




# Tema 5

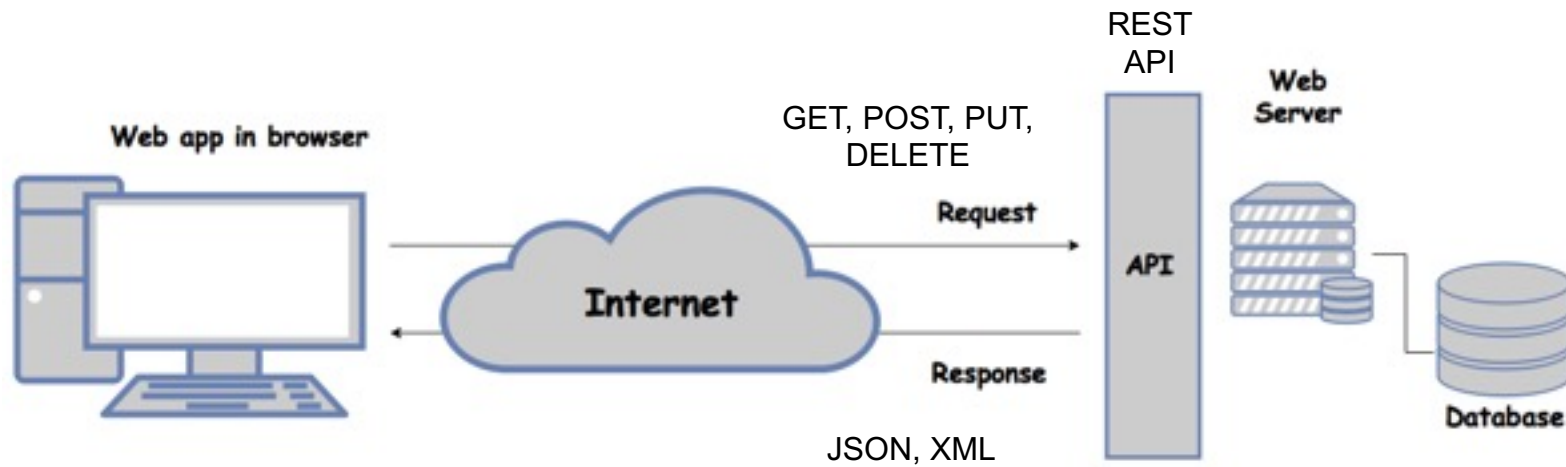
# RESTful

Plataformas de Software Empresariales  
Grado en Ingeniería Informática de Servicios y Aplicaciones  
Curso 2021 /2022

# Introducción

- ▶ Introducción REST
  - ▶ API JAX-RS
  - ▶ JAX-RS: recursos
  - ▶ JAX-RS: comunicación asíncrona
  - ▶ JAX-RS: asociación con HTTP
  - ▶ JAX-RS: representaciones de recursos
  - ▶ **JAX-RS: API cliente**
- 

# Introducción REST



# RESTful

- ▶ JAX-RS define un API estándar para crear, publicar, e **invocar** servicios REST
- ▶ Como ya hemos visto en temas anteriores, REST utiliza los estándares de la web
- ▶ Características:
  - **Todo se identifica como un recurso, y cada recurso se encuentra identificado por una URI**
    - Ej: `http://localhost:8080/servicio_factorial/webresources/factorial`
    - Ej: `http://localhost:8080/servicio_factorial/webresources/factorial?base=3`
  - Un recurso se puede representar en múltiples formatos (definidos por el “media type”)
  - Usa métodos HTTP estándar para interactuar con el recurso: GET para recuperar un recurso, POST para crearlo, PUT para actualizarlo, DELETE para eliminarlo
  - La comunicación con el servicio web es “stateless”

# JAX-RS

- ▶ JAX-RS 2 añade nuevas características al API:
  - **Define un API Cliente que se puede usar para acceder a recursos Web e integrarla con proveedores JAX-RS. Sin este API habría que acceder al servidor REST a través de los comandos de bajo nivel HttpURLConnection**
  - Añade capacidades de procesamiento asíncrono en clientes y servidores
  - Añade restricciones de validación para validar los parámetros y el tipo de retorno

# JAX-RS

## ► Situación actual de la AE Cines:

Cines Luz de Castilla - Segovia

Mostrando 20 películas en 7 páginas  
**Lista de películas disponibles**

ID	Titulo	Actores
1	The Matrix	Keanu Reeves, Laurence Fishburne, Carrie-Anne Moss
2	The Lord of The Rings	Elijah Wood, Ian McKellen, Viggo Mortensen
3	Academy	Leonardo DiCaprio
4	The Shogun	Jack Nicholson, Shelley Long
5	Wagon Impossible	Sam Grimes, Jeremy Renner
6	Terminator	Ronald Reagan, Linda Hamilton
7	Star Wars	Leonardo DiCaprio, Kate Winslet
8	Iron Man	Robert Downey Jr., Gwyneth Paltrow, Terrence Howard
9	Angustias Nostalgias	Brad Pitt, Steve Kiger
10	William Shakespeare	Hilary Swank, Clint Eastwood
11	42nd St	Uma Thurman
12	The Hunger Games	Jennifer Lawrence
13	The Hangover	Bradley Cooper, Zach Galafianis
14	Toy Story	Tom Hanks, Michael Keaton
15	Harry Potter	Daniel Radcliffe, Emma Watson
16	Avatar	Sam Worthington, Sigourney Weaver
17	Born a Free Man	André Kessler, Dan Papp, Felix Pohl
18	The Curious Case of Benjamin Button	Brad Pitt, Cate Blanchett
19	The Bourne Ultimatum	Matt Damon, Julia Stiles
20	The Hitman's Bodyguard	Steve Martin, Kevin Kline

## Proyecto cines

### Web Pages – index.html

```
....  
....  
<h:form>  
  <h1><h:outputText value="Lista de películas disponibles"/></h1>  
  <p:dataTable value="#{movieFacadeREST.findAll()}" var="item">  
    <p:column>  
    ....  
    ....
```

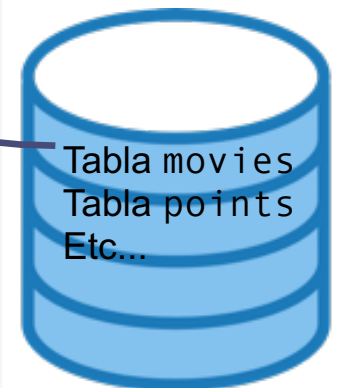
### Paquete rest

```
public class MovieFacadeREST {  
  ....  
  @GET  
  @Override  
  @Produces({...})  
  public List<Movie> findAll() {  
    return super.findAll();  
  }  
  ....  
}
```

### Paquete entities

```
Movie.java  
Points.java  
....
```

Tanto las páginas Web como los servicios rest y las entidades están dentro del mismo proyecto (cines)  
Podemos hacer esta llamada usando EL!!!



BD cines

# JAX-RS

## Proyecto cliente

### Web Pages – index.html

```
....  
<p:dataTable  
value="#{movieFacadeREST.findAll()}"  
var="item">  
....
```

Internet

## Proyecto cines

### Paquete rest

```
public class MovieFacadeREST {  
....  
    @GET  
    @Override  
    @Produces({...})  
    public List<Movie> findAll() {  
        return super.findAll();  
    }  
....  
}
```

### Paquete entities

```
Movie.java  
Points.java  
....
```

Tabla movies  
Tabla points  
Etc...

BD cines

Esta llamada no podemos hacerla, ya que el “Proyecto cliente” no tiene una clase llamada “MovieFacadeREST” (esta clase está en otro proyecto de otro servidor que puede que incluso no sea de nuestra misma empresa)  
¿Cómo accedemos a ese método del servicio REST desde el “Proyecto cliente”?

- **Mediante la URI del servicio**
- **Implementado una clase en “Proyecto cliente” que implemente el API cliente para llamar al servicio REST usando la URI**

# JAX-RS: API cliente

## Proyecto cliente

### Web Pages – index.html

```
....  
<p:dataTable value="#{cliente.findAll()}"  
var="item">  
....
```

### Paquete cliente

```
public class cliente {  
    public List<Movie> findAll() {  
        -- clase de Java que implementa el  
        API cliente y que llama a los recursos  
        REST usando la URI  
    }  
}
```



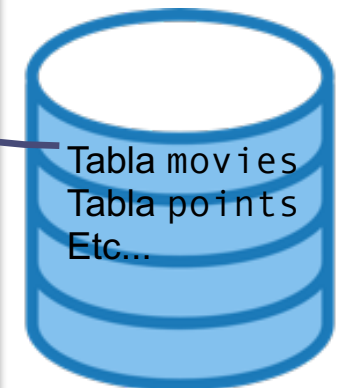
## Proyecto cines

### Paquete rest

```
public class MovieFacadeREST {  
....  
    @GET  
    @Override  
    @Produces({...})  
    public List<Movie> findAll() {  
        return super.findAll();  
    }  
....  
}
```

### Paquete entities

```
Movie.java  
Points.java  
....
```



BD cines



# JAX-RS: recursos

- ▶ Un servicio web **RESTful se puede** definir como un recurso **usando la anotación @Path**:

```
@Path("orders")
public class OrderResource {
    @GET
    public List<Order> getAll() {
        //...
    }

    @GET
    @Path("{oid}")
    public Order getOrder(@PathParam("oid") int id) {
        //...
    }
}
```

# JAX-RS: recursos

- ▶ OrderResource es publicado como recurso RESTful en la ruta “orders” (anotación @Path)
- ▶ La clase Order se marca con la anotación @XmlRootElement para permitir la conversión entre Java y XML (en el caso de los cines viene en las clases entidad)
- ▶ El método getAll proporciona una lista de todas las órdenes; se invoca cuando se accede al recurso a través del método GET de HTTP (anotación @GET)
- ▶ La anotación @Path en el método getOrder lo identifica como un subrecurso al que se accede a través de la ruta orders/{oid}
- ▶ Las llaves en oid lo identifican como un parámetro que se resolverá en tiempo de ejecución asociándolo al valor del parámetro oid en la llamada al método (mediante @PathParam)

**Importante!!** Podemos tener distintos subrecursos y maneras de pasar parámetros a nuestras llamadas. Este es uno de ellos, con @PathParam, en el que el parámetro se pasa como parte de la ruta (de la URI del servicio REST)

# JAX-RS: recursos

- ▶ También tenemos que definir la clase Application con la anotación `@ApplicationPath`, que se usan para especificar la ruta base para todos los recursos RESTful dentro del fichero de proyecto .war
  - Además de esto, la clase Application se utiliza para añadir metadatos adicionales de la aplicación
- ▶ Ej: para una aplicación (ej: **store**), la clase Application se define:

```
@javax.ws.rs.ApplicationPath("webresources")
public class ApplicationConfig extends Application {
    //...
}
```

- ▶ Por lo tanto, con los métodos anteriores se pueden acceder a todas las órdenes haciendo una petición a:

GET a `http://localhost:8080/store/webresources/orders`

- ▶ Y se puede solicitar una orden concreta como:

GET a `http://localhost:8080/store/webresources/orders/1`

Este es un parámetro  
que le estamos pasando  
en el path



# JAX-RS: recursos

- ▶ En el ejemplo anterior, el valor 1 se pasa como el parámetro id del método getOrder. El método del recurso localiza la orden con el número correcto y devuelve un objeto de la clase Order

# JAX-RS: recursos

## ► Más ejemplos:

- En la aplicación de los cines tenemos la clase Application:

```
@javax.ws.rs.ApplicationPath("webresources")  
public class ApplicationConfig extends Application {
```

- Es decir, los servicios REST van a colgar del directorio **webresources** como es habitual
- Y luego tenemos definido un servicio REST para cada entidad. Si nos fijamos en la clase MovieFacadeREST vemos como está definido el servicio REST:

```
@Named  
@Stateless  
@Path("com.anibal.cines.entities.movie")  
public class MovieFacadeREST extends AbstractFacade<Movie> {
```

- Es decir, el path al servicio REST (lo pone por defecto y lo podríamos cambiar si queremos) es **com.anibal.cines.entities.movie**

# JAX-RS: recursos

- ▶ Más ejemplos:
  - Miramos la declaración de los métodos GET:

```
@GET
@Path("/{id}")
@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public Movie find(@PathParam("id") Integer id) {
    return super.find(id);
}

@GET
@Override
@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public List<Movie> findAll() {
    return super.findAll();
}

@GET
@Path("/{from}/{to}")
@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public List<Movie> findRange(@PathParam("from") Integer from, @PathParam("to") Integer to) {
    return super.findRange(new int[]{from, to});
}

@GET
@Path("/count")
@Produces(MediaType.TEXT_PLAIN)
public String countREST() {
    return String.valueOf(super.count());
}
```

# JAX-RS: recursos

- ▶ Más ejemplos... ¿cómo serían las URIs para estos 4 GET)
  - El primer GET acepta un argumento pasado como parámetro del path (nos permite consultar una película pasándole el identificador de la película a consultar):

`localhost:8080/cines/webresources/com.anibal.cines.entities.movie/1`

- El segundo es un GET no recibe ningún argumento (lo que se hace es llamar a un método que devuelve todas las películas)

`localhost:8080/cines/webresources/com.anibal.cines.entities.movie`

- El tercer GET acepta dos argumentos pasados como parámetros del path (nos permite consultar un rango de películas)

`localhost:8080/cines/webresources/com.anibal.cines.entities.movie/1/3`

- El cuarto GET accede a la ruta “count” (fijaros que no es un valor que se le pase como pathparam porque no está entre llaves {})

`localhost:8080/cines/webresources/com.anibal.cines.entities.movie/count`

- Podéis comprobar si funciona utilizando:
  - el navegador
  - algún software como Postman

# JAX-RS: recursos

- ▶ Una URI también puede pasar parámetros query utilizando pares nombre/valor y la anotación **@QueryParam**
- ▶ Por ejemplo, si queremos que el método getAll nos muestre las órdenes que comiencen en una orden determinada de una página:

```
@GET
public List<Order> getAll(@QueryParam("start") int from,
                        @QueryParam("page") int page) {
    //...
}
```

- ▶ Para acceder al recurso se hará:

<http://localhost:8080/store/webresources/orders?start=10&page=20>

**Importante!!** Podemos tener distintos subrecursos y maneras de pasar parámetros a nuestras llamadas. Este es otro, @QueryParam, en el que el parámetro se pasa como un par clave-valor (después de la URI al recurso y separado por ?)



# JAX-RS: asociación con HTTP

- ▶ Como hemos visto se hace por medio de anotaciones:
  - GET → @GET
  - POST → @POST
  - PUT → @PUT
  - DELETE → @DELETE
- ▶ Ya hemos visto ejemplos de GET, veamos ahora un ejemplo de implementación del POST...

# JAX-RS: asociación con HTTP

- ▶ Tenemos el siguiente formulario HTML

```
<form method="post" action="webresources/orders/create">  
  Order Number: <input type="text" name="id"/><br/>  
  Customer Name: <input type="text" name="name"/><br/>  
  <input type="submit" value="Create Order"/>  
</form>
```

- ▶ El método del recurso se implementa como sigue:

```
@POST  
@Path("create")  
@Consumes("application/x-www-form-urlencoded")  
public Order createOrder(@FormParam("id") int id,  
                          @FormParam("name") String name) {  
    Order order = new Order();  
    order.setId(id);  
    order.setName(name);  
    return order;  
}
```

# JAX-RS: asociación con HTTP

- ▶ **@FormParam** hace la asociación con el valor del parámetro del form HTML
- ▶ Equivalentemente podemos hacerlo para PUT y DELETE

# JAX-RS: representaciones de recursos múltiples

- ▶ Por defecto, los recursos RESTful son publicados o consumidos con el tipo MIME \*/\*
  - MIME (Multipurpose Internet Mail Extensions) es un identificador de dos partes para formatos de archivo transmitidos por Internet.
- ▶ Podemos restringir los tipos soportados por peticiones y respuestas usando las anotaciones @Consumes y @Produces
- ▶ Ej: el método GET puede generar representaciones XML o JSON de Order

```
@GET
@Path("/{oid}")
@Produces({"application/xml", "application/json"})
public Order getOrder(@PathParam("oid") int id) {
    ...
}
```

# JAX-RS: API cliente

- ▶ Nuevo en JAX-RS 2
- ▶ Lo podemos utilizar para acceder de manera sencilla a recursos RESTful
- ▶ Ejemplo:

```
Client client = ClientBuilder.newClient();  
Order order = client  
    .target("http://localhost:8080/store/webresources/orders")  
    .path("{oid}")  
    .resolveTemplate("oid", 1)  
    .request()  
    .get(Order.class);
```

- ▶ Es la misma llamada que en el navegador hacemos como:

GET a <http://localhost:8080/store/webresources/orders/1>

# JAX-RS: API cliente

- ▶ ClientBuilder se utiliza para crear una instancia de un cliente dentro del API Cliente
  - Los clientes son objetos muy pesados, por lo que se recomienda construir el mínimo número posible de clientes en la aplicación
- ▶ Creamos el WebTarget al recurso especificando la URI de recursos
- ▶ Le indicamos el path del subrecurso al que tiene que acceder
- ▶ Y con resolveTemplate “resolvemos” el template de la URI
- ▶ Construimos la petición del cliente (request)
- ▶ Llamamos al método GET de HTTP, especificando el tipo Java de la respuesta

# JAX-RS: API cliente

- ▶ Para hacer peticiones POST (y de manera similar PUT):

```
Order order = client
    .target(...)
    .request()
    .post(Entity.entity(new Order(1), "application/json"), Order.class);
```

- ▶ En este método creamos una nueva entidad de un tipo MIME concreto, y se espera una respuesta de tipo Order

# JAX-RS: API cliente

- ▶ Para DELETE:

```
client
    .target("...")
    .path("{oid}")
    .resolveTemplate("oid", 1)
    .request()
    .delete();
```



# JAX-RS: API cliente

- ▶ En muchos casos necesitaremos registrar providers, que nos digan cómo se van a leer o escribir (procesar) las entidades (esto lo veremos en el próximo tema):

```
Order order = client
    .register(OrderReader.class)
    .register(OrderWriter.class)
    .target(...)
    .request()
    .get(Order.class);
```

- ▶ También se pueden hacer peticiones asíncronas, validación de datos, etc...

# Proyecto de desarrollo

Vamos a añadir la funcionalidad para ver información de películas y poder borrarlas

# JAX-RS

- ▶ Vamos a utilizar el API JAX-RS para:
  - Ver las películas (hacer un GET – de todos los valores, sin parámetros de ningún tipo)
  - Ver los detalles de una película (hacer un GET – para una película en concreto, para lo que habrá que usar `@PathParam`)
  - Borrar una película (hacer un DELETE – para una película en concreto, para lo que habrá que usar `@PathParam`)
  - En el tema siguiente veremos como hacer un POST para añadir más películas

# JAX-RS

- ▶ Añadir la clase MovieClientBean y el paquete client

The screenshot shows the 'New Java Class' dialog box. On the left, under 'Steps', step 2 'Name and Location' is selected. The main area is titled 'Name and Location'. The 'Class Name' field contains 'MovieClientBean' and is circled in red. The 'Project' field contains 'cines'. The 'Location' field is a dropdown menu set to 'Source Packages'. The 'Package' field contains 'com.anibal.cines.client' and is also circled in red. At the bottom, the 'Created File' path is displayed as 'cines/cines/cines/src/main/java/com/anibal/cines/client/MovieClientBean.java'. At the bottom right, there are buttons for 'Help', '< Back', 'Next >', 'Finish' (highlighted in blue), and 'Cancel'.

# JAX-RS

- ▶ Añadir las anotaciones `@Named` y `@RequestScoped`
- ▶ `@RequestScoped` permite que el bean sea automáticamente activado con cada petición (de esta manera minimizamos el uso de estos “pesados” beans)
- ▶ Al resolver los imports, asegurarnos de coger los correctos:



# JAX-RS

- ▶ Añadir el siguiente código a la clase (creamos la instancia cliente al construirse el objeto y lo cerramos al destruirse):

```
Client client;  
WebTarget target;
```

```
@PostConstruct  
public void init() {  
    client = ClientBuilder.newClient();  
    target =  
    client.target("http://localhost:8080/cines/webresources/com.anibal.cines.entities.movie");  
}
```

```
@PreDestroy  
public void destroy() {  
    client.close();  
}
```

- **Actualizad la URI por la que corresponda en vuestra práctica!!**
- Se puede comprobar si funciona utilizando:
  - el navegador
  - algún software tipo Postman

# JAX-RS

- ▶ ClientBuilder es el punto de entrada principal del API cliente que utiliza para invocar los servicios REST
- ▶ Después se crea una nueva instancia del cliente utilizando el constructor de clientes por defecto proporcionado por JAX-RS
- ▶ Los clientes son objetos pesados que manejan la infraestructura de comunicación en el lado del cliente, por lo que se recomienda crear únicamente el número necesario de instancias cliente y cerrarlas adecuadamente
- ▶ En el código anterior, la instancia cliente se crea y se destruye con los métodos del ciclo de vida (PostConstruct y PreDestroy)
- ▶ La URI se construye llamando al método “target”

# JAX-RS

- ▶ Añadir el siguiente método a la clase

```
public Movie[] getMovies() {  
    return target  
        .request()  
        .get(Movie[].class);  
}
```

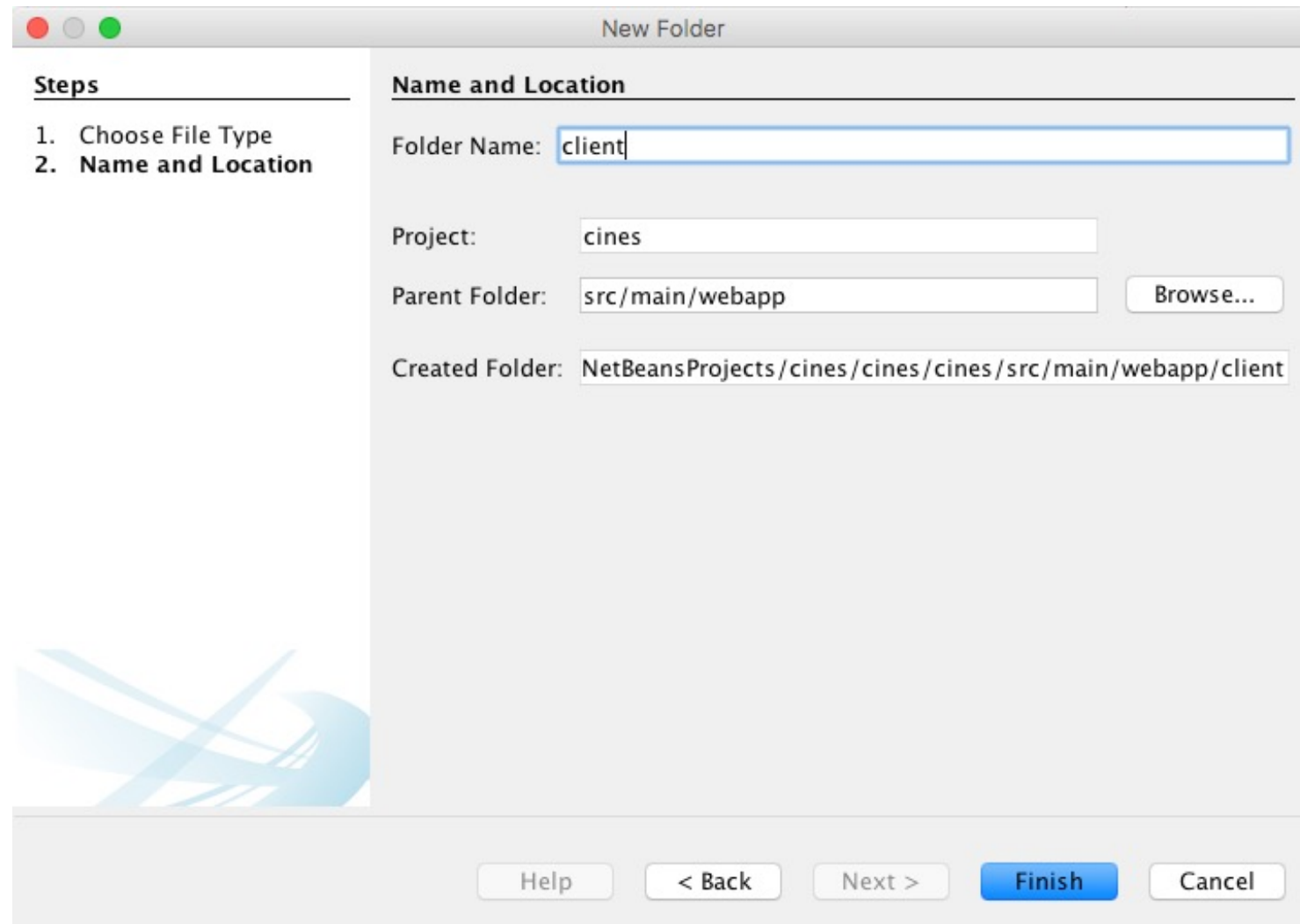
- ▶ Las peticiones REST se realizan llamando al método request
- ▶ El HTTP GET se invoca al llamar al método get
- ▶ El tipo de la respuesta se especifica en el argumento de la llamada al get, es decir, en este caso será Movie[]



# JAX-RS

- ▶ Creamos una nueva carpeta en Web Pages llamada “client”
- ▶ Creamos un nuevo “Facelets Template Client” llamado “movies”

# JAX-RS



The image shows a 'New Folder' dialog box from the NetBeans IDE. The window has a title bar with standard macOS window controls (red, yellow, green buttons) and the title 'New Folder'. On the left, a 'Steps' sidebar lists two steps: '1. Choose File Type' and '2. Name and Location', with the second step being the active one. The main area is titled 'Name and Location' and contains several input fields: 'Folder Name' with the text 'client', 'Project' with 'cines', 'Parent Folder' with 'src/main/webapp' and a 'Browse...' button, and 'Created Folder' showing the full path 'NetBeansProjects/cines/cines/cines/src/main/webapp/client'. At the bottom, there are five buttons: 'Help', '< Back', 'Next >', 'Finish' (highlighted in blue), and 'Cancel'.

**Steps**

1. Choose File Type
2. **Name and Location**

**Name and Location**

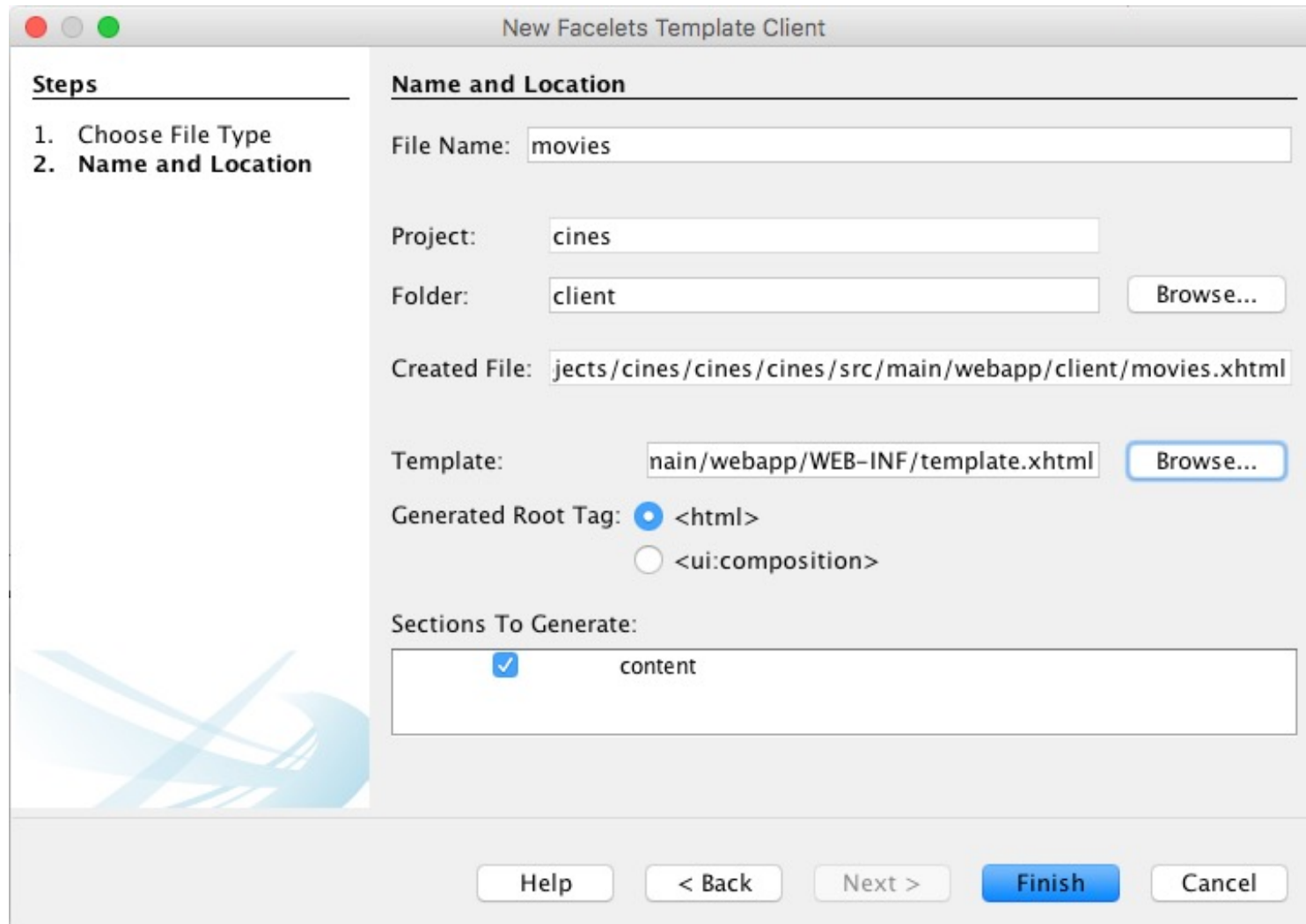
Folder Name:

Project:

Parent Folder:

Created Folder:

# JAX-RS



**New Facelets Template Client**

**Steps**

1. Choose File Type
2. **Name and Location**

**Name and Location**

File Name:

Project:

Folder:

Created File:

Template:

Generated Root Tag: ☒ `<html>`  
☐ `<ui:composition>`

Sections To Generate:

<input checked="" type="checkbox"/>	content
-------------------------------------	---------

# JAX-RS

- ▶ Añadir el siguiente código en movies.xhtml:

MovieBackingBean es la clase en la que vamos a guardar datos, como ya hemos hecho en temas anteriores

```
<h:form>
  <h:selectOneRadio value="#{movieBackingBean.movieId}"
  layout="pageDirection">
    <c:forEach items="#{movieClientBean.movies}" var="m">
      <f:selectItem itemValue="#{m.id}" itemLabel="#{m.name}"/>
    </c:forEach>
  </h:selectOneRadio>

  <p:commandButton value="Detalles" action="movie" />
</h:form>
```

La llamada, usando EL al método getMovies de la clase cliente, que es la que llama al servicio REST usando la URI del servicio

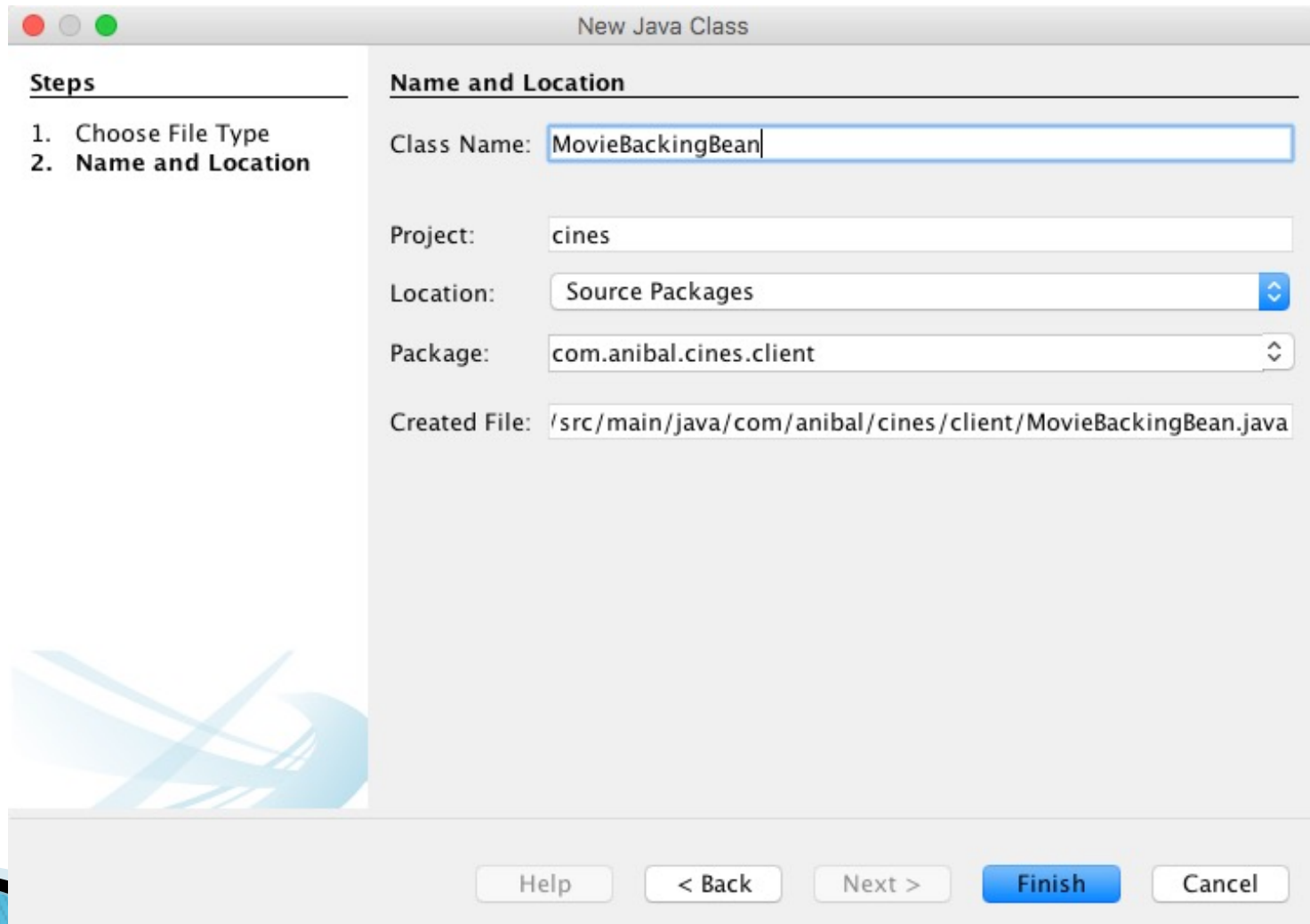
Es lo mismo que hacíamos en el Tema 4, pero allí sacábamos el listado de películas con una llamada al EL y aquí llamamos a la clase cliente REST

# JAX-RS

- ▶ Desde movies.xhtml se invoca el método getMovies en MovieClientBean
- ▶ Se itera en un bucle for sobre la respuesta y se muestra el nombre de cada película con un radio button
- ▶ El valor seleccionado el radio button se almacena en un backing bean (MovieBackingBean)
- ▶ El botón (etiquetado como detalles) nos lleva a la página movie.xhtml en el mismo directorio

# JAX-RS

- ▶ Creamos la clase para el backing bean (MovieBackingBean)



**New Java Class**

**Steps**

1. Choose File Type
2. **Name and Location**

**Name and Location**

Class Name:

Project:

Location:

Package:

Created File:

# JAX-RS

- ▶ Añadimos la variable movieId:

```
int movieId;
```

- ▶ Generamos el getter y el setter
- ▶ Añadimos las anotaciones @Named y @SessionScoped
- ▶ Implementamos la interfaz Serializable

# JAX-RS

- ▶ **Si no existiera la clase Application** (lo normal es que **Netbeans** la **debería haber creado automáticamente**), nos dirá que “REST no está configurado” y tendríamos que añadirla:



- ▶ Presionamos sobre la bombilla y seleccionamos “Configure REST using Java EE 6 specification”
- ▶ Nos habrá creado el paquete “org.netbeans.rest.application.config” con el ApplicationConfig. Podemos ver que ha creado el path de la clase Application para el servicio REST en “webresources”, que es la ruta de la que cuelgan todos nuestros servicios REST.



# JAX-RS

- ▶ Por último, añadid el link correspondiente en el menú del template para que nos redirija a `movies.xhtml` y lo probamos

# JAX-RS

## Cines Luz de Castilla - Segovia

### Menú



[Inicio](#)

[Booking](#)

[Info películas](#)

- ☐ The Matrix
- ☐ The Lord of The Rings
- ☐ Inception
- ☐ The Shining
- ☐ Mission Impossible
- ☐ Terminator
- ☐ Titanic
- ☐ Iron Man
- ☐ Inglorious Bastards
- ☐ Million Dollar Baby
- ☐ Kill Bill
- ☐ The Hunger Games
- ☐ The Hangover
- ☐ Toy Story
- ☐ Harry Potter
- ☐ Avatar
- ☐ Slumdog Millionaire
- ☐ The Curious Case of Benjamin Button
- ☐ The Bourne Ultimatum
- ☐ The Pink Panther

[Detalles](#)

# JAX-RS

- ▶ Creamos la página movie.xhtml (la que mostrará los detalles), y actualizamos su contenido de la siguiente manera:

```
<h:form>
  <h2><h:outputText value="Información de la película
#{movieClientBean.movie.name}"/></h2>
  <p:panelGrid id="info" columns="2">
    <h:outputText value="Identificador película: "/>
    <h:outputText value="#{movieClientBean.movie.id}"/>
    <h:outputText value="Título: "/>
    <h:outputText value="#{movieClientBean.movie.name}"/>
    <h:outputText value="Actores: "/>
    <h:outputText value="#{movieClientBean.movie.actors}"/>
    <h:outputText value="Resumen..."/>
    <h:outputText value=""/>
    <h:outputText value="Trailer..."/>
    <h:outputText value=""/>
    <h:outputText value="Otros..."/>
    <h:outputText value=""/>
  </p:panelGrid>

  <p:commandButton value="Back" action="movies" icon="ui-icon-arrowrefresh-1-w"/>
</h:form>
```

# JAX-RS

- ▶ Inyectamos el backing bean en el client bean para que este último pueda leer el valor de la película seleccionada:

```
@Inject  
MovieBackingBean bean;
```

- ▶ Añadimos el método que nos devuelve la información de la película (que es la que se mostrará más adelante al pulsar en “Detalles”)

```
public Movie getMovie() {  
    Movie m = target  
        .path("{movieId}")  
        .resolveTemplate("movieId", bean.getMovieId())  
        .request()  
        .get(Movie.class);  
    return m;  
}
```

# JAX-RS

- ▶ Añadimos la posibilidad de poder borrar películas
- ▶ Añadimos el botón para hacerlo:

```
<p:commandButton  
    value="Borrar"  
    action="movies"  
    actionListener="#{movieClientBean.deleteMovie()}" />
```

# JAX-RS

- ▶ Y el método para hacer el borrado en el cliente (funciona igual que el GET pero ahora estamos llamando a delete – podríamos hacer lo mismo para cualquiera de los métodos definidos en el servicio REST):

```
public void deleteMovie() {  
    target.path("{movieId}")  
        .resolveTemplate("movieId", bean.getMovieId())  
        .request()  
        .delete();  
}
```

# JAX-RS

**Cines Luz de Castilla - Segovia**

**Menú**

[Inicio](#)  
[Booking](#)  
[Info películas](#)

## Información de la película Inglorious Bastards

Identificador película:	9
Título:	Inglorious Bastards
Actores:	Brad Pitt, Diane Kruger
Resumen...	
Trailer...	
Otros...	

[Back](#) [Borrar](#)

**¿PREGUNTAS?**