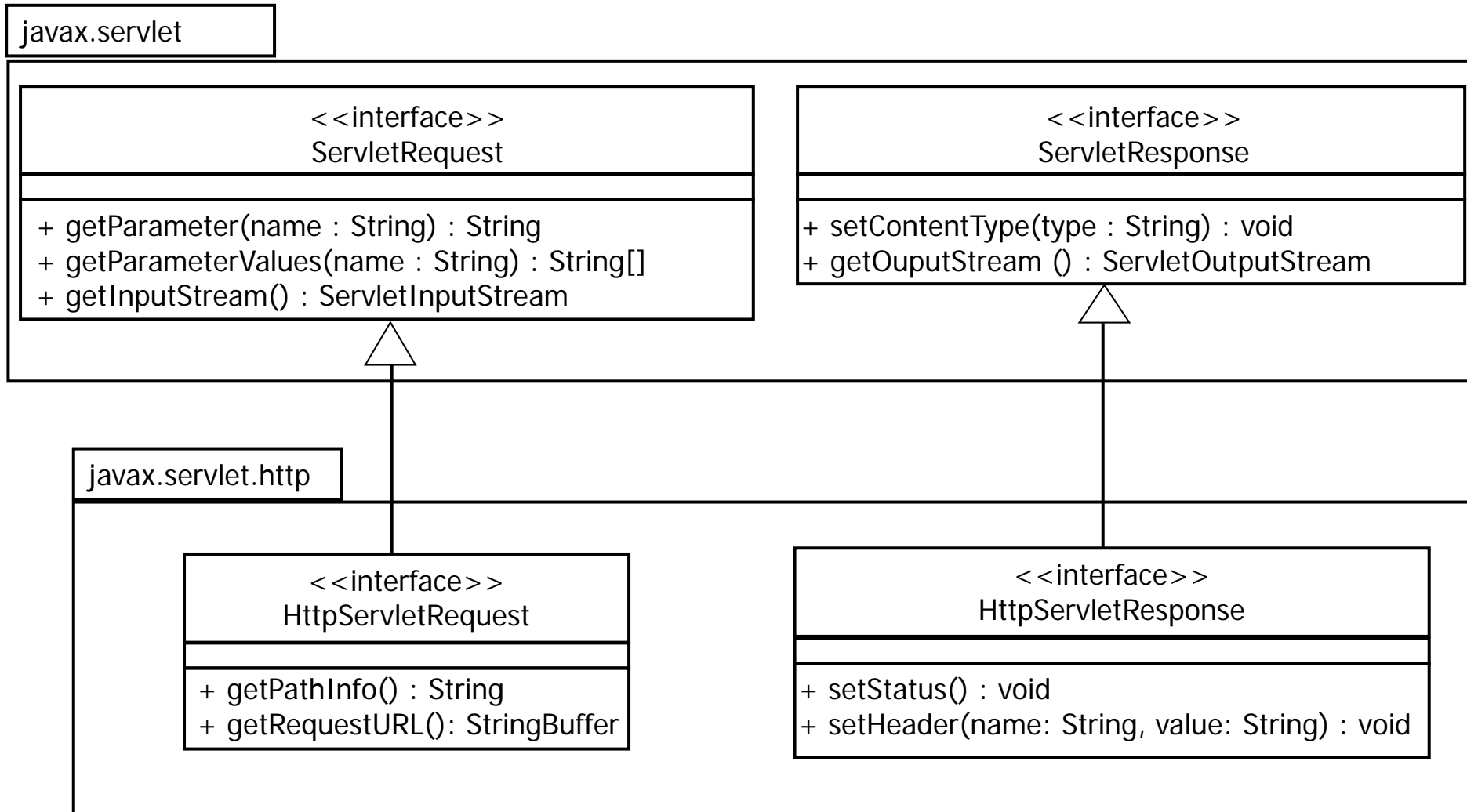
- El desarrollador implementa un servlet extendiendo de **HttpServlet** y redefine los métodos **doXXX** (e.g. **doGet**, **doPost**, **doPut**, **doDelete**, etc.) necesarios, según el tipo de peticiones que vaya a aceptar su servlet
- Cuando el servidor de aplicaciones recibe una petición dirigida a un servlet, invoca la operación correspondiente en función del método de la petición (**doGet**, **doPost**, **doPut**, **doDelete**, etc., según la petición HTTP sea GET, POST, PUT, DELETE, etc.)



Visión Global del Framework de Servlets (3)

- Modelo multi-thread de procesamiento de peticiones
 - Cada vez que el servidor de aplicaciones recibe una petición dirigida a un servlet, la petición se procesa en un thread independiente

Visión Global del Framework de Servlets (4)





Visión Global del Framework de Servlets (5)

■ **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 (5)

■ **HttpServletRequest**

■ **String getPathInfo()**

- Devuelve el fragmento del path de la petición a partir de la URL asociada al Servlet (y siempre empieza por '/')
- Si un Servlet tiene asociado un patrón de URL del tipo `/movies/*` devuelve la parte que encaja con `*`, precedida de '/'

■ Ejemplos

- `/movies` → `getPathInfo` devuelve `null`
- `/movies/` → `getPathInfo` devuelve `"/"`
- `/movies/123` → `getPathInfo` devuelve `"/123"`
- `/movies/123/` → `getPathInfo` devuelve `"/123/"`

■ **String getRequestURL()**

- Devuelve la URL completa que utilizó el cliente para realizar la petición (sin incluir parámetros)



Visión Global del Framework de Servlets (y 6)

■ **ServletResponse**

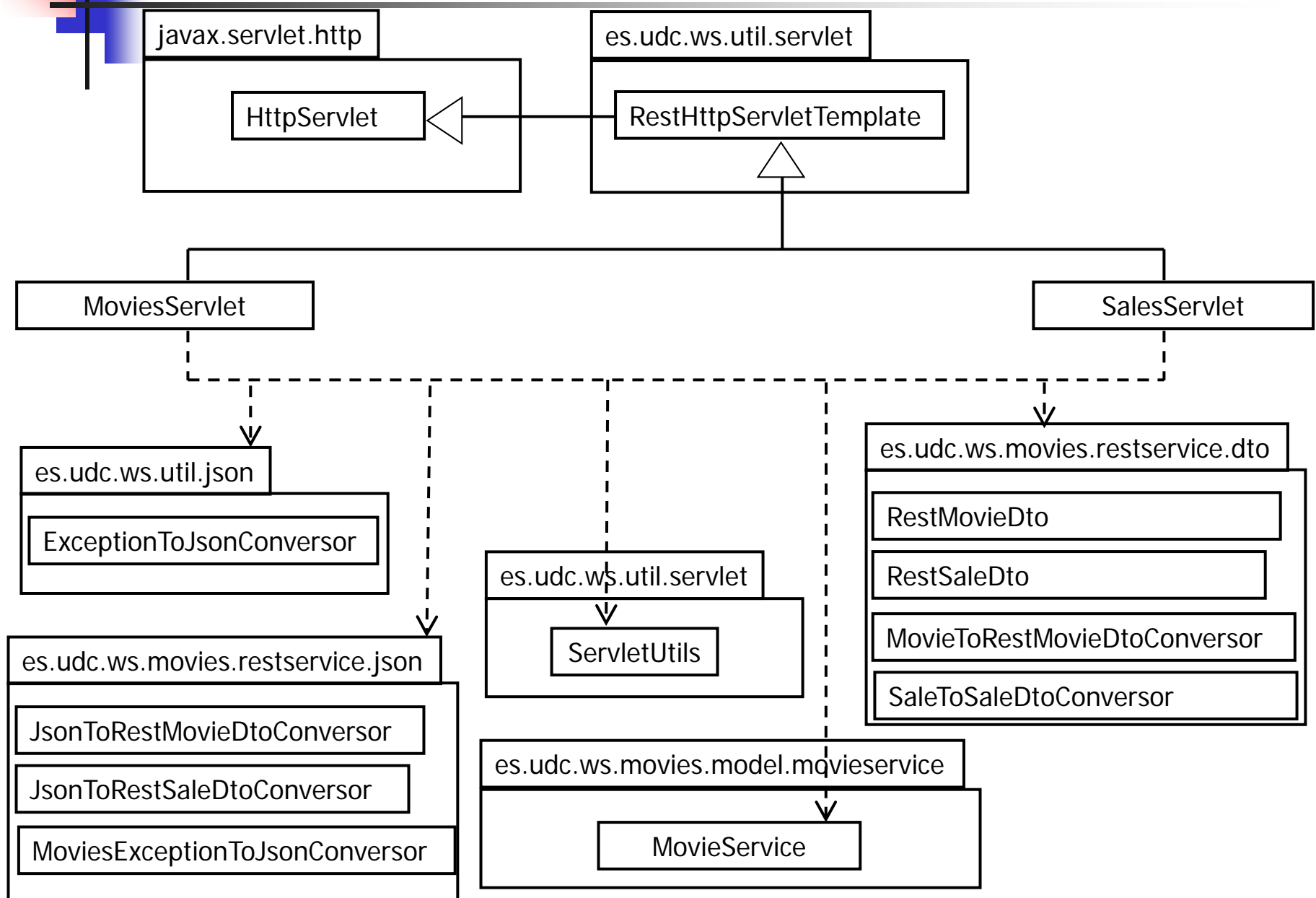
- **void setContentType(String type)**
 - Especifica el tipo de contenido de la respuesta (e.g. "application/json")
- **ServletOutputStream getOutputStream()**
 - Permite escribir el cuerpo de la respuesta
 - **ServletOutputStream** es una clase abstracta que implementa **java.io.OutputStream**

■ **HttpServletResponse**

- **void setStatus(int sc)**
 - Especifica el código de estado de la respuesta
- **void setHeader(String name, String value)**
 - Añade una cabecera con el nombre y el valor especificados a la respuesta

Capa Servicios (1)

[es.udc.ws.movies.restservice.servlets]



RestMovieDto
<ul style="list-style-type: none">- movieId : Long- title : String- runtime : short- description : String- price : float
<ul style="list-style-type: none">+ Constructores+ métodos get/set

RestSaleDto
<ul style="list-style-type: none">- saleId : Long- movieId : Long- expirationDate : String- movieUrl : String
<ul style="list-style-type: none">+ Constructores+ métodos get/set

- **MovieToRestMovieDtoConverter**
 - Realiza conversiones entre **Movie** y **RestMovieDto**
- **SaleToRestSaleDtoConverter**
 - Realiza conversiones entre **sale** y **RestSaleDto**

■ **JsonToRestMovieDtoConverter**

- Realiza conversiones de **RestMovieDto** a/desde JSON
- Su implementación la vimos en la sección 5.4

■ **JsonToRestSaleDtoConverter**

- Transforma a JSON los objetos **RestSaleDto**

■ **MoviesExceptionToJsonConverter**

- Transforma las excepciones del Modelo a su representación JSON, que será un objeto con los siguientes campos:
 - Un campo llamado **errorType** cuyo valor indica el tipo de excepción/error (e.g. **SaleExpiration**)
 - Un campo por cada propiedad de la excepción

■ **ExceptionToJsonConverter**

- Transforma las excepciones del módulo de utilidades a su representación JSON siguiendo el mismo esquema que **MoviesExceptionToJsonConverter**

ServletUtils

- Clase con métodos utilidad para implementar Servlets

ServletUtils

```
+ writeServiceResponse(response : HttpServletResponse, responseCode:int, rootNode : JsonNode,  
    headers : Map<String,String>) : void  
+ normalizePath(url : String) : String  
+ getMandatoryParameter(req : HttpServletRequest, paramName : String) : String  
+ getMandatoryParameterAsLong(req : HttpServletRequest, paramName : String) : Long  
+ checkEmptyPath(req : HttpServletRequest) : void  
+ getIdFromPath(req : HttpServletRequest, resourceName : String) : Long
```

- **writeServiceResponse** escribe una respuesta HTTP. Recibe como parámetros
 - Un objeto de tipo **HttpServletResponse**
 - El código a enviar en la respuesta
 - El cuerpo a enviar en la respuesta
 - Objeto **JsonNode** de Jackson que debe ser el nodo raíz del árbol correspondiente al objeto o array que se quiere enviar
 - Un mapa con los nombres y valores de cabeceras a añadir a la respuesta

ServletUtils

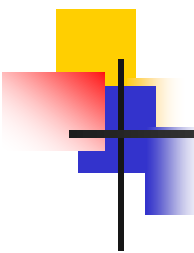
- **normalizePath** recibe un String y devuelve otro String igual al recibido quitándole el carácter '/' en caso de que finalice con él
- **getMandatoryParameter** obtiene el valor del parámetro indicado o lanza una **InputValidationException** si el parámetro no viene en la petición
- **getMandatoryParameterAsLong** hace lo mismo que el método anterior pero además convierte el valor del parámetro a Long y lanza una **InputValidationException** si no es posible
- **checkEmptyPath** comprueba si el path de una petición HTTP es vacío, nulo o está compuesto solo por "/" y en caso negativo lanza una **InputValidationException**
- **getIdFromPath** obtiene un identificador de tipo Long a partir del path de una petición HTTP que siga el formato "/<id>". En caso de no poder obtener el identificador lanza una **InputValidationException**

```
public class RestHttpServletTemplate extends HttpServlet {

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

            processGet(req, resp);

        } catch (InstanceNotFoundException ex) {
            ServletUtils.writeServiceResponse(resp,
                HttpServletResponse.SC_NOT_FOUND,
                ExceptionToJsonConversor.toInstanceNotFoundException(ex), null);
        } catch (InputValidationException ex) {
            ServletUtils.writeServiceResponse(resp,
                HttpServletResponse.SC_BAD_REQUEST,
                ExceptionToJsonConversor.toInputValidationException(ex), null);
        } catch (ParsingException ex) {
            ServletUtils.writeServiceResponse(resp,
                HttpServletResponse.SC_BAD_REQUEST,
                ExceptionToJsonConversor.toInputValidationException(
                    new InputValidationException(ex.getMessage())), null);
        }
    }
}
```



```
protected void processGet(HttpServletRequest req,
    HttpServletResponse resp) throws IOException,
    InstanceNotFoundException, InputValidationException {

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

```
@Override
```

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

```
protected void processPost(HttpServletRequest req,
    HttpServletResponse resp) throws IOException,
    InstanceNotFoundException, InputValidationException {...}
```

```
// doPut + processPut
```

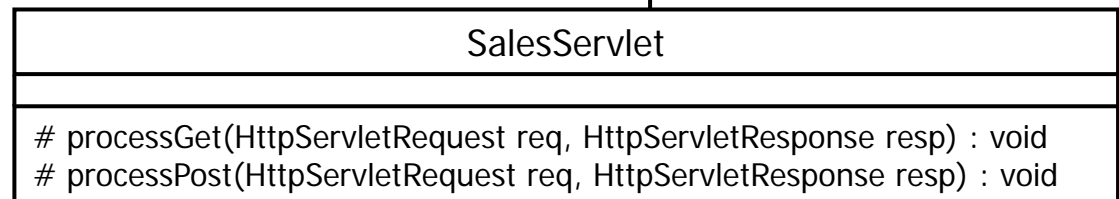
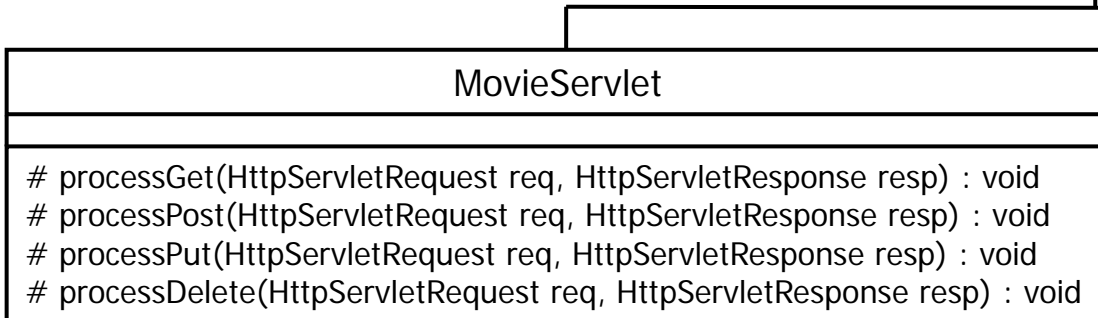
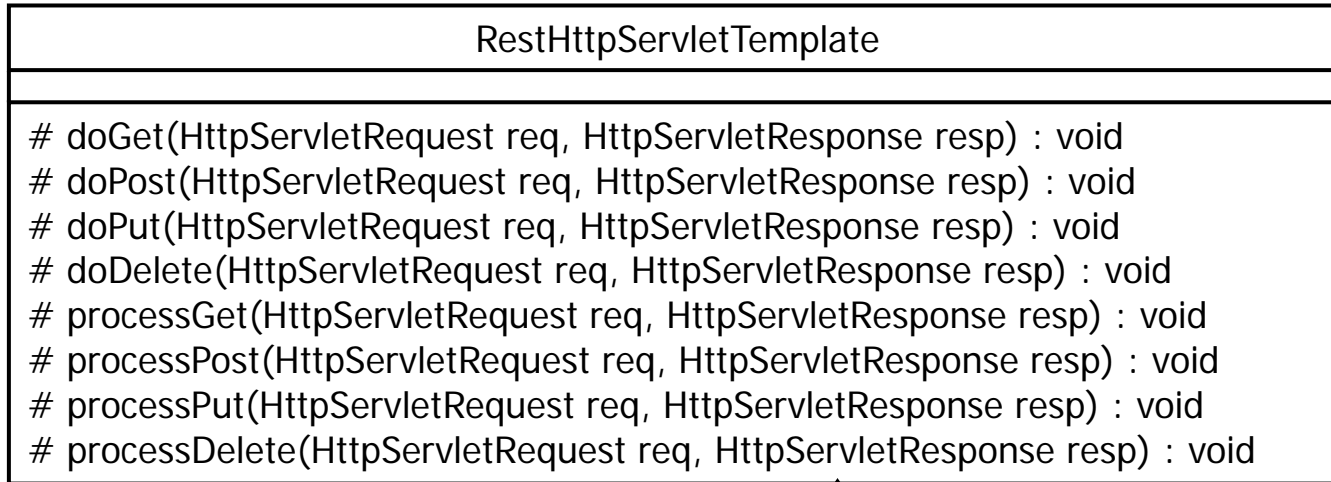
```
...
```

```
// doDelete + processDelete
```

```
...
```

```
}
```


RestHttpServletTemplate



■ RestHttpServletTemplate

- Clase para facilitar la implementación de Servlets que invoquen a operaciones de una capa Modelo que lance excepciones definidas en **ws-util**
- Extiende a **HttpServlet**
- Patrón Template Method: implementa las operaciones **doXXX** (Get, Post, Put y Delete) llamando a la operación **processXXX**
 - Se capturan las excepciones del módulo **ws-util** al procesar cualquier petición y se envía la respuesta adecuada para cada una de ellas → los Servlets que extiendan de esta clase no tienen que preocuparse de tratar estas excepciones
- La implementación por defecto de **processXXX** devuelve un código de respuesta **409 (SC_NOT_IMPLEMENTED)** para indicar que esa operación no está implementada
 - Los Servlets que extiendan de esta clase solo tienen que implementar los métodos **processXXX** acordes a los métodos HTTP que deban soportar

```
public class MoviesServlet extends RestHttpServletTemplate {

    @Override
    protected void processGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException, InputValidationException {

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

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

        List<RestMovieDto> movieDtos =
            MovieToRestMovieDtoConversor.toRestMovieDtos(movies);

        ServletUtils.writeServiceResponse(resp,
            HttpServletResponse.SC_OK,
            JsonToRestMovieDtoConversor.toArrayNode(movieDtos),
            null);

    }
}
```

@Override

```
protected void processPost(HttpServletRequest req, HttpServletResponse resp)
    throws IOException, InputValidationException {
```

```
    ServletUtils.checkEmptyPath(req);
```

```
    RestMovieDto movieDto =
```

```
        JsonToRestMovieDtoConversor.toRestMovieDto(req.getInputStream());
```

```
    Movie movie = MovieToRestMovieDtoConversor.toMovie(movieDto);
```

```
    movie = MovieServiceFactory.getService().addMovie(movie);
```

```
    movieDto = MovieToRestMovieDtoConversor.toRestMovieDto(movie);
```

```
    String movieURL =
```

```
        ServletUtils.normalizePath(req.getRequestURL().toString()) + "/" +
        movie.getMovieId();
```

```
    Map<String, String> headers = new HashMap<>(1);
```

```
    headers.put("Location", movieURL);
```

```
    ServletUtils.writeServiceResponse(resp,
```

```
        HttpServletResponse.SC_CREATED,
```

```
        JsonToRestMovieDtoConversor.toObjectNode(movieDto),
```

```
        headers);
```

```
}
```

@Override

```
protected void processDelete(HttpServletRequest req, HttpServletResponse resp)
    throws IOException, InputValidationException, InstanceNotFoundException {

    Long movieId = ServletUtils.getIdFromPath(req, "movie");

    try {
        MovieServiceFactory.getService().removeMovie(movieId);
    } catch (MovieNotRemovableException ex) {
        ServletUtils.writeServiceResponse(resp,
            HttpServletResponse.SC_FORBIDDEN,
            MoviesExceptionToJsonConversor.toMovieNotRemovableException(ex),
            null);
        return;
    }

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



```
@Override
```

```
protected void processPut(HttpServletRequest req, HttpServletResponse resp)  
    throws IOException, InputValidationException,  
        InstanceNotFoundException {
```

```
    Long movieId = ServletUtils.getIdFromPath(req, "movie");
```

```
    RestMovieDto movieDto =
```

```
        JsonToRestMovieDtoConversor.toRestMovieDto(req.getInputStream());
```

```
    if (!movieId.equals(movieDto.getMovieId())) {
```

```
        throw new InputValidationException("Invalid Request: invalid movieId");
```

```
    }
```

```
    Movie movie = MovieToRestMovieDtoConversor.toMovie(movieDto);
```

```
    MovieServiceFactory.getService().updateMovie(movie);
```

```
    ServletUtils.writeServiceResponse(resp,
```

```
        HttpServletResponse.SC_NO_CONTENT, null, null);
```

```
}
```

```
}
```

■ SalesServlet

- **processPost**: similar al de **MoviesServlet** (invocando a la operación **buyMovie** del modelo)
 - A diferencia del **processPost** de **MovieServlet**, el cliente envía los datos de entrada como parámetros en el cuerpo de la petición en lugar de usar una representación JSON (en un POST los parámetros van en el cuerpo y no en la URL)
 - Los parámetros se leen de la misma forma que si fuese una petición GET y los parámetros figurasen en la URL (usando los métodos **getParameter** o **getParameterValues** de **HttpServletRequest**)
- **processGet**: obtiene el identificador de la venta del path de la petición (de forma similar a **processDelete** de **MoviesServlet**)
- 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



Empaquetamiento de una aplicación Web (1)

- **jar cvf aplicacionWeb.war directorio**
 - Opciones similares al comando Unix **tar**
 - El nombre de una aplicación Web no tiene porque coincidir con el de su fichero **.war**
 - El nombre se decide al instalar el fichero **.war** en el servidor de aplicaciones
- Maven genera automáticamente ficheros **.war** para los módulos con empaquetamiento "war"
- Estructura de un fichero **.war**
 - Directorio **WEB-INF/classes**
 - Ficheros **.class** que conforman la aplicación Web, agrupados en directorios según su estructura en paquetes
 - ¡Sin ficheros fuente!
 - Directorio **WEB-INF/lib**
 - Ficheros **.jar** de librerías que usa la aplicación
 - ¡Sin ficheros fuente!



Empaquetamiento de una aplicación Web (y 2)

- Estructura de un fichero **.war** (cont)
 - **WEB-INF/web.xml**
 - Configuración estándar de la aplicación Web
 - Directorio raíz y subdirectorios
 - Vista de la aplicación (ej.: ficheros HTML, páginas JSP, imágenes, etc.)
 - Visible a los navegadores
 - Lo que hay debajo de **WEB-INF** sólo es visible a los servlets y páginas JSP de la aplicación
- Un fichero **.war** se puede instalar (deployment) en cualquier servidor de aplicaciones Java EE / Jakarta EE



web.xml (1)

[Capa Servicios]

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <display-name>JavaExamples Movies Service</display-name>

    <!-- 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>
```

...

```
</web-app>
```



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**



Visión global de HttpClient (1)

- Para la implementación de la capa Acceso a Servicios utilizaremos Jakarta Commons HttpClient
 - Framework open-source de Apache
 - Forma parte de Apache HttpComponents (<http://hc.apache.org>)
 - Utilizaremos la "Fluent Facade API"
 - Simplifica el uso de HttpClient, permitiendo que el desarrollador no tenga que ocuparse de aspectos como el manejo y liberación de conexiones
 - Utiliza la idea de "Method chaining"
 - No permite utilizar toda la funcionalidad de HttpClient, solo la más común

Visión global de HttpClient (2)

Request

- Representa una petición HTTP
- Por cada tipo de operación HTTP (GET, POST, etc.) existe un método del mismo nombre que recibe la URL del recurso
- El método **bodyStream** permite asignar un contenido cualquiera al cuerpo de la petición
- El método **execute** envía la petición
- El método **returnResponse** permite obtener un objeto **HttpResponse** que representa la respuesta a la petición
- Ejemplo

```
HttpResponse response =  
    Request.Post("http://...").  
        bodyStream(toInputStream(someJsonContent),  
            ContentType.create("application/json")).  
        execute().returnResponse();
```



Visión global de HttpClient (3)

Request (cont.)

- En peticiones GET los parámetros se envían en la URL
- En peticiones POST, van en el cuerpo de la petición en un formato establecido (el mismo utilizado para enviar datos de formularios HTML)
- El método **bodyForm** permite incluir en el cuerpo de la petición parámetros en este formato
- Ejemplo

```
HttpResponse response =  
    Request.Post("http://...").  
        bodyForm(Form.form().add("param1", "value1").add(  
            "param2", "value2").build()).  
        execute().returnResponse();
```

- ¿Parámetros o representación ad-hoc en el cuerpo?
 - Usar parámetros es más simple pero está limitado al envío de pares campo/valor



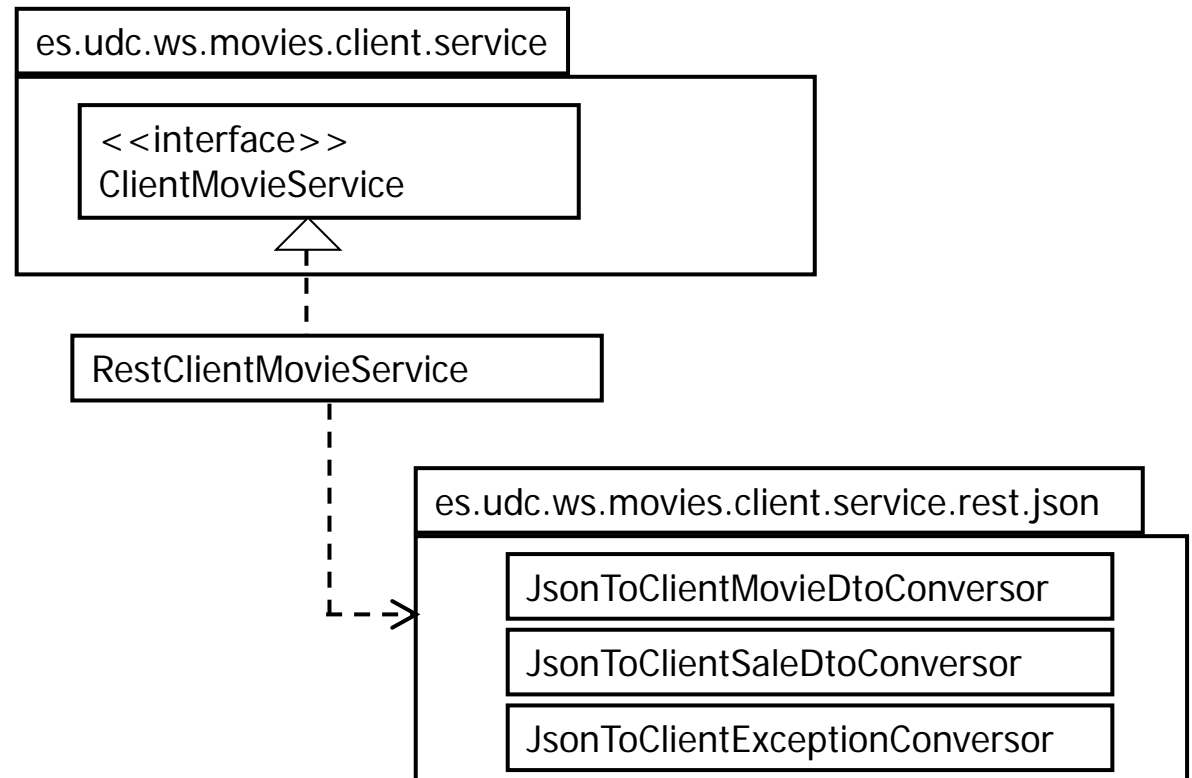
Visión global de HttpClient (y 4)

■ **HttpResponse**

- Representa la respuesta a una petición HTTP
- El código de respuesta puede obtenerse llamando a **getStatusLine().getStatusCode()**
 - La clase **HttpStatus** Declara constantes para cada uno de los códigos de estado: **SC_OK** (200), **SC_NOT_FOUND** (404), **SC_INTERNAL_SERVER_ERROR** (500), etc.
- El cuerpo de la respuesta puede obtenerse llamando a **getEntity().getContent()**
- El método **getFirstHeader(name)** permite obtener un objeto **Header** representando la cabecera con el nombre especificado
- El método **getValue()** de **Header** permite obtener su valor

Capa Acceso a Servicios (1)

[es.udc.ws.movies.client.service.rest]

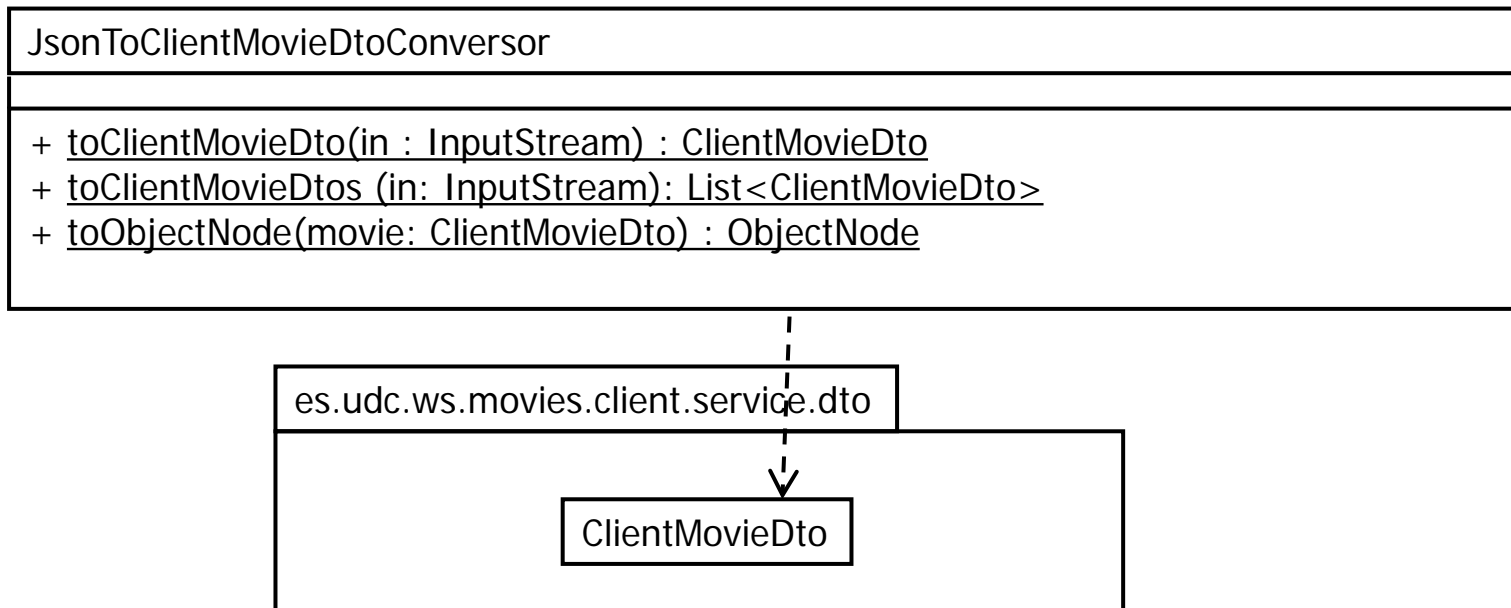


■ **RestClientMovieService**

- Utiliza Jakarta Commons HttpClient para realizar las peticiones HTTP
- Utiliza las clases del subpaquete **json** para parsear / generar JSON

■ **JsonToClientMovieDtoConverter**

- Realiza conversiones de **ClientMovieDto** a/desde JSON
- **toClientMovieDto** y **toObjectNode**
 - Su implementación es muy similar a la de los métodos equivalentes en **JsonToRestMovieDtoConverter** (ver sección 5.4), pero usando **ClientMovieDto** en lugar de **RestMovieDto**
- **toClientMovieDtos**
 - Recibe un stream conteniendo una lista de películas en JSON
 - Se utiliza para parsear las películas resultado de una consulta



■ **JsonToClientSaleDtoConverter**

- Obtiene los datos de una venta (**ClientSaleDto**) desde su representación en JSON
- No se necesitan métodos que conviertan una venta a su representación JSON, ni que conviertan listas de ventas

■ **JsonToClientExceptionConverter**

- Convierte desde la representación en JSON de las excepciones a las excepciones usadas por el cliente

@Override

```
public Long addMovie(ClientMovieDto movie) throws InputValidationException {
    try {
        HttpResponse response =
            Request.Post(getEndpointAddress() + "movies").
                bodyStream(toInputStream(movie),
                    ContentType.create("application/json")).
                execute().returnResponse();

        validateStatusCode(HttpStatus.SC_CREATED, response);

        return JsonToClientMovieDtoConversor.toClientMovieDto(
            response.getEntity().getContent()).getMovieId();

    } catch (InputValidationException e) {
        throw e;
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

```
@Override
public List<ClientMovieDto> findMovies(String keywords) {
    try {
        HttpResponse response =
            Request.Get(getEndpointAddress() + "movies?keywords="
                + URLEncoder.encode(keywords, "UTF-8")).
                execute().returnResponse();

        validateStatusCode(HttpStatus.SC_OK, response);

        return JsonToClientMovieDtoConversor.toClientMovieDtos(
            response.getEntity().getContent());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

```
@Override
public void removeMovie(Long movieId) throws InstanceNotFoundException,
    ClientMovieNotRemovableException {

    try {
        HttpResponse response =
            Request.Delete(getEndpointAddress() + "movies/" + movieId).
                execute().returnResponse();

        validateStatusCode(HttpStatus.SC_NO_CONTENT, response);

    } catch (InstanceNotFoundException |
        ClientMovieNotRemovableException e) {
        throw e;
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

@Override

```
public Long buyMovie(Long movieId, String userId, String creditCardNumber)
    throws InstanceNotFoundException, InputValidationException {
```

```
    try {
```

```
        HttpResponse response =
```

```
            Request.Post(getEndpointAddress() + "sales").
```

```
                bodyForm(Form.form().
```

```
                    add("movieId", Long.toString(movieId)).
```

```
                    add("userId", userId).
```

```
                    add("creditCardNumber", creditCardNumber).
```

```
                    build()).
```

```
            execute().returnResponse();
```

```
            validateStatusCode(HttpStatus.SC_CREATED, response);
```

```
            return JsonToClientSaleDtoConversor.toClientSaleDto(
```

```
                response.getEntity().getContent()).getSaleId();
```

```
        } catch (InputValidationException | InstanceNotFoundException e) {
            throw e;
```

```
        } catch (Exception e) {
            throw new RuntimeException(e);
```

```
        }
```

```
    }
```



```
private InputStream toInputStream(ClientMovieDto movie) {
    try {
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        ObjectMapper objectMapper = ObjectMapperFactory.instance();

        objectMapper.
            writer(new DefaultPrettyPrinter()).
            writeValue(outputStream,
                JsonToClientMovieDtoConversor.toObjectNode(movie));

        return new ByteArrayInputStream(outputStream.toByteArray());

    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

```
private void validateStatusCode(int successCode, HttpResponseMessage response)
    throws Exception {
    try {
        int statusCode = response.getStatusLine().getStatusCode();
        /* Success? */
        if (statusCode == successCode) { return; }
        /* Handler error. */
        switch (statusCode) {
            case HttpStatus.SC_NOT_FOUND:
                throw JsonToClientExceptionConversor.fromNotFoundErrorCode(
                    response.getEntity().getContent());
            case HttpStatus.SC_BAD_REQUEST:
                throw JsonToClientExceptionConversor.fromBadRequestErrorCode(
                    response.getEntity().getContent());
            case HttpStatus.SC_FORBIDDEN:
                throw JsonToClientExceptionConversor.fromForbiddenErrorCode(
                    response.getEntity().getContent());
            case HttpStatus.SC_GONE:
                throw JsonToClientExceptionConversor.fromGoneErrorCode(
                    response.getEntity().getContent());
            default:
                throw new RuntimeException("HTTP error; status code = "
                    + statusCode);
        }
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```



Comentarios

- **toInputStream** convierte un objeto **ClientMovieDto** a su representación JSON y devuelve un **InputStream** del que poder leer el JSON
- No se muestran los métodos
 - **updateMovie**: invoca un PUT sobre el recurso `/movies/{id}` enviando los datos de la película en el cuerpo (similar a **addMovie**)
 - **getMovieUrl**: invoca un GET sobre el recurso `/sales/{id}` y devuelve la URL del objeto **ClientSaleDto** obtenido como respuesta
 - **getEndpointAddress**: Obtiene la parte fija de la URL desde el fichero de configuración usando la clase **ConfigurationParametersManager**
- **validateStatusCode**
 - En el caso de estudio el código de error HTTP permitiría identificar la excepción
 - Sin embargo, en general, debe usarse el cuerpo del mensaje ya que podría haber varias excepciones asociadas al mismo código HTTP (ver implementación de los métodos **JsonToClientExceptionConversor.fromXXXErrorCode**)



Validación de documentos (1)

- Tanto los clientes como el servicio del ejemplo comprueban que el JSON está bien formado
 - Lo hace Jackson automáticamente
- Ni los clientes ni el servicio comprueban que el documento es válido
 - **Sin embargo, se hacen las comprobaciones necesarias**
 - La capa modelo comprueba que los parámetros y datos recibidos son válidos (e.g. números de tarjeta de crédito)



Validación de documentos (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 JSON/XML del protocolo sin que dejen de funcionar clientes y/o servicios
 - **NOTA:** Con JSON Schema sería posible hacer una validación no estricta, es decir, validar solamente un conjunto fijo de campos y no dar error si el documento contiene más campos
 - Puede conseguirse no usando **additionalProperties** en los objetos (o especificando **true** como valor)
 - Esta característica, que no tienen los esquemas XML, ayuda a facilitar la evolución en el tiempo



Validación de documentos (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 JSON/XML. En este momento, los clientes no están actualizados
 - Sin embargo, no dejan de funcionar, dado que en el JSON/XML que reciben sólo preguntan por los tags que conocen (e.g. **movieId**, **title**, **runtime**, etc.)
 - 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 (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)