UNIVERSIDADE DA CORUÑA

Departamento de Ciencias de la Computación y Tecnologías de la Información

Tema 7. Caso de Estudio

Integración de Aplicaciones

Índice

- Introducción
  - Descripción del caso de estudio
  - Diseño por capas
  - Protocolo REST
  - Arquitectura
- Implementación de la Capa Servicio con Jakarta Restful Web Services (JAX-RS)
- Implementación de la Capa Acceso al Servicio

Introducción

- Implementación de un servicio y un cliente REST
  - Utilizando diseño por capas
  - El protocolo REST (información a intercambiar entre cliente y servidor en XML) está prestablecido
  - Utilizando JAX-RS para implementar el servidor
  - Utilizando la API cliente definida por JAX-RS para implementar la Capa Acceso al Servicio
  - Tanto en el cliente como en el servidor se utilizará JAXB
    - No es necesario parsear y generar “manualmente” XML con JDOM (u otra API)
    - Permitirá intercambiar la información en formato JSON además de XML

Descripción del Caso de Estudio (1)

- Retomamos el ejemplo visto en ISD
  - La Capa Modelo desarrollada se expuso como servicio web usando REST (implementación con Servlets) y RPC (Thrift / SOAP)

  ➔ Ahora la expondremos como REST
    - Implementándola con JAX-RS y JAXB
    - Siguiendo el mismo protocolo REST que el definido para la implementación con Servlets
  - Se implementó la Capa Acceso al Servicio para acceder a los servicios REST (con HTTPClient) y Thrift

  ➔ Ahora proporcionaremos una nueva implementación para acceder al servicio REST utilizando la API cliente de JAX-RS

  - Dado que las dos implementaciones REST del servicio siguen el mismo protocolo, se puede acceder a cualquiera de ellas con cualquiera de las dos implementaciones REST de la Capa Acceso al Servicio

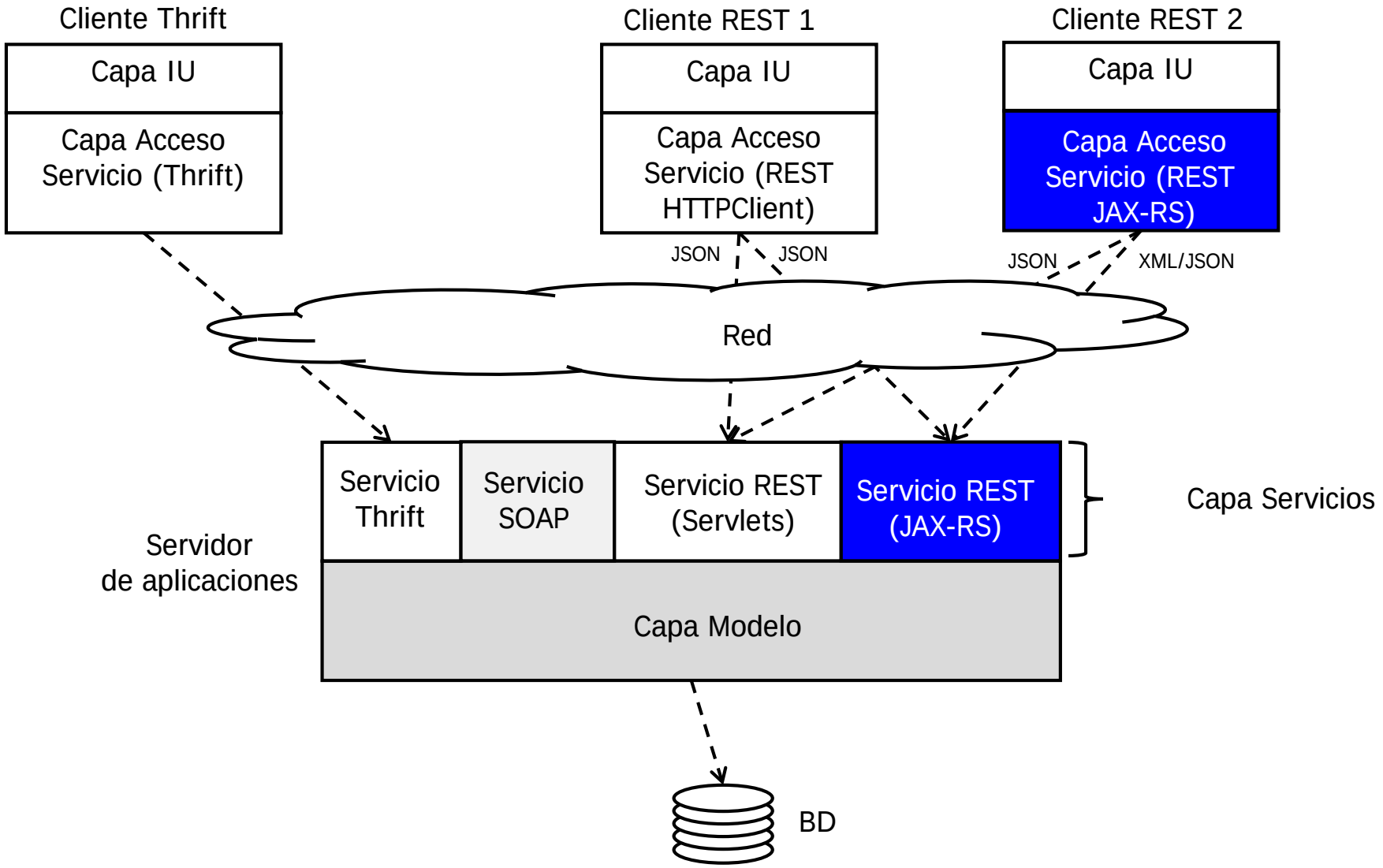
Descripción del Caso de Estudio (2)

- El cliente es una sencilla aplicación de línea de comandos
  - Puede configurarse (sin necesidad de recompilación) para usar la Capa Acceso al Servicio deseada

Descripción del Caso de Estudio (y 3)

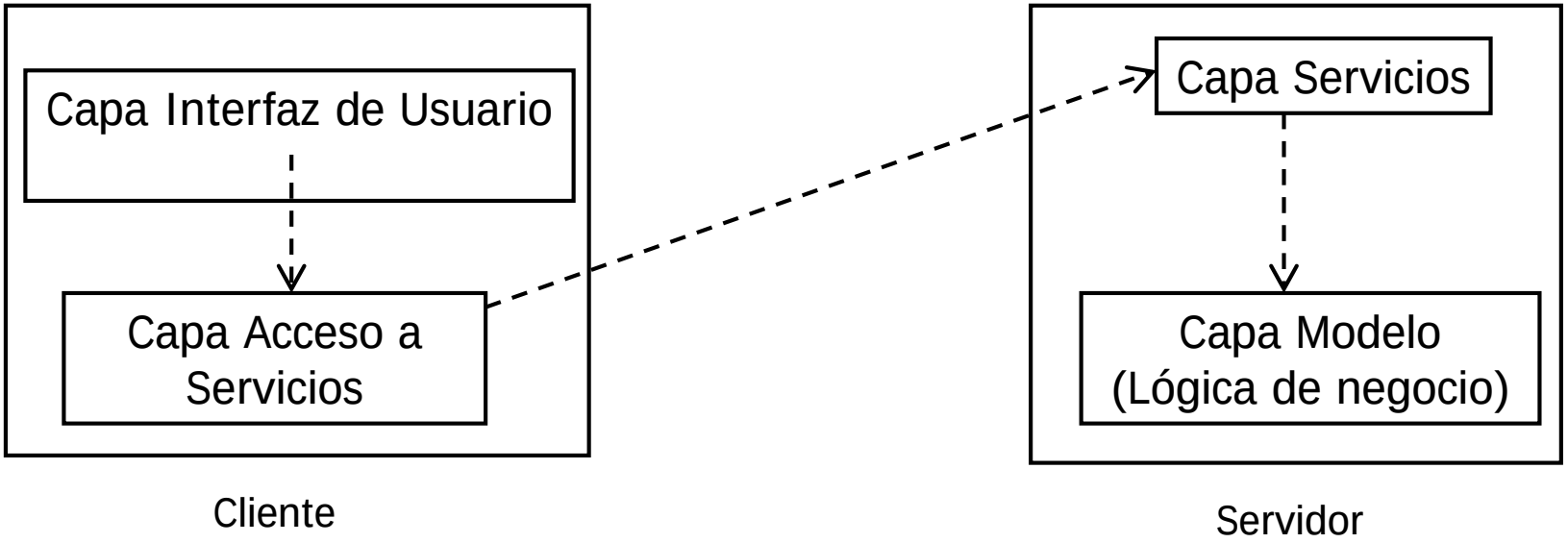
- Cliente **`MovieServiceClient`**
  - Añadir película
    - **`MovieServiceClient -a <title> <hours> <minutes> <description> <price>`**
  - Borrar película
    - **`MovieServiceClient -r <movieId>`**
  - Actualizar película
    - **`MovieServiceClient -u <movieId> <title> <hours> <minutes> <description> <price>`**
  - Buscar películas por palabras clave en el título
    - **`MovieServiceClient -f <keywords>`**
  - Comprar película
    - **`MovieServiceClient -b <movieId> <userId> <creditCardNumber>`**
  - Ver película (recuperar venta)
    - **`MovieServiceClient -g <saleId>`**

Arquitectura General de Movies



Diseño por Capas (1)

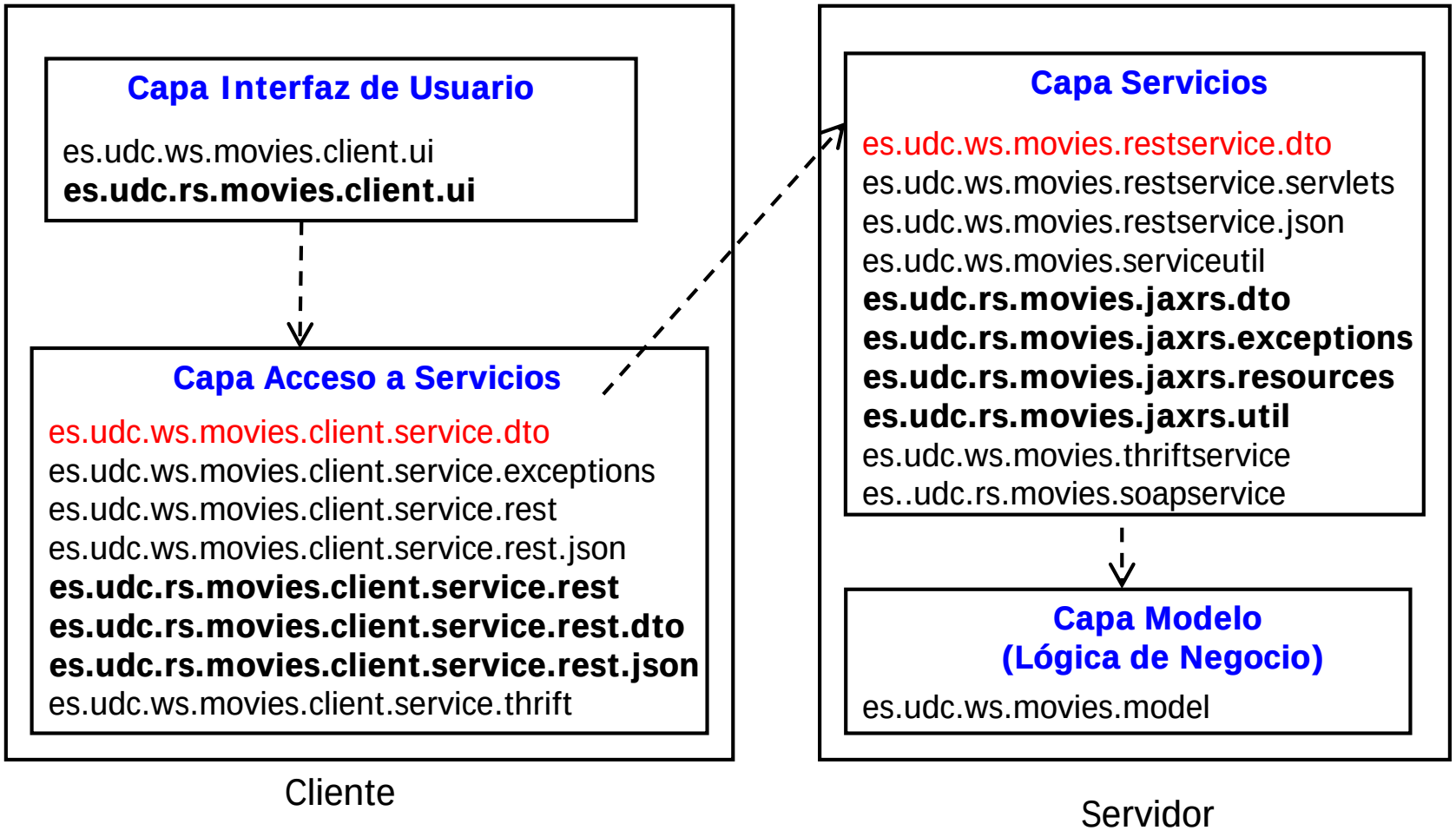
- En el caso de estudio usaremos el diseño por capas para ocultar las tecnologías de acceso e implementación de servicios



Diseño por Capas (y 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 al servicio
  - Ofrece una API que oculta la tecnología usada para acceder al servicio
- Capa **Servicios**
  - Utiliza una tecnología para implementar el servicio y delega en la capa modelo
- Capa **Modelo (Lógica de Negocio)**
  - Implementa la lógica que expone el servicio
  - No depende de la tecnología de implementación del servicio
  - Ofrece una API que oculta la tecnología usada para implementar la lógica de negocio

Principales Paquetes por Capas



Protocolo REST (1)

- Seguimos la misma aproximación que la mayoría de servicios web REST “recientes”
  - URLs únicos 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 usamos hipermedia ni representaciones autodescriptivas

  - NOTA
    - A diferencia de lo que hacíamos en ISD, aquí utilizamos en primer lugar una representación XML para los recursos
    - Más adelante veremos cómo hacer el cambio para admitir también representación JSON

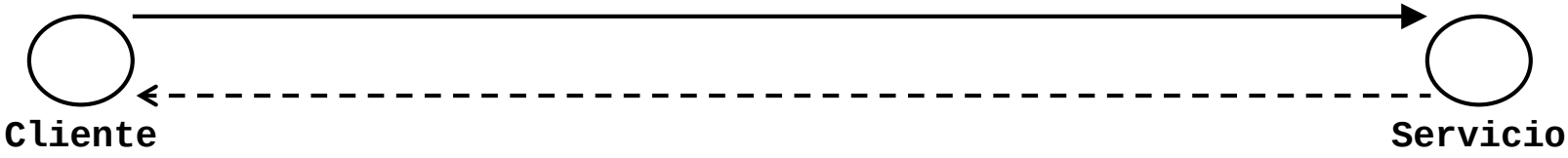
Protocolo REST (2)

- Recursos:
  - **`/movies`** Recurso colección
    - **`GET`**
      - Lista todas las películas
      - La información de cada película se representa en XML
      - El parámetro **`keywords`** permite filtrarlas por palabras clave en el título: **`/movies?keywords=Star+Wars`**
    - **`POST`**
      - Añade una nueva película
      - La película se envía en el cuerpo de la petición en formato XML
      - Devuelve el código de respuesta HTTP: 201 Created
      - Devuelve el 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 información de películas

Petición GET a http://XXX/rs-movies-service/movies?keywords=Star+Wars



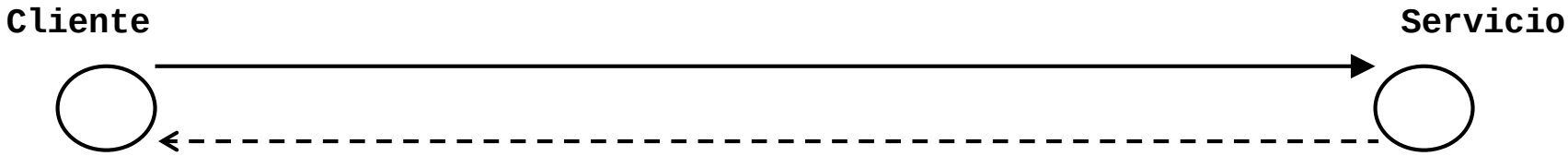

```
HTTP/1.1 200 OK
…

<?xml version="1.0" encoding="UTF-8"?>
<movieDtoJaxbs>
    <movie xmlns="http://ws.udc.es/movies/xml" movieId="3">
        <title>Star Wars: Episode IV - A New Hope</title>
        ...
    </movie>
    <movie xmlns="http://ws.udc.es/movies/xml" movieId="5">
        <title>Star Wars: Episode V - The Empire Strikes Back</title>
        ...
    </movie>
</movieDtoJaxbs>
```

Protocolo REST (4)

- Añadir la información de una película

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

```
<?xml version="1.0" encoding="UTF-8"?>
<movie xmlns="http://ws.udc.es/movies/xml">
    <title>Star Wars: Episode IV - A New Hope</title>
    <runtime>121</runtime>
    ...
</movie>
```




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

<?xml version="1.0" encoding="UTF-8"?>
<movie xmlns="http://ws.udc.es/movies/xml" movieId="3">
    <title>Star Wars: Episode IV - A New Hope</title>
    ...
</movie>
```

Protocolo REST (5)

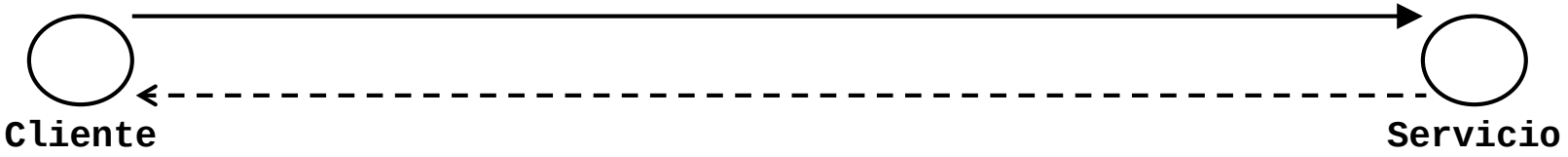
- Recursos:
  - **`/movies/{id}`** Recurso individual por película
    - **GET**
      - Obtiene la información de la película
      - La información de cada película se representa en XML

      NOTA: ws-movies-service (ISD) no expone esta funcionalidad
    - **PUT**
      - Modifica la película
      - La película se envía en el cuerpo de la petición en formato XML
      - 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)

- Obtener la información de una película

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



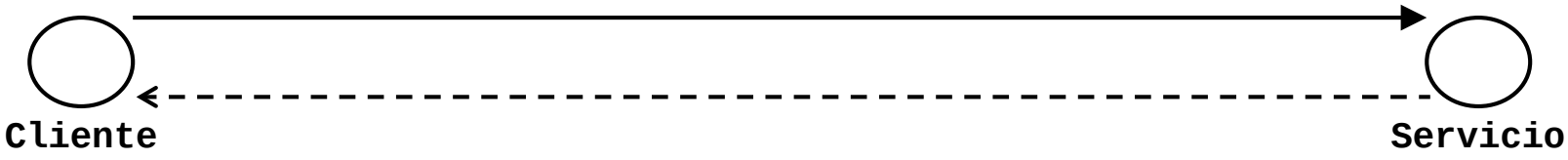

```
HTTP/1.1 200 OK
…

<?xml version="1.0" encoding="UTF-8"?>
<movie xmlns="http://ws.udc.es/movies/xml" movieId="3">
    <title>Star Wars: Episode IV - A New Hope</title>
    <runtime>121</runtime>
    <price>20.0</price>
    <description>...</description>
</movie>
```

Protocolo REST (7)

- Actualizar la información de una película

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

```
<?xml version="1.0" encoding="UTF-8"?>
<movie xmlns="http://ws.udc.es/movies/xml" movieId="3">
    <title>Star Wars: Episode IV - A New Hope</title>
    <runtime>120</runtime>
    ...
</movie>
```

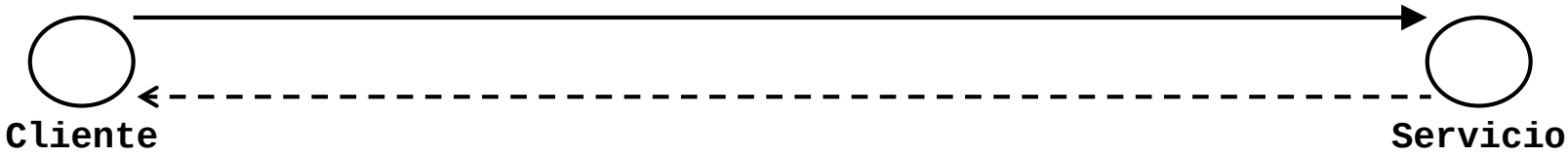



```
HTTP/1.1 204 No Content
…
```

Protocolo REST (8)

- Eliminar la información de una película

Petición DELETE a http://XXX/rs-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 el URL de la nueva película usando la cabecera estándar HTTP **`Location`**
      - El cuerpo de la respuesta devuelve los datos de la nueva venta creada en XML
  - **`/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 (10)

- Añadir la información de una venta

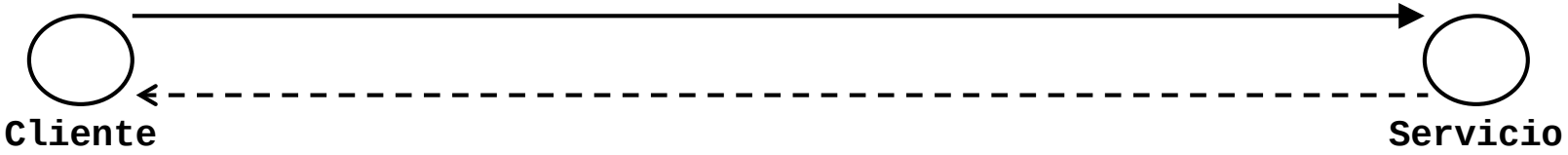
```
Petición POST a http://XXX/rs-movies-service/sales
  (En el cuerpo):
      movieId=7&userId=username&creditCardNumber=1234567890123456
```


```
HTTP/1.1 201 Created
…
Location: http://XXX/rs-movies-service/sales/15

<?xml version="1.0" encoding="UTF-8"?>
<sale xmlns="http://ws.udc.es/movies/xml" saleId="15">
    <movieId>7</movieId>
    <movieUrl>http://ws-movies.udc.es/sale/stream/...</movieUrl>
    <expirationDate>2013-08-03T12:00:00/>
</sale>
```

Protocolo REST (11)

- Obtener la información de una venta

Petición GET a http://XXX/rs-movies-service/sales/15




```
HTTP/1.1 200 OK
…

<?xml version="1.0" encoding="UTF-8"?>
<sale xmlns="http://ws.udc.es/movies/xml" saleId="15">
    <movieId>7</movieId>
    <movieUrl>http://ws-movies.udc.es/sale/stream/...</movieUrl>
    <expirationDate>2013-08-03T12:00:00/>
</sale>
```

Protocolo REST

- 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`**
  - No está permitido realizar una acción: 403 Forbidden
    - Similar a **`MovieNotRemovableException`**
  - Error interno de ejecución: 500 Internal Error
- El cuerpo del mensaje puede llevar información adicional
  - Representación en XML de los datos de la excepción

Protocolo REST (y 13)

- Ejemplo de representación para la excepción **`InstanceNotFoundException`**

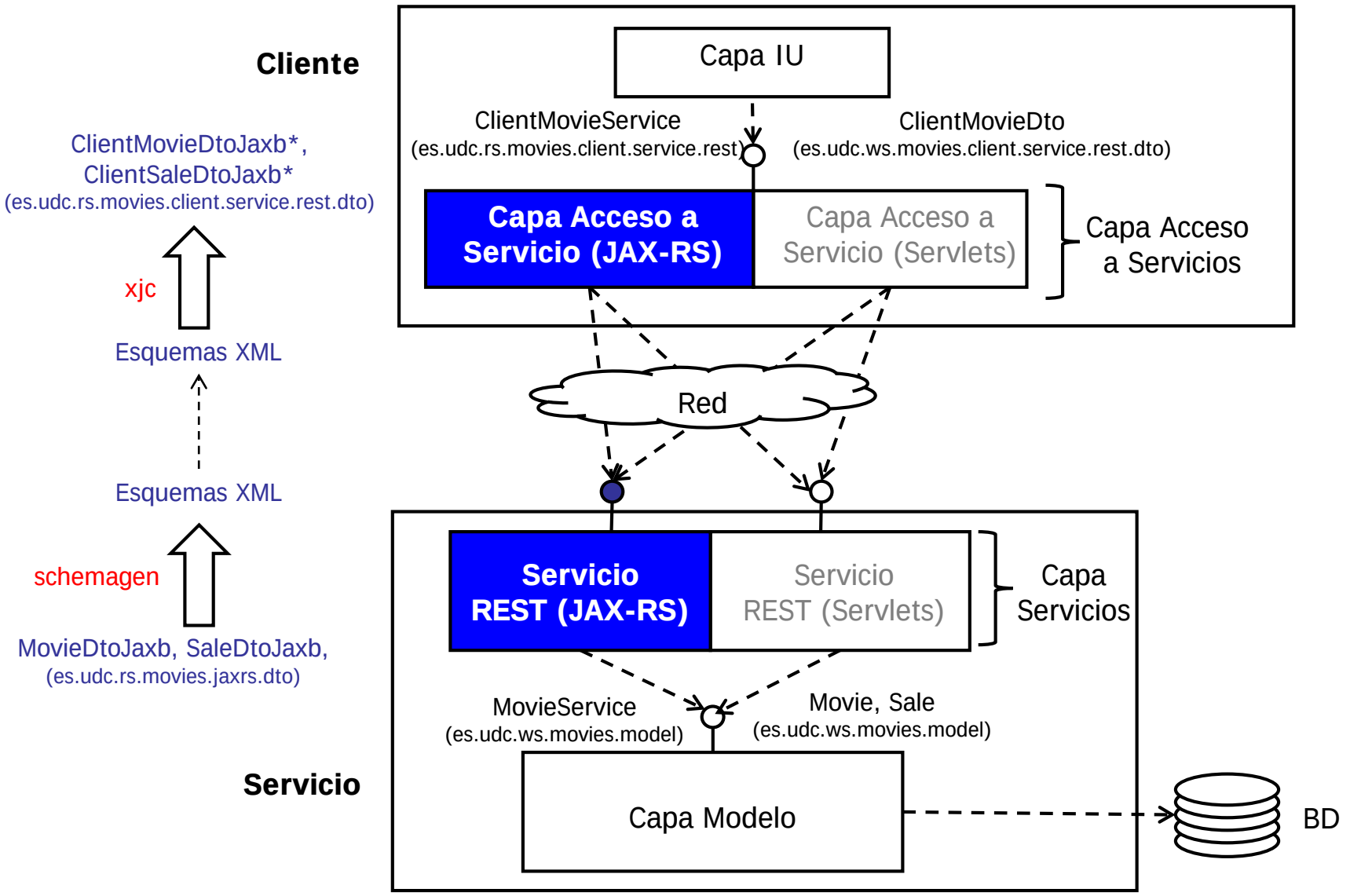
```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<instanceNotFoundException xmlns="http://ws.udc.es/movies/xml" errorType="InstanceNotFound">
    <instanceId>1234</instanceId>
    <instanceType>Movie</instanceType>
</instanceNotFoundException>
```

Consideraciones Diseño REST

- 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

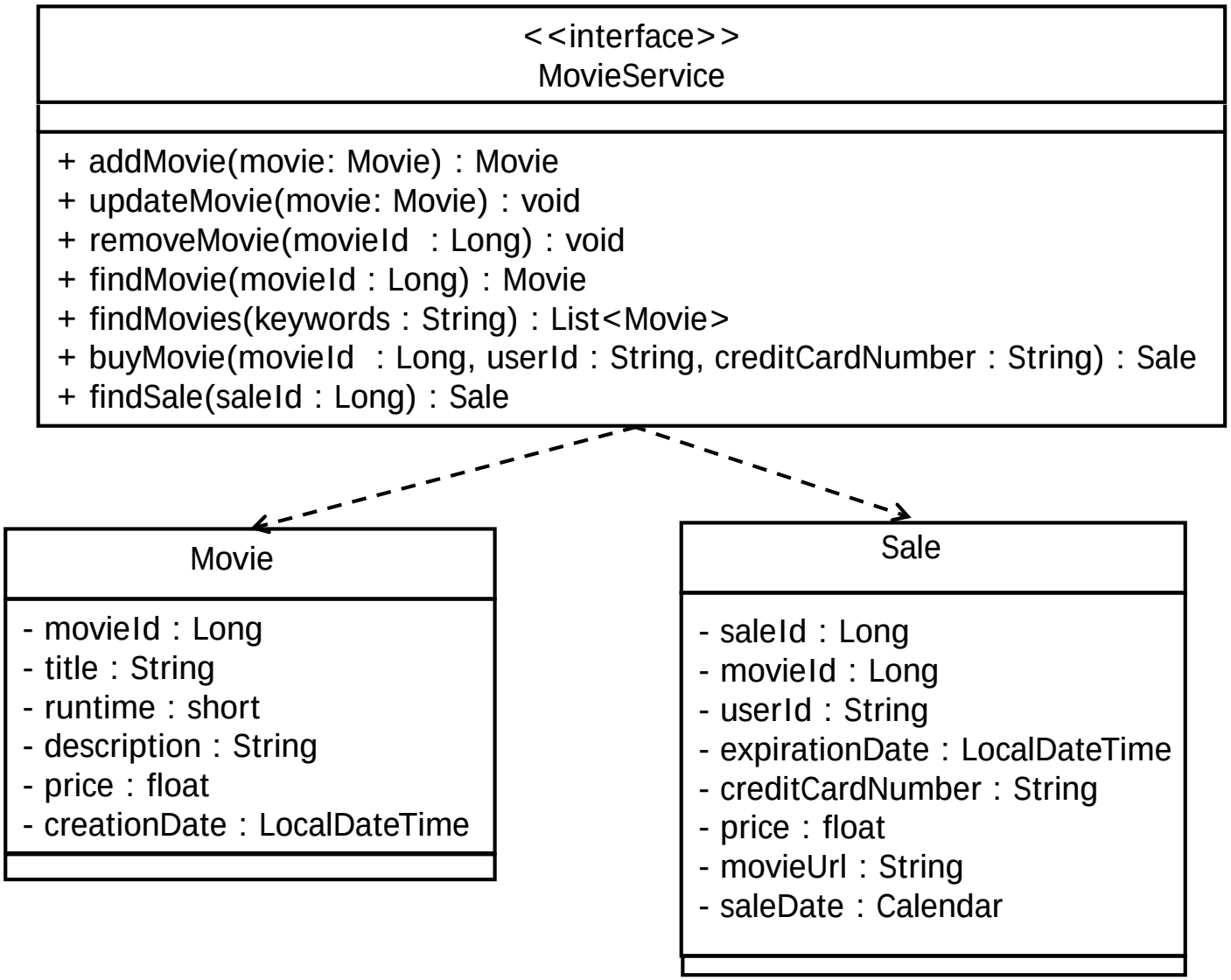
- Este diseño considera a la venta como dependiente de la película a la que se refiere
  - Representa de forma natural el que una venta siempre va asociada a una película
  - Pero no permitiría representar fácilmente consultas sobre todas las ventas (algo muy habitual)
  - Por eso, en este caso hemos considerado más adecuado el diseño anterior
    - Misma idea que la seguida en ISD

Arquitectura del Caso de Estudio



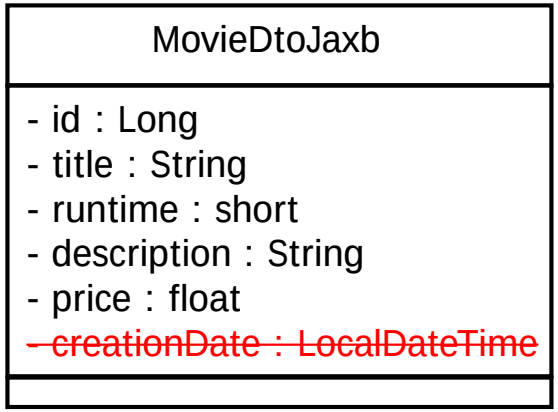
NOTA: No se incluye la Capa Servicios SOAP / Thrift, ni la capa Acceso a Servicio

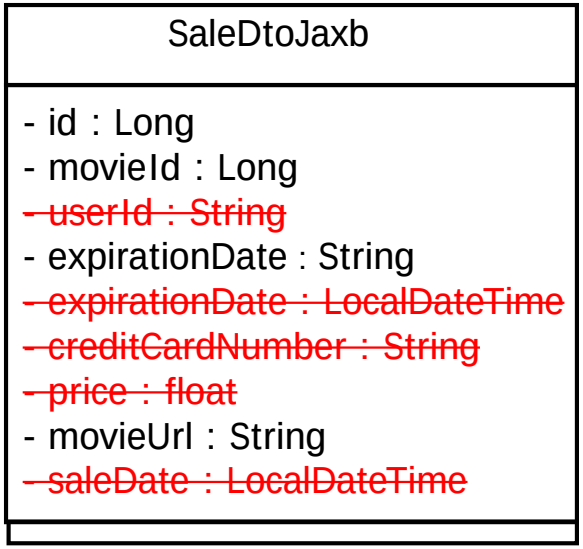
Interfaz de la Capa Modelo



Objetos de la Capa Servicio REST (JAX-RS)

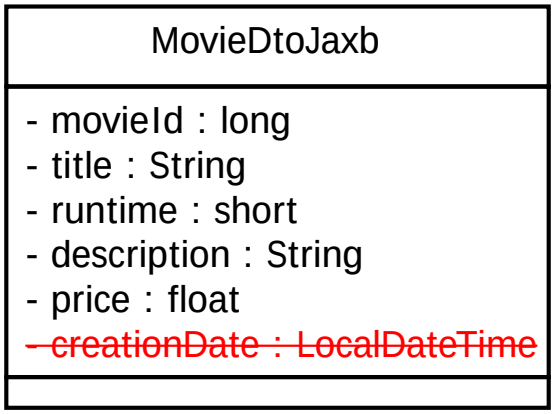
- Se utilizará JAXB para serializarlos a XML
  - La capa servicios utiliza DTOs adicionales
    - Es necesario anotarlos convenientemente
    - Dependientes de la tecnología

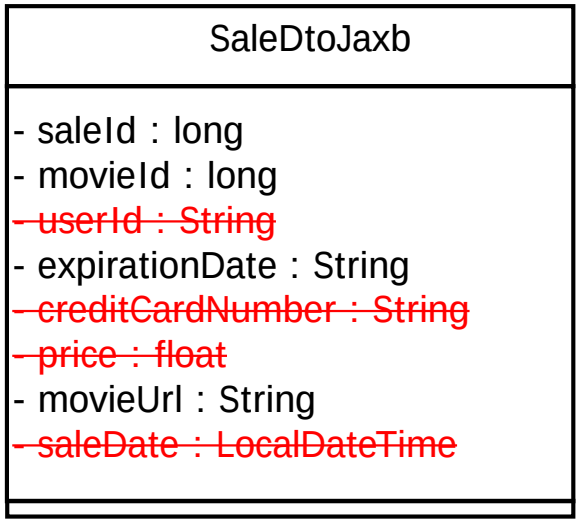




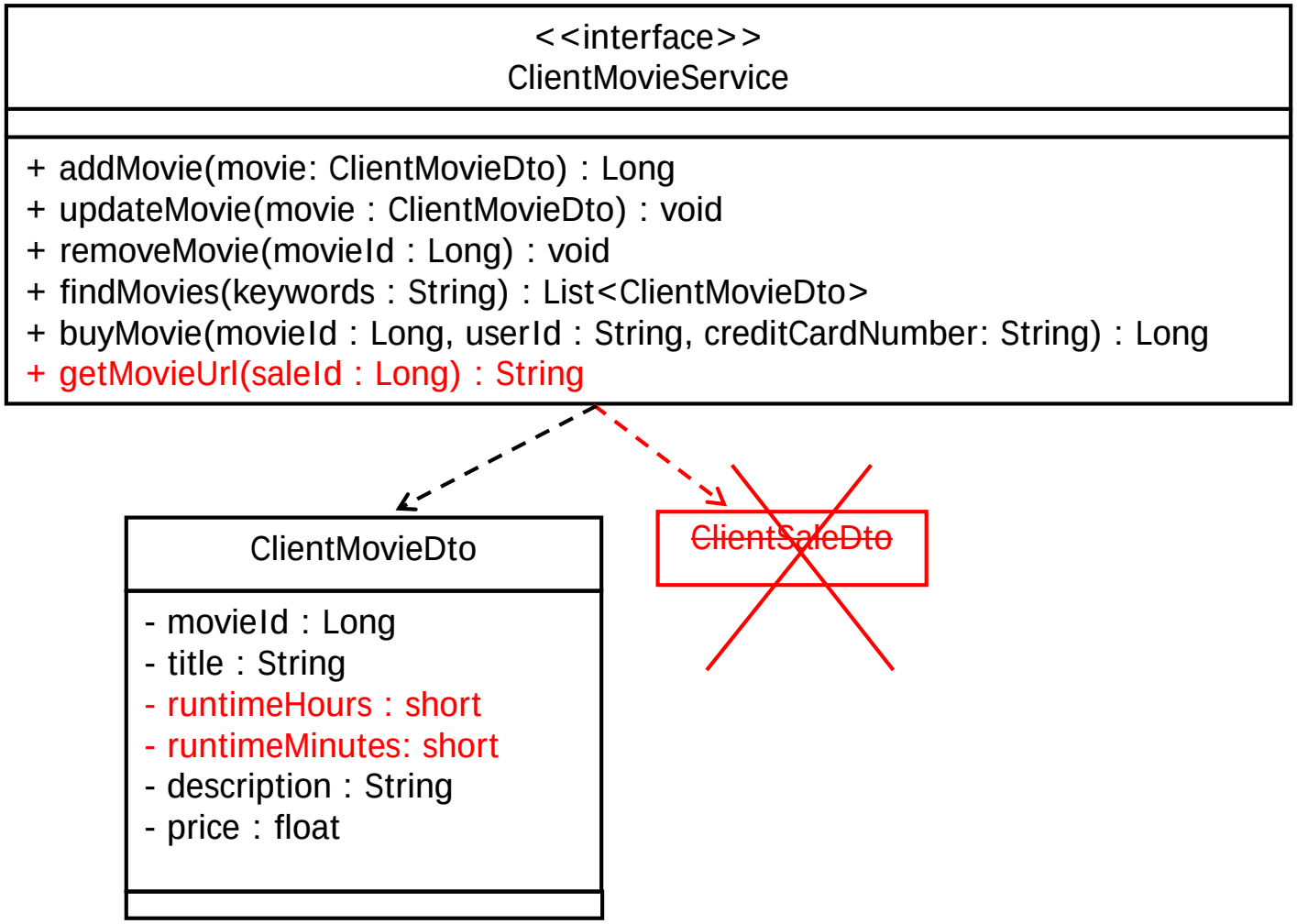
Objetos de la Capa Acceso a Servicio REST (JAX-RS)

- Se utilizará JAXB para deserializarlos desde XML
  - Generados por el compilador de esquemas





Interfaz de la Capa Acceso al Servicio



MovieResource.java (1)

```
@Path("movies")
public class MovieResource {

  @GET
  @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
  public List<MovieDtoJaxb> findMovies (
      @QueryParam("keywords") String keywords) {

    List<Movie> movies =
      MovieServiceFactory.getService().findMovies(keywords);
    return MovieToMovieDtoJaxbConversor.toMovieDtoJaxbList(movies);

}
```

MovieResource.java (2)

```
@POST
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Response addMovie(MovieDtoJaxb movieDto, @Context UriInfo ui)
    throws InputValidationException {

  Movie movie = MovieToMovieDtoJaxbConversor.toMovie(movieDtoJaxb);
  movie = MovieServiceFactory.getService().addMovie(movie);
  MovieDtoJaxb resultMovieDto =
    MovieToMovieDtoJaxbConversor.toMovieDtoJaxb(movie);

  final String requestUri = ui.getRequestUri().toString();
  return Response.created(URI.create(requestUri +
     (requestUri.endsWith("/") ? "" : "/") + movie.getMovieId()))
       .entity(resultMovieDto).build();

}
```

MovieResource.java (3)

```
@PUT
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Path("/{id}")
public void updateMovie(MovieDtoJaxb movieDto,
    @PathParam("id") String id) throws InputValidationException,
                                       InstanceNotFoundException {
  Long movieId;
  try {
     movieId = Long.valueOf(id);
  } catch (NumberFormatException ex) {
     throw new InputValidationException("Invalid Request: "
           + "unable to parse movie id '" + id + "'");
  }
  if (!movieId.equals(movieDto.getId())) {
      throw new InputValidationException("Invalid Request: invalid movie Id '" +
         movieDto.getId() + "' for movie '" + movieId + "'");
  }
  Movie movie = MovieToMovieDtoJaxbConversor.toMovie(movieDto);
  MovieServiceFactory.getService().updateMovie(movie);
}
```

MovieResource.java (4)

```
@DELETE
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Path("/{id}")
public void deleteMovie(@PathParam("id") String id)
    throws InputValidationException, InstanceNotFoundException,
           MovieNotRemovableException {

  Long movieId;
  try {
     movieId = Long.valueOf(id);
  } catch (NumberFormatException ex) {
     throw new InputValidationException("Invalid Request: "
           + "unable to parse movie id '" + id + "'");
  }

  MovieServiceFactory.getService().removeMovie(movieId);

}
```

MovieResource.java (y 5)

```
@GET
@Path("/{id}")
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public MovieDtoJaxb findById(@PathParam("id") String id)
    throws InputValidationException, InstanceNotFoundException {

  Long movieId;
  try {
     movieId = Long.valueOf(id);
  } catch (NumberFormatException ex) {
     throw new InputValidationException("Invalid Request: "
             + "unable to parse movie id '" + id + "'");
  }

  return MovieToMovieDtoJaxbConversor.toMovieDtoJaxb(
    MovieServiceFactory.getService().findMovie(movieId));
 }

}
```

SaleResource.java (1)

```
@Path("sales")
public class SaleResource {

  @GET
  @Path("/{id}")
  @Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
  public SaleDtoJaxb findById(@PathParam("id") String id)
      throws InputValidationException, InstanceNotFoundException,
             SaleExpirationException {

    Long saleId;
    try {
       saleId = Long.valueOf(id);
    } catch (NumberFormatException ex) {
       throw new InputValidationException("Invalid Request: "
             + "unable to parse sale id '" + id + "'");
    }

    Sale sale = MovieServiceFactory.getService().findSale(saleId);
    return SaleToSaleDtoJaxbConversor.toSaleDtoJaxb(sale);

  }
```

SaleResource.java (2)

```
@POST
@Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
public Response buyMovie(
     @FormParam("movieId") String movieId,
     @FormParam("userId") String userId,
     @FormParam("creditCardNumber") String creditCardNumber,
     @Context UriInfo ui)
     throws InputValidationException, InstanceNotFoundException {

  if (movieId == null) {
    throw new InputValidationException("Invalid Request: "
           + "parameter 'movieId' is mandatory");
  }
  if (userId == null) {
    throw new InputValidationException("Invalid Request: "
          + "parameter 'userId' is mandatory");
  }
  if (creditCardNumber == null) {
    throw new InputValidationException("Invalid Request: "
          + "parameter 'creditCardNumber' is mandatory");
  }
```

SaleResource.java (y 3)

```
    Long movieIdAsLong;
    try {
       movieIdAsLong = Long.valueOf(movieId);
    } catch (NumberFormatException ex) {
       throw new InputValidationException("Invalid Request: "
             + "parameter 'movieId' is invalid '" + movieId + "'");
    }

    Sale sale = MovieServiceFactory.getService()
      .buyMovie(movieIdAsLong, userId, creditCardNumber);
    SaleDtoJaxb resultSaleDto =
      SaleToSaleDtoJaxbConversor.toSaleDtoJaxb(sale);

    String requestUri = ui.getRequestUri().toString();
    return Response.created(
        URI.create(requestUri + (requestUri.endsWith("/") ? "" : "/")
        + sale.getSaleId())).entity(resultSaleDto)
        .build();  }
}
```

es.udc.rs.movies.jaxrs.dto.package-info.java

```
@XmlSchema(namespace = "http://ws.udc.es/movies/xml",
    elementFormDefault = jakarta.xml.bind.annotation.XmlNsForm.QUALIFIED)
@XmlAccessorType(XmlAccessType.FIELD)
package es.udc.rs.movies.jaxrs.dto;
```

es.udc.rs.movies.jaxrs.dto.MovieDtoJaxb.java

```
@XmlRootElement(name = "movie")
@XmlType(name="movieType",
    propOrder = {"id", "title", "runtime", "price", "description"})
public class MovieDtoJaxb {

  @XmlAttribute(name = "movieId", required = true)
  private Long id;
  @XmlElement(required = true)
  private String title;
  @XmlElement(required = true)
  private short runtime;
  @XmlElement(required = true)
  private float price;
  @XmlElement(required = true)
  private String description;

  // Constructores, métodos get/set y método toString
  ...

}
```

es.udc.rs.movies.jaxrs.dto.SaleDtoJaxb.java

```
@XmlRootElement(name="sale")
@XmlType(name="saleType",
    propOrder = {"id", "movieId", "movieUrl","expirationDate"})
public class SaleDtoJaxb {

  @XmlAttribute(name = "saleId", required = true)
  private Long id;
  @XmlElement(required = true)
  private Long movieId;
  @XmlElement(required = true)
  private String movieUrl;
  @XmlElement(required = true)
  private String expirationDate;

  // Constructores, métodos get/set y método toString
  ...
}
```

rs-movies-service: pom.xml

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>jaxb2-maven-plugin</artifactId>
  <version>${jaxb2MavenPlugin}</version>
  <executions>
    <execution>
      <id>schemagen</id>
      <goals>
        <goal>schemagen</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <createJavaDocAnnotations>false</createJavaDocAnnotations>
    <sources>
      <source>src/main/java/es/udc/rs/movies/jaxrs/dto</source>
    </sources>
  </configuration>
  ...
</plugin>
```

Tipos de Contenido (1)

- Dependiendo del entorno, las implementaciones de JAX-RS pueden proporcionar implementaciones de **`MessageBodyReader`** y **`MessageBodyWriter`** para serializar/deserializar objetos JAXB a/desde XML
  - Por tanto es posible que los métodos de una clase recurso reciban como parámetro o devuelvan objetos de ese tipo
  - El tipo consumido o producido por los métodos en cuestión debe ser XML
- En el caso de Jersey, existen múltiples librerías que se pueden integrar y que proporcionan implementaciones de **`MessageBodyReader`** y **`MessageBodyWriter`** para serializar/deserializar objetos JAXB a/desde JSON
  - El tipo consumido o producido por los métodos en cuestión debe ser JSON

Tipos de contenido (2)

- Jersey proporciona soporte para JSON a través de unos módulos de extensión que integran diferentes frameworks: MOXy, Jackson, Jettison, JSON-P, JSON-B
  - Existen diferentes representaciones JSON
  - Utilizaremos el Jackson porque simplifica la generación de JSON compatible con los ejemplos de ws-movies

Tipos de contenido (3)

- Pasos para utilizar Jackson
  - Declarar en el **`pom.xml`** la dependencia con la extensión de Jersey que integra el framework Jackson

```
<dependency>
   <groupId>org.glassfish.jersey.media</groupId>
   <artifactId>jersey-media-json-jackson</artifactId>
   <version>${jersey.version}</version>
</dependency>
```

  - Registrar en cliente y servidor la clase **`JacksonFeature`**
    - En el cliente se puede hacer a través del método **`register`** de la clase **`Client`**
      - **Necesario registrar provider adicional**

```
client = ClientBuilder.newClient();
client.register(JacksonFeature.class);
client.register(JaxbJsonContextResolver.class);
```

    - En el servidor es opcional, si se detecta en el classpath se utiliza de forma automática

Tipos de contenido (4)

- Pasos para utilizar Jackson (cont.)
  - Registrar en el servidor la clase **`JacksonFeature`** (opcional)
    - Se puede hacer creando una clase que extienda a **`Application`** o su hija **`ResourceConfig`** ...

```
package es.udc.rs.movies.jaxrs.config;
...
public class MoviesAppConfig extends ResourceConfig {
   public MoviesAppConfig() {
      register(JacksonFeature.class);
   }
}
```

    - ... y especificándola en el **`web.xml`** como un parámetro de inicialización del Servlet de Jersey

```
<servlet>
  <servlet-name>RESTFulServlet</servlet-name>
  <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
   ...
   <init-param>
     <param-name>jakarta.ws.rs.Application</param-name>
     <param-value>es.udc.rs.movies.jaxrs.config.MoviesAppConfig</param-value>
   </init-param>
</servlet>
```

Tipos de contenido (5)

- Pasos para utilizar Jackson (cont.)
  - En el cliente es necesario crear un Context Provider que implemente **`ContextResolver<ObjectMapper>`** para soportar elementos de JAXB

```
package es.udc.rs.movies.client.service.rest.json;
...
@Provider
public class JaxbJsonContextResolver implements ContextResolver<ObjectMapper> {

  private final ObjectMapper mapper;

  public JaxbJsonContextResolver() {
      this.mapper = new ObjectMapper();
      this.mapper.registerModule(new JaxbAnnotationModule());
      this.mapper.addMixIn(JAXBElement.class, JAXBElementMixin.class);
}

  @Override
  public ObjectMapper getContext(Class<?> objectType) {
      return mapper;
  }

}
```

Tipos de Contenido (y 6)

- Se pueden utilizar los mismos métodos para consumir/producir XML y JSON a partir de objetos JAXB
  - Se especifican ambos tipos de contenido en las anotaciones **`@Consumes`** y/o **`@Produces`**
    - Se elegirá el **`MessageBodyReader`** adecuado en función del tipo de contenido especificado en la petición (cabecera **`Content-Type`**)
    - Se elegirá el **`MessageBodyWriter`** adecuado en función del tipo de contenido especificado en la respuesta (que tendrá que ser uno de los contenidos en la cabecera **`Accept`** de la petición)
  - Cuando se devuelve una entidad es necesario especificar el tipo de contenido devuelto (a través del método **`type`** del **`ResponseBuilder`**)
    - Si no se especifica, como en nuestro caso, por defecto la respuesta se envía con el primer tipo de contenido soportado, de entre los recibidos en la cabecera **`Accept`** de la petición
  - Como veremos, nuestra Capa Acceso a Servicios únicamente especificará un tipo, que podrá ser XML o JSON

Clases JAXB (1)

- Todos los DTOs se han incluido en el mismo paquete java porque queremos que pertenezcan al mismo espacio de nombres (impuesto por el protocolo REST existente)
  - El espacio de nombres se ha especificado a nivel de paquete con la anotación **`@XmlSchema`**
  - A partir del paquete **`es.udc.rs.movies.jaxrs.dto`** se generará un esquema XML para el espacio de nombres **`http://ws.udc.es/movies/xml`**

- Las clases **`MovieToMovieDtoJaxbConversor`** y **`SaleToSaleDtoJaxbConversor`** se encargan de convertir entre los objetos del modelo y los objetos del servicio (clases JAXB)

Clases JAXB (2)

- Las clases **`MovieDtoJaxb`** y **`SaleDtoJaxb`** se han anotado con **`@XmlRootElement`** porque puede haber elementos raíz de documentos XML que tengan el tipo representado por esas clases
  - En la anotación se indica el nombre del tag raíz del documento en cada caso
- Se ha utilizado la anotación **`@XmlType`** en todas las clases para indicar el nombre del tipo complejo que se genera para cada una y el orden de los elementos en el tipo
- Se ha utilizado **`@XmlAttribute`** para modificar el nombre por defecto generado para los elementos correspondientes a la propiedad **`id`**, tanto en **`MovieDtoJaxb`** como en **`SaleDtoJaxb`** y para indicar que se serialice como atributo en lugar de elemento en la representación XML
- Se ha utilizado la anotación **`@XmlElement`** para indicar campos obligatorios, pero también se podría haber utilizado para modificar nombres de campos

Clases JAXB (3)

- **`MovieDtoJaxbList`**
  - El método **`findMovies`** de **`MovieResource`** devuelve una lista de **`MovieDtoJaxb: List<MovieDtoJaxb>`**
  - Se ha implementado de esta forma porque simplifica, en el caso de JSON, la generación de una representación compatible con ISD

```
[
    {
        "movieId": 3,
        "title": "Star Wars: Episode IV - A New Hope",
        "runtime": 121,
        "price": 20.0,
        "description": "Star Wars V - The empire strikes back"
    },
    ...
]
```

  - Podríamos haber creado una clase **`MovieDtoJaxbList`** para dar nombre al tag raíz (**`movies`**), pero la representación JSON por defecto para Jackson sería más compleja y no compatible con la de ISD

Clases JAXB (4)

```
@XmlRootElement(name="movies")
@XmlType(name="movieListType")
public class MovieDtoJaxbList {

  @XmlElement(name = "movie")
  private List<MovieDtoJaxb> movies = null;

  public MovieDtoJaxbList() {
    this.movies = new ArrayList<MovieDtoJaxb>();
  }

  // Constructores y métodos get/set
  ...

}
```

Clases JAXB (5)

- LocalDateTime
  - El objeto **`Sale`** que devuelve el modelo tiene un atributo de tipo **`LocalDateTime`** que queremos serializar como un String con un formato concreto
    - En el conversor **`Sale`** -> **`SaleDtoJaxb`** hemos realizado la transformación
    - En el tema 8 se comentará cómo se podría modelar directamente como un tipo **`LocalDateTime`**

Clases JAXB (y 6)

- Se ha configurado el **`pom.xml`** para que se generen los esquemas correspondientes a las clases JAXB
  - En este caso se genera un solo esquema (todos los elementos pertenecen al mismo espacio de nombres)
  - Se genera en **`target/generated-resources/schemagen`**, en particular con el nombre **`schema1.xsd`**
  - Será utilizado en el cliente para generar automáticamente las clases JAXB a las que poder deserializar documentos conformes a ese esquema XML

Tipos de los Parámetros

- En los métodos que reciben como parámetro el identificador de una película o de una venta (e.g. **`findById`**)
  - Lo reciben como un parámetro de tipo **`String`**
    - Podría haberse declarado como de tipo **`Long`** para que se hiciese la conversión de tipos automáticamente
      - Se devolvería un error 404 si falla la conversión de tipos
  - En el URI no se ha especificado ninguna expresión regular que deba seguir
    - Podría haberse declarado una expresión regular para indicar que el valor debe ser una secuencia de dígitos
      - Se devolvería un error 404 si el valor no encaja con la expresión
- Razón: Cuando el valor del parámetro es inválido (no se puede convertir a **`Long`**) se quiere devolver un error 400
  - Por compatibilidad con el protocolo REST (implementación con Servlets)

@FormParam

- El método **buyMovie** de **SalesResource** ilustra el uso de la anotación **@FormParam** para recibir los valores de parámetros codificados en el cuerpo de una petición POST como parámetros del método
  - Se esperan recibir 3 parámetros con nombre
    - **movieId**
    - **userId**
    - **creditCardNumber**
  - En caso de que el parámetro no esté incluido en el cuerpo de la petición, tomará valor **null**

SaleExpirationExceptionMapper.java (1)

```
@Provider
public class SaleExpirationExceptionMapper implements
    ExceptionMapper<SaleExpirationException> {

  @Override
  public Response toResponse(SaleExpirationException ex) {
    return Response
      .status(Response.Status.GONE)
      .entity(new SaleExpirationExceptionDtoJaxb(ex.getSaleId(),
              ex.getExpirationDate().toString())).build();
  }

}
```

SaleExpirationExceptionDtoJaxb.java (1)

```
@XmlRootElement(name="saleExpirationException")
@XmlType(name="saleExpirationExceptionType",
    propOrder = {"saleId", "expirationDate"})
public class SaleExpirationExceptionDtoJaxb {

  @XmlAttribute(required = true)
  private String errorType;
  @XmlElement(required = true)
  private String saleId;
  @XmlElement(required = true)
  private String expirationDate;

  public SaleExpirationExceptionDtoJaxb() {
  }
```

SaleExpirationExceptionDtoJaxb.java (y 2)

```
  public SaleExpirationExceptionDtoJaxb(Long saleId, String expirationDate) {
      this.errorType = "SaleExpiration";
      if (saleId != null) {
           this.saleId = saleId.toString();
      }
      this.expirationDate = expirationDate;
  }

  // Método get y set
  ...
}
```

Tratamiento de Excepciones (1)

- Se utilizan Exception Mapping providers para enviar una respuesta de error al cliente cuando se lanza alguna de las siguientes excepciones
  - **`InstanceNotFoundException`**
    - **`InstanceNotFoundExceptionMapper`**
      - Código 404 más cuerpo con información de la excepción
      - Esta excepción es lanzada por la Capa Modelo
  - **`InputValidationException`**
    - **`InputValidationExceptionMapper`**
      - Código 400 más cuerpo con información de la excepción
      - Esta excepción a veces es creada y lanzada desde la propia implementación de los métodos de las clases recurso, y otras veces es lanzada por la Capa Modelo
  - **`SaleExpirationException`**
    - **`SaleExpirationExceptionMapper`**
      - Código 410 más cuerpo con información de la excepción
      - Esta excepción es lanzada por la Capa Modelo

Tratamiento de Excepciones (2)

- Se utilizan Exception Mapping providers para enviar una respuesta de error al cliente cuando se lanza alguna de las siguientes excepciones (cont.)
  - **`MovieNotRemovableException`**
    - **`MovieNotRemovableExceptionMapper`**
      - Código 403 más cuerpo con información de la excepción
      - Esta excepción es lanzada por la Capa Modelo

Tratamiento de Excepciones (3)

- Cuando se genera la respuesta de error
  - Especificamos un objeto JAXB como la entidad a enviar
    - Las excepciones que estamos usando están en paquetes comunes
      - Compartidos por todas las implementaciones y capas
      - Incluso podrían utilizarse en otras aplicaciones (las definidas en **`ws-util`**)
    - Se han creado clases JAXB que representan los datos (de las excepciones) que se quieren enviar en las respuestas de error
      - En los Exception Mapping Providers se crean instancias de estas clases a partir de los datos de la excepción correspondiente
      - Contienen propiedades con tipos acordes a lo que se quiere enviar
        - » E.g. **`SaleExpirationExceptionDtoJaxb`** almacena como un **`String`** la fecha encapsulada en una **`SaleExpirationException`** como una propiedad de tipo **`LocalDateTime`** (la fecha se convierte a cadena utilizando el método toString de LocalDateTime)
        - » Se ha añadido una propiedad adicional **`errorType`** para añadir de forma sencilla la información del tipo de error en representaciones JSON (tb en XML aunque redundante)

Tratamiento de Excepciones (4)

- Clases JAXB
  - **`InputValidationExceptionDtoJaxb`**, **`InstanceNotFoundExceptionDtoJaxb`**, **`SaleExpirationExceptionDtoJaxb`**
  - Están anotadas con **`@XmlRootElement`** porque se envían documentos XML cuyo tag raíz tiene el tipo representado por estas clases
  - Están incluidas en el paquete **`es.udc.rs.movies.jaxrs.dto`** porque queremos que pertenezcan al espacio de nombres que representa ese paquete (**`http://ws.udc.es/movies/xml`**)
    - Impuesto por el protocolo REST que queremos implementar

Tratamiento de Excepciones (y 5)

- Ejemplo para **`InstanceNotFound`**
  - XML

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<instanceNotFoundException xmlns="http://ws.udc.es/movies/xml"
  errorType="InstanceNotFound">
    <instanceId>1</instanceId>
    <instanceType>Movie</instanceType>
</instanceNotFoundException>
```

  - JSON

```json
{
    "instanceId": "1",
    "instanceType": "Movie",
    "errorType": "InstanceNotFound"
}
```

rs-movies-client: pom.xml

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>jaxb2-maven-plugin</artifactId>
  <version>${jaxb2MavenPlugin}</version>
  <executions>
    <execution>
      <id>xjc_movie</id>
      <goals>
        <goal>xjc</goal>
      </goals>
      <configuration>
        <sources>
          <source>
           ../rs-movies-service/target/generated-resources/schemagen/schema1.xsd</source>
        </sources>
        <xjbSources>
          <xjbSource>src/main/resources/moviesBindings.xjb</xjbSource>
        </xjbSources>
        <outputDirectory>src/main/java</outputDirectory>
        <clearOutputDir>false</clearOutputDir>
        <packageName>es.udc.rs.movies.client.service.rest.dto</packageName>
      </configuration>
    </execution>
 </executions>
  ...
</plugin>
```

moviesBindings.xjb

```
<jxb:bindings version="1.0" xmlns:jxb=http://jakarta.ee/xml/ns/jaxb
   xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <jxb:bindings schemaLocation=
      "../../../../rs-movies-service/target/generated-resources/schemagen/schema1.xsd">
     <jxb:bindings node="//xs:complexType[@name='movieType']">
        <jxb:class name="ClientMovieDtoJaxb"/>
         <jxb:bindings node=".//xs:attribute[@name='movieId']">
            <jxb:property name="id"/>
         </jxb:bindings>
     </jxb:bindings>
     <jxb:bindings node="//xs:complexType[@name='saleExpirationExceptionType']">
        <jxb:class name="ClientSaleExpirationExceptionDtoJaxb"/>
     </jxb:bindings>
     <jxb:bindings node="//xs:complexType[@name='inputValidationExceptionType']">
        <jxb:class name="ClientInputValidationExceptionDtoJaxb"/>
     </jxb:bindings>
     ...
     <jxb:bindings node="//xs:complexType[@name='saleType']">
        <jxb:class name="ClientSaleDtoJaxb"/>
         <jxb:bindings node=".//xs:attribute[@name='saleId']">
            <jxb:property name="id"/>
         ...
```

JAXB

- Se ha configurado el pom.xml para que se generen las clases JAXB a partir los esquemas creados en el servidor
  - Los esquemas se cogen directamente del subsistema **`rs-movies-service`**
    - **`../rs-movies-service/target/generated-resources/schemagen/`**
  - Las clases de cada esquema se generan en un paquete (en este caso, todas en el mismo)
    - **`schema1.xsd`** ➔ **`es.udc.rs.movies.client.service.rest.dto`**
  - Se utilizan ficheros de bindings para personalizar ciertos aspectos de las clases generadas
    - Nombres de clases y atributos

RestClientMovieService (1)

```
public abstract class RestClientMovieService implements ClientMovieService {
  private static Client client = null;
  private final static String ENDPOINT_ADDRESS_PARAMETER =
      "RestClientMovieService.endpointAddress";
  private WebTarget endPointWebTarget = null;

  private static Client getClient() {
    if (client == null) {
      client = ClientBuilder.newClient();
      client.register(JacksonFeature.class);
      client.register(JaxbJsonContextResolver.class);
    }
    return client;
  }

  private WebTarget getEndpointWebTarget() {
    if (endPointWebTarget == null) {
      endPointWebTarget = getClient().target(
        ConfigurationParametersManager.getParameter(ENDPOINT_ADDRESS_PARAMETER));
    }
    return endPointWebTarget;
  }
protected abstract MediaType getMediaType();
```

RestClientMovieService (2)

```
@Override
public Long addMovie(ClientMovieDto movie) throws InputValidationException {
  WebTarget wt = getEndpointWebTarget().path("movies");
  Response response = wt
    .request()
    .accept(this.getMediaType())
    .post(Entity.entity(MovieDtoToMovieDtoJaxbConversor.toJaxbMovie(movie),
        this.getMediaType()));
  try {
    validateResponse(Response.Status.CREATED.getStatusCode(), response);
    ClientMovieDtoJaxb resultMovie =
      response.readEntity(ClientMovieDtoJaxb.class);
    return resultMovie.getId();
  } catch (InputValidationException ex) {
      throw ex;
  } catch (Exception ex) {
      throw new RuntimeException(ex);
  } finally {
      if (response != null) {
        response.close();
      }
  }
}
```

RestClientMovieService (3)

```
@Override
public List<ClientMovieDto> findMovies(String keywords) {
  WebTarget wt = getEndpointWebTarget().path("movies")
    .queryParam("keywords", keywords);
  Response response = wt.request()
    .accept(this.getMediaType())
    .get();
  try {
    validateResponse(Response.Status.OK.getStatusCode(), response);
    List<ClientMovieDtoJaxb> movies =
        response.readEntity(new GenericType<List<ClientMovieDtoJaxb>>(){});
    return MovieDtoToMovieDtoJaxbConversor.toMovieDtos(movies);
  } catch (Exception ex) {
      throw new RuntimeException(ex);
  } finally {
      if (response != null) {
        response.close();
      }
  }
}
```

RestClientMovieService (4)

```
@Override
public Long buyMovie(Long movieId, String userId, String creditCardNumber)
  throws InstanceNotFoundException, InputValidationException {
  WebTarget wt = getEndpointWebTarget().path("sales");
  Form form = new Form();
  form.param("movieId", Long.toString(movieId));
  form.param("userId", userId);
  form.param("creditCardNumber", creditCardNumber);
  Response response = wt.request()
    .accept(this.getMediaType())
    .post(Entity.form(form));
  try {
    validateResponse(Response.Status.CREATED.getStatusCode(), response);
    ClientSaleDtoJaxb sale = response.readEntity(ClientSaleDtoJaxb.class);
    return sale.getId();
  } catch (InputValidationException | InstanceNotFoundException ex) {
      throw ex;
  } catch (Exception ex) {
      throw new RuntimeException(ex);
  } finally {
     if (response != null) {
       response.close();
     }
  }
}
```

RestClientMovieService (5)

```
@Override
public void updateMovie(ClientMovieDto movie)
    throws InputValidationException,
           InstanceNotFoundException { ... }

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

@Override
public String getMovieUrl(Long saleId)
    throws InstanceNotFoundException,
           ClientSaleExpirationException { ... }
```

RestClientMovieService (6)

```
private void validateResponse(int expectedStatusCode, Response response)
  throws InstanceNotFoundException, ClientSaleExpirationException,
         InputValidationException, ClientMovieNotRemovableException {

  Response.Status statusCode = Response.Status
    .fromStatusCode(response.getStatus());

  if (statusCode.getStatusCode() != expectedStatusCode) { return; }

  String contentType = response.getMediaType() != null ?
    response.getMediaType().toString() : null;
  boolean expectedContentType = this.getMediaType().toString()
    .equalsIgnoreCase(contentType);
  if (!expectedContentType && statusCode != Response.Status.NO_CONTENT) {
    throw new RuntimeException("HTTP error; status code = " + statusCode);
  }

  switch (statusCode) {
  case NOT_FOUND: {
    ClientInstanceNotFoundExceptionDtoJaxb exDto =
      response.readEntity(ClientInstanceNotFoundExceptionDtoJaxb.class);
      throw JaxbExceptionConversor.toInstanceNotFoundException(exDto);
```

RestClientMovieService (y 7)

```
      case BAD_REQUEST: {
        ClientInputValidationExceptionDtoJaxb exDto =
          response.readEntity(ClientInputValidationExceptionDtoJaxb.class);
          throw JaxbExceptionConversor.toInputValidationException(exDto);
      }
      case GONE: {
        ClientSaleExpirationExceptionDtoJaxb exDto =
          response.readEntity(ClientSaleExpirationExceptionDtoJaxb.class);
          throw JaxbExceptionConversor.toSaleExpirationException(exDto);
      }
      case FORBIDDEN: {
        ClientMovieNotRemovableExceptionDtoJaxb exDto =
          response.readEntity(ClientMovieNotRemovableExceptionDtoJaxb.class);
          throw JaxbExceptionConversor.toMovieNotRemovableException(exDto);
      }
      default:
        throw new RuntimeException("HTTP error; status code = "+ statusCode);
      }
    }
}
```

Comentarios (1)

- Al objeto **`Client`** solamente se le especifica un Context provider
  - La API cliente de JAX-RS proporciona las mismas implementaciones de **`MessageBodyReader`** y **`MessageBodyWriter`** que el servidor, para serializar/deserializar objetos JAXB a/desde XML o JSON
  - La clase **`JaxbJsonContextResolver`** es necesaria para que Jackson sea capaz de resolver correctamente elementos JAXB

- El código de **`RestClientMovieService`** es común para la Capa Acceso a Servicio que interactúa con el servicio REST en XML y en JSON
  - El método **`getMediaType`** (abstracto) es utilizado para obtener el tipo de contenido a enviar/recibir del servidor
    - Cuando se envía una petición se utiliza
      - Para especificar el tipo de contenido que acepta el cliente (método **`accept`** de **`WebTarget`**)
      - Para especificar el tipo de los datos enviados como cuerpo de la petición (argumento de **`Entity.entity`** indicado en el método usado para la petición concreta – put/post)

Comentarios (2)

- Las respuestas siempre se obtienen como objetos de tipo **`Response`**
  - Igual que vimos en el tutorial de JAX-RS

- Las entidades especificadas como cuerpo de las peticiones y las obtenidas de los cuerpos de las respuestas son instancias de las clases JAXB generadas a partir de los esquemas XML
  - Ejemplos
    - En **`addMovie`** se especifica como entidad a enviar en la petición un objeto de tipo **`ClientMovieDtoJaxb`**
    - En **`findMovies`** se obtiene el cuerpo de la respuesta como un objeto de tipo **`List<ClientMovieDtoJaxb>`**
      - **`new GenericType<List<ClientMovieDtoJaxb>>(){}`**
  - La clase **`MovieDtoToMovieDtoJaxbConversor`** se encarga de realizar las conversiones entre los objetos con los que trabaja la capa cliente y los objetos con los que trabaja esta implementación de la Capa Acceso al servicio (objetos JAXB)

Comentarios (3)

- En el método **`buyMovie`** es necesario enviar parámetros como cuerpo de una petición POST
  - Se utiliza la clase **`Form`** (de la API cliente de JAX-RS)
    - Clase utilidad para leer/escribir parámetros de un formulario
    - Puede ser especificada como la entidad a enviar cuando se invoca al método post sobre un objeto de tipo **`Invocation.Builder`**
    - Mediante el método **`param`** se añaden pares
      - Nombre de parámetro
      - Valor del parámetro
  - Para crear la entidad a enviar se ha usado el método **`Entity.form(xx)`**
    - Alternativamente, se podría haber utilizado
      - **`Entity.entity(xx, "application/x-www-form-urlencoded")`**

Comentarios (4)

- El método **`validate`**
  - Recibe el código de respuesta esperado y el objeto **`Response`** con la respuesta recibida
  - Comprueba si el código de respuesta es el esperado y en ese caso finaliza
  - Comprueba que el tipo de contenido recibido es el esperado por el cliente (llamando al método abstracto **`getMediaType`**)
    - Si no lo es y se ha recibido un código de error se lanza una excepción de Runtime
  - Si se ha recibido un código de error 400, 404, 410 o 403
    - Sabemos que el servidor ha incluido información sobre las excepciones **`InputValidationException`**, **`InstanceNotFoundException`**, **`SaleExpirationException`** o **`MovieNotRemovableException`** respectivamente, en el cuerpo de la respuesta
    - Se obtiene el cuerpo de la respuesta como un objeto JAXB
    - Se utiliza la clase **`JaxbExceptionConversor`** para generar la excepción correspondiente a partir del objeto JAXB
  - En otro caso, se lanza una excepción de Runtime

Comentarios (y 5)

- No se muestra el código de los métodos **`updateMovie`**, **`removeMovie`** y **`getMovieUrl`**
  - Similar al de los métodos mostrados

RestClientMovieService{Xml/Json} (1)

```
public class RestClientMovieServiceXml extends RestClientMovieService {
  @Override
  protected MediaType getMediaType() {
    return MediaType.APPLICATION_XML_TYPE;
  }
}
```

```
public class RestClientMovieServiceJson extends RestClientMovieService {
  @Override
  protected MediaType getMediaType() {
    return MediaType.APPLICATION_JSON_TYPE;
  }
}
```

`RestClientMovieService{Xml/Json}` (y 2)

- Las clases **`RestClientMovieServiceXml`** y **`RestClientMovieServiceJson`** únicamente implementan el método **`getMediaType`** para indicar el tipo de contenido con el que trabajar (XML o JSON)

- En el fichero de configuración de la aplicación cliente de línea de comandos habrá que especificar una de estas dos clases, en la propiedad que indica qué clase debe instanciar la factoría que abstrae la Capa Acceso al Servicio
  - **`ClientMovieServiceFactory.className`**

`Validate`: Varias Excepciones Comparten Código de Error

- Cuando varias excepciones comparten código de error HTTP
  - No es posible intentar invocar **`readEntity`** varias veces con diferentes clases
  - Es posible utilizar JAXB directamente para distinguir la excepción concreta devuelta, cuando el documento XML / JSON incluyen un elemento raíz diferenciable

```
JAXBContext jaxbContext = JAXBContext.newInstance("...");
Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();
JAXBElement<Object> jaxbException =
  (JAXBElement<Object>) jaxbUnmarshaller.unmarshal(xmlStreamReader);
if (jaxbException.getValue() instanceof
                              InstanceNotFoundExceptionDtoJaxb) ...
```

    - Es el caso de XML pero no de JSON que por claridad no se usan envoltorios adicionales
  - La recomendación en este caso es crear una única clase para encapsular todas las excepciones con JAXB (ilustrado en tema 8)