



# Tema 6

# Java API para JSON

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

# Introducción

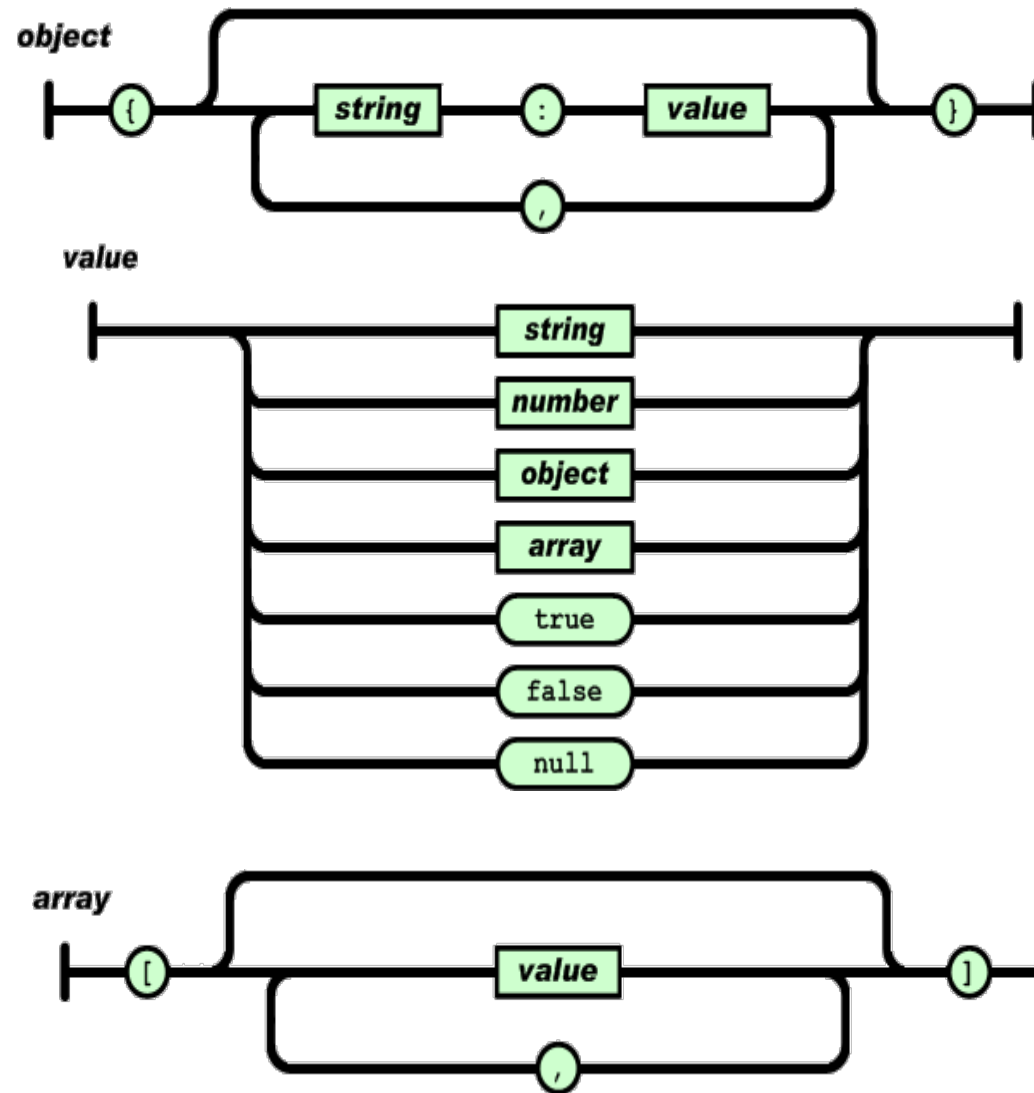
## ► Objetivos

- Aprender el funcionamiento básico del API de Java para procesamiento de JSON
- Consumir JSON
- Generar JSON
- Implementar Entity Providers de JAX-RS que usen JSON

# JSON

- ▶ El API para procesamiento de JSON es nuevo en Java EE 7 (JSR 353)
- ▶ JSON es un formato ligero para el intercambio de datos
  - Es mucho más sencillo escribir un parser para analizar JSON que XML
- ▶ **El formato hace que tanto su lectura como su escritura sean sencillos para humanos y máquinas**
- ▶ Se utiliza en entornos donde el tamaño del flujo de datos entre cliente y servidor es de vital importancia
- ▶ Una estructura JSON se puede construir de las siguientes maneras:
  - Como una colección de pares nombre/valor (donde valor puede ser un String, número, array, etc...)
  - Como una lista ordenada de valores

# JSON



# JSON

▶ Ej:

```
{  
  "name": "The Matrix",  
  "actors": [  
    "Keanu Reeves",  
    "Laurence Fishburne",  
    "Carrie-Ann Moss"  
  ],  
  "year": 1999  
}
```

- ▶ El objeto tiene tres pares nombre valor:
- El primero es “name” que contiene un string con el nombre de la película
  - El segundo es “actors” que contiene un array con los actores de la película
  - El tercero es “year” que contiene el año de lanzamiento de la película

# JSON

- ▶ JSON se está convirtiendo en la primera opción a la hora de consumir o crear servicios web
- ▶ **En Java se pueden usar multitud de librerías para implementar procesamiento de JSON, pero estas librerías tienen que incluirse en la aplicación empresarial, aumentando el tamaño del archivo desplegado**
- ▶ El API proporcionado por Java a partir de la versión 7 permite parsear y generar JSON con un API pequeña, portable y sencilla
- ▶ Podemos usar dos versiones del API
  - Streaming API
  - Object Model API

# JSON – Streaming API

- ▶ Nos proporciona los mecanismos para parsear y generar JSON como un stream
- ▶ Proporciona **un parser basado en eventos** y los métodos para parsear los distintos eventos del stream → esto nos proporciona más control sobre el procesamiento del JSON
- ▶ Es muy útil cuando tenemos que realizar procesamiento local y necesitamos acceder a partes específicas del JSON
- ▶ El API de streaming es un API de bajo nivel diseñado para procesar grandes cantidades de datos JSON de manera eficiente

# Generando JSON con el Streaming API

- ▶ El API de streaming proporciona los mecanismos para generar JSON bien formados y ponerlos en un stream de salida (escribiendo evento a evento)
- ▶ Para ello se utiliza la clase `JsonGenerator` y los distintos métodos **writeXXX** que nos proporciona

▶ Ej:

```
JsonGeneratorFactory factory = Json.createGeneratorFactory(null);  
JsonGenerator gen = factory.createGenerator(System.out);  
gen.writeStartObject().writeEnd();
```

- `JsonGenerator` se obtiene a partir de un `JsonGeneratorFactory`
- Se configura que la escritura se realice en el stream de salida (en este ejemplo `System.out`)
- Se crea un objeto JSON vacío `{ }` (`.writeStartObject()` y `.writeEnd()`)



# Generando JSON con el Streaming API

- ▶ Para generar un objeto con dos pares nombre/valor:

```
{  
  "apple":"red",  
  "banana":"yellow"  
}
```

- ▶ Pondríamos lo siguiente:

```
gen.writeStartObject()  
    .write("apple", "red")  
    .write("banana", "yellow")  
    .writeEnd();
```

- ▶ El nombre/valor se escribe usando el método write(), que por defecto toma como nombre el primer parámetro y como valor el segundo parámetro

# Generando JSON con el Streaming API

- ▶ Para un array con dos objetos JSON y cada uno de ellos con un par nombre/valor:

```
[  
  { "apple": "red" },  
  { "banana": "yellow" }  
]
```

```
gen.writeStartArray()  
  .writeStartObject()  
    .write("apple", "red")  
  .writeEnd()  
  .writeStartObject()  
    .write("banana", "yellow")  
  .writeEnd()  
.writeEnd();
```

# Generando JSON con el Streaming API

- ▶ Para una estructura anidada:

```
{  
  "name": "The Matrix",  
  "actors": [  
    "Keanu Reeves",  
    "Laurence Fishburne",  
    "Carrie-Ann Moss"  
  ],  
  "year": 1999  
}
```

```
gen.writeStartObject()  
  .write("name", "The Matrix")  
  .writeStartArray("actors")  
    .write("Keanu Reeves")  
    .write("Laurence Fishburne")  
    .write("Carrie-Ann Moss")  
  .writeEnd()  
  .write("year", 1999)  
  .writeEnd();
```


# Consumiendo JSON con el Streaming API

- ▶ La clase **JsonParser** contiene los métodos para parsear datos JSON
- ▶ Proporciona acceso “hacia delante” y de sólo lectura a los datos
- ▶ El objeto **JsonParser** se crea a partir de un stream de entrada (InputStream)

```
JsonParser parser = Json.createParser(new FileInputStream(...));
```

- ▶ Se pueden crear múltiples parsers usando **JsonParserFactory**
- ▶ Utiliza el método “**next()**” para obtener el evento del próximo estado de parseo
  - Los eventos pueden ser de los siguientes tipos:

# Consumiendo JSON con el Streaming API

- ▶ START\_OBJECT → representa el comienzo de un JSON
  - ▶ END\_OBJECT → representa el final de un JSON
  - ▶ KEY\_NAME → representa a la clave del par clave–valor
  - ▶ VALUE\_STRING → valor de tipo string
  - ▶ VALUE\_NUMBER → valor de tipo numérico
  - ▶ VALUE\_TRUE → valor de tipo true
  - ▶ VALUE\_FALSE → valor de tipo false
  - ▶ VALUE\_NULL → valor de tipo null
  - ▶ START\_ARRAY → representa el comienzo de un array
  - ▶ END\_ARRAY → representa el final de un array
- 

# Consumiendo JSON con el Streaming API

- ▶ El parser con los eventos `START_OBJECT` y `END_OBJECT` se corresponde a un JSON vacío  $\rightarrow \{ \}$
- ▶ Para un objeto con dos pares nombre/valor:

```
{  
  "apple": "red",  
  "banana": "yellow"  
}
```

- ▶ Se tienen los siguientes eventos (el método `next()` empezará por `START_OBJECT`, luego irá a `KEY_NAME`, etc...):

```
START_OBJECT  
  KEY_NAME ("apple"): VALUE_STRING ("red"),  
  KEY_NAME ("banana"): VALUE_STRING ("yellow")  
END_OBJECT
```

# Consumiendo JSON con el Streaming API

- ▶ Los eventos generados para un array con dos objetos JSON:

```
START_ARRAY  
  START_OBJECT KEY_NAME ("apple"): VALUE_STRING ("red") END_OBJECT,  
  START_OBJECT KEY_NAME ("banana"): VALUE_STRING ("yellow") END_OBJECT  
END_ARRAY
```

- ▶ O para una estructura anidada:

```
START_OBJECT  
  KEY_NAME ("title"): VALUE_STRING ("The Matrix"),  
  KEY_NAME ("year"): VALUE_NUMBER (1999),  
  KEY_NAME ("cast"): START_ARRAY  
    VALUE_STRING ("Keanu Reeves"),  
    VALUE_STRING ("Laurence Fishburne"),  
    VALUE_STRING ("Carrie-Anne Moss")  
  END_ARRAY  
END_OBJECT
```

# JAX-RS Entity Providers

- ▶ En el tema anterior vimos que dentro de la API cliente de JAX-RS podemos registrar providers:
  - “En muchos casos necesitaremos registrar providers, que nos digan cómo se van a leer o escribir (procesar) las entidades”

Order o = client		Order o = client
.target(...)		.register(OrderReader.class)
.request()	→	.target(...)
.get(Order.class);		.request()
		.get(Order.class);

- ▶ Los Entity Providers nos proporcionan el mapeo entre lo que nos llega del servicio REST y los tipos Java



# JAX-RS Entity Providers

- ▶ La información que nos llega en un mensaje HTTP puede ser de distintos tipos Java: String, byte[], java.io.InputStream, etc... que tienen un mapeo por defecto
- ▶ **Pero las aplicaciones pueden proporcionar su propio mapeo a los objetos particulares de la aplicación implementando las interfaces `MessageBodyReader` y `MessageBodyWriter`**
- ▶ La interfaz `MessageBodyReader` proporciona la conversión de un stream a un tipo Java (mediante el método **`readFrom`**)
- ▶ La interfaz `MessageBodyWriter` proporciona la conversión de un tipo Java a un stream (mediante el método **`writeTo`**)

# JAX-RS Entity Providers

- ▶ `MessageBodyWriter` (ejemplo para generar un JSON con un identificador (clase `Order`)):

```
@Provider
@Produces("application/json")
public class OrderWriter implements MessageBodyWriter<Order> {
    //...
    @Override
    public void writeTo(Order o, Class<?> type,
                        Type t,
                        Annotation[] as,
                        MediaType mt,
                        MultivaluedMap<String, Object> mm,
                        OutputStream out)
        throws IOException, WebApplicationException {
        JsonGeneratorFactory factory = Json.createGeneratorFactory();
        JsonGenerator gen = factory.createGenerator(out);
        {
            gen.writeStartObject()
            "id":"17"
            .writeEnd();
        }
    }
}
```

Le pasamos el objeto a escribir (un objeto de la clase "Order" con un valor en el identificador... por ejemplo supongamos que el valor que tiene es 17)

# JAX-RS Entity Providers

- ▶ El método `writeTo` escribe el JSON en el stream de respuesta HTTP
- ▶ Utiliza el API de streaming que acabamos de ver
- ▶ Procesa el objeto Java “Order o” y lo envía por medio del “OutputStream out”
- ▶ La anotación `@Provider` es obligatoria para que JAX-RS lo detecte
- ▶ La anotación `@Produces` indica el tipo de dato media que se produce

# JAX-RS Entity...

- ▶ `MessageBodyReader`  
(ejemplo para  
consumir un JSON  
con un identificador):

```
{  
  "id": "17"  
}
```

```
START_OBJECT  
  KEY_NAME ("id"): VALUE_NUMBER (17),  
END_OBJECT
```

```
@Provider  
@Consumes("application/json")  
public class OrderReader implements MessageBodyReader<Order> {  
  //...  
  @Override  
  public Order readFrom(Class<Order> type,  
                        Type t,  
                        Annotation[] as,  
                        MediaType mt,  
                        MultivaluedMap<String, String> mm,  
                        InputStream in)  
    throws IOException, WebApplicationException {  
    Order o = new Order();  
    JsonParser parser = Json.createParser(in);  
    while (parser.hasNext()) {  
      switch (parser.next()) {  
        case KEY_NAME:  
          String key = parser.getString();  
          parser.next();  
          switch (key) {  
            case "id":  
              o.setId(parser.getIntValue());  
              break;  
            default:  
              break;  
          }  
          break;  
        default:  
          break;  
      }  
    }  
    return o;  
  }  
}
```

# Proyecto de desarrollo

Vamos a incluir la funcionalidad  
de añadir películas

# JSON

- ▶ Vamos a utilizar el API Java para procesamiento de JSON para:
  - Incluir la opción para añadir una nueva película a la aplicación (implementaremos un POST en el API cliente)
  - Hacer que el API cliente utilice JSON para el intercambio de información (lo haremos para el POST y para el GET de id)
  - Definir entity providers de JAX-RS que permitirán leer (MovieReader) y escribir (MovieWriter) JSON para la aplicación empresarial

# JSON

- ▶ Poner el botón para añadir una película en el fichero movies.xhtml

```
<p:commandButton value="Nueva Película" action="addmovie" />
```



# JSON

- ▶ Dentro de la carpeta “client” crear un nuevo Facelets Template Client con la interfaz para añadir una película. Lo llamaremos “addmovie”.

The screenshot shows a 'New Facelets Template Client' dialog box with the following fields and options:

- Steps:**
  1. Choose File Type
  2. **Name and Location**
- Name and Location:**
  - File Name:** addmovie
  - Project:** cines
  - Folder:** client (with a 'Browse...' button)
  - Created File:** ts/cines/cines/cines/src/main/webapp/client/addmovie.xhtml
  - Template:** nain/webapp/WEB-INF/template.xhtml (with a 'Browse...' button)
  - Generated Root Tag:** ☒ <html> ☐ <ui:composition>
  - Sections To Generate:** ☒ content
- Buttons:** Help, < Back, Next >, **Finish**, Cancel



# JSON



Añadir nueva película

Título:

Actores:

Añadir

- ▶ Añadir lo siguiente a addmovie.xhtml:

```
<h1>Añadir nueva película</h1>
<h:form>
  <table cellpadding="5" cellspacing="5">
    <tr>
      <th align="left">Título:</th>
      <td><h:inputText value="#{movieBackingBean.movieName}"/> </td>
    </tr>
    <tr>
      <th align="left">Actores:</th>
      <td><h:inputText value="#{movieBackingBean.actors}"/></td>
    </tr>
  </table>
  <p:commandButton
    value="Añadir"
    action="movies"
    actionListener="#{movieClientBean.addMovie()}" />
</h:form>
```

Guardamos los datos  
introducidos por el  
usuario en  
MovieBackingBean



Método para añadir la  
película en  
MovieClientBean



# JSON

- ▶ El método solicita el nombre y los actores de la película a añadir y los guarda en el backing bean
- ▶ Luego llama al método addMovie() del cliente

# JSON

- ▶ Añadir movieName y actors al MovieBackingBean:

```
String movieName;  
String actors;
```

- ▶ Después generar los getter y los setter

# JSON

- ▶ Escribimos el método para añadir la película en la base de datos:

```
public void addMovie() {  
    Movie m = new Movie();  
    m.setId(1);  
    m.setName(bean.getMovieName());  
    m.setActors(bean.getActors());  
    target.register(MovieWriter.class)  
        .request()  
        .post(Entity.entity(m, MediaType.APPLICATION_JSON));  
}
```

Hacemos que utilice JSON para el intercambio de información



# JSON

- ▶ Además también modificamos el método `getMovie` para añadir el Entity Provider

```
public Movie getMovie() {  
    return target  
        .register(MovieReader.class)  
        .path("{movieId}")  
        .resolveTemplate("movieId", bean.getMovieId())  
        .request(MediaType.APPLICATION_JSON)  
        .get(Movie.class);  
}
```

Hacemos que utilice JSON para el intercambio de información

`.register(MovieReader.class)`

`.path("{movieId}")`

`.resolveTemplate("movieId", bean.getMovieId())`

`.request(MediaType.APPLICATION_JSON)`

`.get(Movie.class);`

# JSON

- ▶ Creamos una nueva clase llamada “MovieReader” y lo metemos dentro de un un nuevo paquete llamado “json”

New Java Class

**Steps**

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

**Name and Location**

Class Name:

Project:

Location:

Package:

Created File:

Help < Back Next > Finish Cancel

# JSON

- ▶ Añadimos las anotaciones (a nivel de clase) del MessageBodyReader

```
@Provider  
@Consumes(MediaType.APPLICATION_JSON)
```

- ▶ `@Provider` para que lo identifique la API JAX-RS
- ▶ `@Consumes` para que indique el tipo de datos que va a leer

# JSON

- ▶ Como estamos construyendo un componente Reader, tenemos que hacer que la clase implemente `MessageBodyReader<Movie>`

```
public class MovieReader implements MessageBodyReader<Movie> {
```

- ▶ Ahora añadir los import y después presionad en la “bombilla” para hacer que implemente todos los métodos abstractos (implement all abstract methods), lo que hará que se implementen los métodos `isReadable` y `readFrom`



# JSON

- ▶ Cambiar la implementación de isReadable por:

```
return Movie.class.isAssignableFrom(type);
```

- ▶ Este método determina si el MessageBodyReader puede producir una instancia de un tipo particular

# JSON

- ▶ Actualizar el método readFrom construyendo el parser que reciba una película, la parsee y la guarde en un objeto movie.
- ▶ El parser tiene que leer películas de nuestra AE, que cuando se devuelve en JSON tendrán la siguiente forma:

```
{  
  "id":1,  
  "name":"The Matrix",  
  "actors":"Keanu Reeves, Laurence  
    Fishburne, Carrie-Ann Moss"  
}
```

```
public Movie readFrom(Class<Movie> type,  
    Type genericType,  
    Annotation[] annotations,  
    MediaType mediaType,  
    MultivaluedMap<String, String> httpHeaders,  
    InputStream entityStream)  
    throws IOException, WebApplicationException {
```

```
    Movie movie = new Movie();  
    JsonParser parser = Json.createParser(entityStream);
```

```
    while (parser.hasNext()) {  
        switch (parser.next()) {  
            case KEY_NAME:  
                String key = parser.getString();  
                parser.next();  
                switch (key) {  
                    case "id":  
                        movie.setId(parser.getInt());  
                        break;  
                    case "name":  
                        movie.setName(parser.getString());  
                        break;  
                    case "actors":  
                        movie.setActors(parser.getString());  
                        break;  
                    default:  
                        break;  
                }  
                break;  
            default:  
                break;  
        }  
    }  
    return movie;  
}
```

# JSON

- ▶ Crear una nueva clase que sea MovieWriter
- ▶ Añadir las siguientes anotaciones:

`@Provider`

`@Produces(MediaType.APPLICATION_JSON)`

# JSON

- ▶ Como estamos construyendo un componente Writer, tenemos que hacer que la clase implemente `MessageBodyWriter<Movie>`

```
public class MovieWriter implements MessageBodyWriter<Movie> {
```

- ▶ Al igual que antes, presionar en la “bombilla” y hacer que implemente todos los métodos abstractos (implement all abstract methods), lo que hará que se implementen los métodos `isWriteable`, `getSize`, `writeTo`

# JSON

- ▶ Cambiar la implementación de `isWriteable` por:

```
return Movie.class.isAssignableFrom(type);
```

- ▶ Este método determina si `MessageBodyWriter` soporta un tipo particular

# JSON

- ▶ Añadir la implementación del método `getSize` como:

```
return -1;
```

- ▶ Este método está deprecado, pero se mantiene por compatibilidad. Se recomienda que siempre devuelva `-1`

# JSON

```
{  
    "id":1,  
    "name":"The Matrix",  
    "actors":"Keanu Reeves, Laurence  
        Fishburne, Carrie-Ann Moss"  
}
```

- ▶ Actualizar el método writeTo:

```
public void writeTo(Movie t, Class<?> type, Type genericType, Annotation[]  
annotations, MediaType mediaType, MultivaluedMap<String, Object> httpHeaders,  
OutputStream entityStream) throws IOException, WebApplicationException {  
    JsonGenerator gen = Json.createGenerator(entityStream);  
    gen.writeStartObject()  
        .write("id", t.getId())  
        .write("name", t.getName())  
        .write("actors", t.getActors())  
        .writeEnd();  
    gen.flush();  
}
```

- ▶ Escribimos un tipo en un mensaje HTTP. JsonGenerator escribe los datos JSON y lo pone en el stream de salida

# JSON

Ahora limpiamos, construimos y ejecutamos:

**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 Bourne Ultimatum

☐ The Pink Panther

**Detalles**

**Nueva Película**



# JSON

**Cines Luz de Castilla - Segovia**

**Menú**

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

**Añadir nueva película**  

Título:

Actores:

**Añadir**

# JSON

**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 Bourne Ultimatum

☐ The Pink Panther

☐ Pelicula 1

Detalles

Nueva Película

# JSON

- ▶ Otra comprobación fácil de que funciona correctamente es obligar a que lo que devuelve el servicio REST sea JSON
- ▶ Después podemos hacer la llamada desde el navegador y ver si efectivamente nos devuelve un JSON
- ▶ Ejemplo, para el id 1 debería devolver:

---

```
{"id":1,"name":"The Matrix","actors":"Keanu Reeves, Laurence Fishburne, Carrie-Ann Moss"}
```

# Entrega 3

- ▶ Tercera (y última) entrega de la aplicación de los cines:
  - Añadir RESTful (Tema 5)
  - Añadir JSON (Tema 6)
- ▶ Fecha límite: Viernes, 13 de mayo a las 23:59
- ▶ **El resto de temas de de Java EE para la aplicación de los cines (que veremos en las próximas clases) no serán de entrega obligatoria en evaluación continua... pero se recomienda hacerlos primero en la aplicación de los cines y preguntar las dudas ya que lo que se explique en esos temas hay que implementarlo en la práctica final!!! (JAAS y websockets)**

**¿PREGUNTAS?**