Eusko Jaurlaritzaren Informatika Elkartea
Sociedad Informática del Gobierno Vasco

UDA - Utilidades de desarrollo de aplicaciones

Generación de informes

Fecha: 19/02/2013 | **Referencia:**

EJIE S.A.
Mediterráneo, 14
Tel. 945 01 73 00*
Fax. 945 01 73 01
01010 Vitoria-Gasteiz
Posta-kutxatila / Apartado: 809
01080 Vitoria-Gasteiz
www.ejie.es

Control de documentación

Título de documento: Generación de informes

Histórico de versiones

Código:	Versión:	Fecha:	Resumen de cambios:
	1.0.0	19/02/2013	Primera versión.

Cambios producidos desde la última versión

Control de difusión

Responsable: Ander Martínez

Aprobado por:

Firma: Fecha:

Distribución:

Referencias de archivo

Autor:

Nombre archivo:

Localización:

Contenido

1 Introducción

En el presente documento pretende explicar la manera en la que se ha decidido implementar la generación de informes en UDA, incluyendo diversos ejemplos.

Los informes generados en UDA pueden ser de diversa índole en cuanto a su tipo:

- Valores separados por comas (CSV)
- Hoja de cálculo de Excel (XLS o XLSX)
- Hoja de cálculo en formato abierto (ODS)
- Documento portátil (PDF)

En función del tipo de informe, se deberán crear diferentes estructuras de datos como se verá más adelante, aunque se ha intentado homogeneizar las diversas implementaciones.

2 Arquitectura

En el presence capítulo se van a detallar las librerías requeridas en función del tipo de informe así como las clases incluídas en la librería x38.

2.1 Librerías requeridas

La siguiente lista contiene las librerías (.jar) requeridas en la generación de cada tipo de documento:

- CSV
- XLS:
  - poi-3.7.jar

dependencias_xls.xml

- XLSX:
  - poi-3.7.jar
  - poi-ooxml-3.7.jar
  - poi-ooxml-schemas-3.7.jar
  - dom4j-1.6.1.jar **(en tiempo de ejecución)**

dependencias_xlsx.xml

- ODS:
  - odfdom-java-0.8.7.jar

dependencias_ods.xml

- PDF:
  - itext-2.1.7.jar
  - jasperreports-4.7.1.jar

dependencias_pdf.xml

2.2 Clases de x38

La siguiente lista contiene las clases contenidas en x38 para la generación de informes:

- com.ejie.x38.reports.ReportData:
  - Clase para la transmisión de los datos de los informes.

- com.ejie.x38.reports.CSVReportView
  - Clase encargada de la generación de informes .CSV.
- com.ejie.x38.reports.ODSReportView
  - Clase encargada de la generación de informes .ODS.
- com.ejie.x38.reports.PDFReportView
  - Clase encargada de la generación de informes .PDF.
- com.ejie.x38.reports.AbstractPOIExcelView.java
  - Clase de abstracta para la generación de ficheros XLS y XLS
- com.ejie.x38.reports.XLSReportView *(extiende de AbstractPOIExcelView)*
  - Clase encargada de la generación de informes .XLS.
- com.ejie.x38.reports.XLSXReportView *(extiende de AbstractPOIExcelView)*
  - Clase encargada de la generación de informes .XLSX

3 Configuración

La generación de informes se define a nivel de WAR. Por tanto cada aplicación deberá realizar tantas configuraciones como módulos web tenga.

3.1 reports-config.xml

La configuración relativa a la generación de informes se declara en el fichero “reports-config.xml” indicando el nombre de la vista (por defecto) y la clase clases de UDA encargada del proceso.

```
<!-- UDA exporters -->
<bean id="csvReport"  class="com.ejie.x38.reports.CSVReportView" />
<bean id="odsReport"  class="com.ejie.x38.reports.ODSReportView" />
<bean id="xlsReport"  class="com.ejie.x38.reports.XLSReportView" />
<bean id="xlsxReport" class="com.ejie.x38.reports.XLSXReportView" />
```

3.2 mvc-config.xml

La gestión de las vistas para los informes se realiza mediante la clase *XmlViewResolver* de Spring que referenciará al fichero “reports-config.xml”

```
<!-- Reports -->
<bean class="org.springframework.web.servlet.view.XmlViewResolver">
   <property name="location">
       <value>/WEB-INF/spring/reports-config.xml</value>
   </property>
   <property name="order" value="0" />

</bean>
```

3.3 app-config.xml

Se incluye la referencia al nuevo fichero de configuración, en el fichero maestro de las configuraciones.

```
<import resource="reports-config.xml"/>
```

4 Funcionamiento

El funcionamiento de los informes es bastante similar independientemente del tipo de documento como se verá en los siguientes apartados.

La manera en la que Spring determina el tipo de vista debe utilizar, y por tanto el tipo de informe a generar se realiza a través del dato retornado en el método del Controller. Se puede devolver tanto un String como un objeto View, siendo la primera opción la más común en UDA. La relación entre el String y el tipo de archivo se define en el fichero reports-config.xml y por defecto es la siguiente:

- csvReport ➔ Informe CSV
- xlsReport ➔ Informe XLS
- xlsxReport ➔ Informe XLSX
- odsReport ➔ Informe ODS

NOTA: Se recomienda no cambiar las vistas para homogenizar su uso en todas las aplicaciones UDA.

En el caso de los PDF, como se verá más adelante, se definirá una entrada por cada informe a generar ya que deberá definirse su plantilla Jasper correspondiente. Por ejemplo:

```
<bean id="pdfUsuario" p:url="/WEB-INF/resources/reports/usuario.jasper"
      class="com.ejie.x38.reports.PDFReportView" />
```

El controller encargado del informe, deberá tener un método que cumpla lo siguiente:

- Las peticiones para la generación de informes se realizan mediante el método POST, ya que se utiliza un componente de RUP para la gestión de esperas como se verá más adelante (rup.report). Aunque es posible usar peticiones GET sin usar el componente de RUP (o se puede dejar sin método asociado para que capture ambos tipos de métodos).
- El envío de los datos entre el Controller y la vista se realiza a través del objeto ModelMap de Spring, que se podrá instaciar (**new** ModelMap( )) o declarar como parámetro del método.
- Los datos se almacenarán en una variable “reportData” del modelo y serán objetos de tipo ReportData (uno solo o una lista dependiendo del tipo).
- En los informes en los que se necesiten como dato las columnas relativas a un grid de la página (y si se utiliza el componente rup.report) se recibirá como parámetro cuyo nombre será “**columns**”.
- El retorno del método marcará el tipo de informe a generar según lo comentado anteriormente.
- El nombre del fichero resultante se declarará como un String almacenado en la variable “fileName” del modelo. No requiere extensión ya que la vista se encarga de añadírsela para evitar problemas.

```
@RequestMapping(value = "csvReport", method = RequestMethod.POST)
protected ModelAndView getCSVReport(@ModelAttribute Usuario filterUsuario,
                                    @ModelAttribute Pagination pagination,
                                    ModelMap modelMap,
      @RequestParam(value = "columns", required = false) String columns){
      //Nombre fichero
      modelMap.put("fileName", "datosCSV");
      …
      //Generación del CVS
      return new ModelAndView("csvReport", modelMap);
}
```

4.1 Estructura de datos (ReportData)

La transferencia de datos entre el controllador y la vista encargada de la generación del informe se realiza a través de un POJO de x38 “ReportData”. Esta clase contiene las siguientes propiedades:

- sheetName (String) → Nombre de la hoja que contiene los datos (para ficheros de hojas de cálculo).
- headerNames (Map <String, String>) → Nombre de las columnas (usado para introspección)
- modelData (List<T>) → Lista de objetos del modelo del que se quiere generar informe.
- showHeaders (bolean) → Indica si se mostrará línea con cabecera de columnas (por defecto *true*), salvo en PDF que depende de una plantilla.

El atributo *headerNames* se utiliza para indicar el nombre de los atributos del modelo que se desean plasmar en el informe y que se obtienen mediante instrospección, por lo que será de obligatorio cumplimiento.

ReportData cuenta con una función estática de nombre *parseColumns* para facilitar el procesamiento del parámetro *columns* en los casos en los que se quiera enviar el nombre y posición de las columnas de un grid al informe. Ejemplo:

```
reportData.setHeaderNames(ReportData.parseColumns(columns));
```

Pueden existir casuísticas en las que se requiera modificar las columnas enviadas desde el navegador. Para ello se definirá un mapa de tipo <String, String>. El *key* del mapa será el nombre de la columna enviada desde la JSP y el *value* será el nombre que se usará en el generador del informe. En el caso de querer eliminar una columna, estableceremos el valor a vacío “”.

```
// cabeceras hoja
Map<String, String> columnsMatch = new HashMap<String, String>();

// Cambiar columna
columnsMatch.put("xxx.columna1", "xxx.columna2");
// Eliminar columna
columnsMatch.put("xxx.columnaN", "");

reportData.setHeaderNames(ReportData.parseColumns(columns, columnsMatch));
```

Si se desea gestionar manualmente los datos de las cabeceras (por ejemplo si no se tiene grid), se deberá crear un *LinkedHashMap* cuya clave (key) será el nombre del atributo del objeto de Modelo y cuyo valor (value) será el literal que aparecerá como texto (que deberá ser internacionalizado dependiendo del idioma). Ejemplo:

```
Usuario {
      String id;
      String nombre;
      ...
}

LinkedHashMap<String, String> mapa = new LinkedHashMap<String, String>();
mapa.put("id", "Identificador");
mapa.put("nombre", "Nombre");
...
reportData.setHeaderNames(mapa);
```

El informe en CSV sólo contiene una “hoja” por lo que el atributo *sheetName* no es de obligado cumplimiento (si se rellena no se aplica), en cambio los informes XLS, XLSX u ODS si que pueden contener diferentes hojas y se podrá definir el nombre para cada una de ellas. Por este motivo, a la hora de enviar los datos en el caso de CSV se enviará un objeto ReportData a la vista, mientras que en los otros tres casos se enviará una lista de objetos ReportData.

El informe en PDF es algo diferente ya que no se utiliza esta estructura para el envío de los datos como se verá más adelante.

4.1.1. Tablas de datos con agrupamiento

Los componenetes de RUP permiten el agrupamiento de datos en las tablas mediante una de sus columnas.

En el caso de que se quiera representar estos mismos grupos en el informe a generar existen los siguientes atributos del objeto ReportData:

- grouping (bolean) → Indica si se desea aplicar el agrupamiento (por defecto *false*).
- groupColumnName (String) → Indica el nombre de la columna por la que se agrupa.
- showGroupColumng (bolean) → Indica si se desea mostrar la columna por la que se agrupa(por defecto *false*),

NOTA: Estas propiedades no son aplicables al PDF ya que este se genera en base a una plantilla.

4.2 Metadatos de jerarquía (JerarquiaMetadata)

Al generar el informe de un grid con jerarquía es posible que se desee que el informe tenga el mismo aspecto visual que la tabla en pantalla. Para ello existe el objeto *JerarquiaMetadata* que contiene las siguientes propiedades:

- **FILTRO**
  - showFiltered (bolean) → Indica si se genera una columna para indicar los elementos que cumplen el filtro de búsqueda (por defecto *true*).
  - filterToken (String) → valor que se mostrará como indicador de que el elemento cumple el filtro (por defecto *).
  - filterHeaderName (String) → nombre de la columna usada para aplicar la agrupación. Tendrá el valor definido en la JS como “groupingView.groupingField” (a nivel del grid).
- **TABULACIÓN**
  - showTabbed (bolean) → Indica si se aplica la tabulación de una determinada columna para mostrar visualmente la jerarquía (por defecto *true*).
  - tabToken (String) → valor que utilizará para aplicar la tabulación (por defecto “ “).
  - tabColumnName (String) → nombre de la columna sobre la que se aplicará la tabulación. Tendrá el valor definido en la JS como “expandColName”.
- **EXPANDIDOS/CONTRAIDOS**
  - showIcon (bolean) → Indica si se mostrarán los iconos de elemento expandido/contraido (por defecto *false*).
  - iconExpanded (String) → valor que se utilizará para los elementos expandidos (por defecto [-]).
  - iconUnexpanded (String) → valor que se utilizará para los elementos expandidos (por defecto [+]).
  - iconNoChild (String) → valor que se utilizará para los elementos expandidos (por defecto [ ]).
  - iconColumnName (String) → nombre de la columna sobre la que se aplicarán los iconos. Tendrá el valor definido en la JS como “expandColName”.
  - iconBeanAtribute (String) → nombre del atributo del Model utilizado como indicador de elemento contraido. Tendrá el valor definido en la JS como “relatedColName”.

- iconCollapsedList (List<String>) → lista de elementos contraidos. Se obtiene del objeto *Pagination* mediante el método .getTree().

A continuación se detalla um ejemplo:

```
//Filtro
jmd.setShowFiltered(true);
jmd.setFilterToken("*");
jmd.setFilterHeaderName("Filtrados");

//Tabulacion
jmd.setShowTabbed(true);
jmd.setTabToken("   ");
jmd.setTabColumnName("nombre");

//Iconos
jmd.setShowIcon(true);
jmd.setIconExpanded("[-]");
jmd.setIconUnexpanded("[+]");
jmd.setIconExpanded("[ ]");
jmd.setIconColumnName("nombre")
jmd.setIconBeanAtribute("id");
jmd.setIconCollapsedList(pagination.getTree());
```

Una vez instanciado y configurado el objeto, bastará con asociarlo al *ReportData* que se usará para la generación de informe de la siguiente forma:

```
usuarioExcelDataPage.setJerarquiaMetadada(jmd);
```

4.3 Informe CSV

Este informe contiene “una página” por lo que sólo se guardará un objeto de tipo ReportData en el modelo.

A continuación se detalla un ejemplo de un informe CSV:

```
@RequestMapping(value = "csvReport", method = RequestMethod.POST)
protected ModelAndView getCSVReport(
            @ModelAttribute Usuario filterUsuario,
            @ModelAttribute Pagination pagination,
            ModelMap modelMap,
            @RequestParam(value = "columns", required = false) String columns){

      //Obtener datos
      List<Usuario> listUsuarioAll = ...

      //Nombre fichero
      modelMap.put("fileName", "datosCSV");

      //Datos
      ReportData<Usuario> reportData = new ReportData<Usuario>();
            //cabeceras hoja
            reportData.setHeaderNames(ReportData.parseColumns(columns));
            //datos hoja
            reportData.setModelData(listUsuarioAll);
      modelMap.put("reportData", reportData);

      //Generación del CVS
      return new ModelAndView("csvReport", modelMap);
```

```
}
```

Se ha definido como separador de columnas el punto y coma “;” para que al abrir el fichero con Excel, se repartan las columnas en las diferentes celdas. En caso de querer cambiar el separador se podrá hacer definiendolo en el modelo mediante el atributo “separator":

```
modelMap.put("separator", ",");
```

4.4 Informe XLS, XLSX u ODS

Estos informes pueden contener una o varias páginas por lo que se guardará una lista de objetos del tipo ReportData en el modelo.

Su implementación solo difiere en el retorno:

- XLS → `return new ModelAndView("xlsReport", modelMap);`
- XLSX → `return new ModelAndView("xlsxReport", modelMap);`
- ODS → `return new ModelAndView("odsReport", modelMap);`

A continuación se detalla un ejemplo de un informe XLS, XLSX u ODS (se ha utilizado el literal XXX para que sea genérico):

```
@RequestMapping(value = "xxxReport", method = RequestMethod.POST)
protected ModelAndView getXXXReport(
            @ModelAttribute Usuario filterUsuario,
            @ModelAttribute Pagination pagination,
            ModelMap modelMap,
            @RequestParam(value = "columns", required = false) String columns){

      //Obtener datos
      List<Usuario> listUsuarioAll = ...
      List<Usuario> listUsuarioPage = ...

      //Nombre fichero
      modelMap.put("fileName", "datosXXX");

      //Datos
      List<Object> reportData = new ArrayList<Object>();
            //Hoja 1
            ReportData<Usuario> usuarioExcelDataAll = new ReportData<Usuario>();
                  //nombre hoja
                  usuarioExcelDataAll.setSheetName("Todos los usuarios");
                  //cabeceras hoja
                  usuarioExcelDataAll.setHeaderNames(
                        ReportData.parseColumns(columns));
                  //datos hoja
                  usuarioExcelDataAll.setModelData(listUsuarioAll);
            reportData.add(usuarioExcelDataAll);
            //Hoja 2
            ReportData<Usuario> usuarioExcelDataPage = new ReportData<Usuario>();
                  //nombre hoja
                  usuarioExcelDataPage.setSheetName("Página 1 de usuarios");
                  //cabeceras hoja
                  usuarioExcelDataPage.setHeaderNames(
                        ReportData.parseColumns(columns));
                  //datos hoja
                  usuarioExcelDataPage.setModelData(listUsuarioPage);
```

```
            reportData.add(usuarioExcelDataPage);
            //Hoja n
            ...
        modelMap.put("reportData", reportData);

        //Generación del informe
        return new ModelAndView("xxxReport", modelMap);
}
```

4.5 Informe PDF

El informe en formato PDF difiere de los anteriores informes ya que no hace uso del POJO *ReportData* ya que los datos se envían directamente insertándolos en el modelo.

```
@RequestMapping(value="/pdfReport")
public ModelAndView generarPDFJasperReport(
            @ModelAttribute Usuario filterUsuario,
            @ModelAttribute Pagination pagination,
            ModelMap modelMap,
            @RequestParam(value = "isInline", required = false) boolean isInline){

        //Obtener datos
        List<Usuario> usuarios = ...

        //Nombre fichero
        modelMap.put("fileName", "datosPDF");

        //En línea (no descarga fichero) ?
        modelMap.put("isInline", isInline);

        //Titulo y cabeceras (parameter)
        modelMap.put("TITULO", "Listado usuarios");
        modelMap.put("COL_NOMBRE", "Nombre");
        modelMap.put("COL_APE1", "Apellido 1");
        modelMap.put("COL_APE2", "Apellido 2");

        //Datos (field)
        modelMap.put("usuarios", usuarios);

        //Generación del PDF
        return new ModelAndView("pdfUsuario", modelMap);
}
```

4.5.1. Plantilla de Jasper

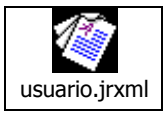
Se ha optado por usar plantillas de Jasper Report a la hora de plasmar los datos en un informe PDF. La plantilla asociada al informe se declara en el fichero reports-config.xml del War correspondiente.

La ruta destinada a alojar las plantillas será: \xxxYYYWar\WebContent\WEB-INF\resources\reports

Existe la posibilidad de indicar la plantilla en formato .jrxml o en formato .jasper (compilada), aunque se recomienda esta última ya que evitamos que Spring tenga que compilar la plantilla en tiempo de ejecución la primera vez que se utilice.

Se recomienda el uso de la herramienta iReport para la generación de plantillas y su compilación.

A continuación se adjunta la plantilla para el código del ejemplo:



4.5.2. Opción en línea

Existe la posibilidad de mostrar directamente el informe PDF en el navegador (pestaña nueva) sin necesidad de mostrar el diálogo de descarga. Para ello se puede introducir un atributo en el modelo de nombre *isInline*, que indica si se desea que el fichero tenga como "Content-disposition" el valor "inline" en lugar del "attachment" con el nombre del fichero (esta última es la opción por defecto).

En este caso, el nombre del posible fichero resultante (si se pulsa en guardar) no será el nombre indicado como atributo *fileName*, si no que será el nombre de la vista, es decir, el literal que se retorna del controlador.

Al abrirse el fichero en una nueva ventana (por ejemplo con un window.open desde JavaScript), el método de la petición será GET. Además en este caso no tiene sentido el uso del componente rup.report ya que el informe se visualizará una nueva pestaña en la que se espera a la carga del fichero.

5 Manual de migración

En este apartado se detallan lo pasos para incluir la exportación de datos en aquellas aplicaciones ya desarrolladas con UDA antes de que existiera esta funcionalidad:

1. Agregar las librerías necesarias al EAR: Se deberá añadir al pom.xml de la aplicación (EAR) las dependencias requeridas en cada caso y lanzar su tarea Ant asociada (EAR – mavenRunDependencies):
   a. poi-3.7.jar
   b. poi-ooxml-3.7.jar
   c. poi-ooxml-schemas-3.7.jar
   d. dom4j-1.6.1.jar
   e. odfdom-java-0.8.7.jar
   f. itext-2.1.7.jar
   g. jasperreports-4.5.1.jar
2. Añadir el fichero **reports-config.xml** con el siguiente contenido

```
<beans xmlns="http://www.springframework.org/schema/beans"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:p="http://www.springframework.org/schema/p"
            xsi:schemaLocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-
      3.1.xsd">

      <!-- UDA exporters -->
      <bean id="csvReport" class="com.ejie.x38.reports.CSVReportView" />
      <bean id="odsReport" class="com.ejie.x38.reports.ODSReportView" />
      <bean id="xlsReport" class="com.ejie.x38.reports.XLSReportView" />
      <bean id="xlsxReport" class="com.ejie.x38.reports.XLSXReportView" />

</beans>
```

   NOTA: La ruta del fichero es “xxxYYYWar\WebContent\WEB-INF\spring\reports-config.xml”

3. Definir el fichero reports-config.xml como vista en el fichero **mvc-config.xml**:

```
      <!-- Reports -->
      <bean class="org.springframework.web.servlet.view.XmlViewResolver">
         <property name="location">
             <value>/WEB-INF/spring/reports-config.xml</value>
         </property>
         <property name="order" value="0" />

      </bean>
```

4. Añadir la referencia al fichero reports-config.xml en el fichero **app-config.xml**:

```
      <import resource="jackson-config.xml" />
      <import resource="validation-config.xml"/>
      <import resource="mvc-config.xml" />
      <import resource="log-config.xml"/>
      <import resource="security-core-config.xml"/>
      <import resource="security-config.xml"/>
      <import resource="reports-config.xml"/>
```

5. Incluir la versión del jar de x38 que contiene las clases para generar informes (versión > 2.1.1)

6 Componente rup-report

Entre los componentes de RUP se encuentra rup-report. La finalidad de este componente es ayudar al desarrollador a presentar los botones para generación de documentos así como implementar un conjunto de diálogos de espera y error para mejorar la experiencia de usuario.

Como ejemplo destacar la funcionalidad de este componente que automáticamente envía como parámetro las columnas asociadas a un *grid* para que simplemente se deba obtener el parámetro y a través del modelo enviarlo a la vista, siendo su gestión y manipulación transparante al desarrollador.

Para más información al respecto, consultar la documentación del componente.

7 Implementación propia

Las funcionalidades implementadas en los informes de UDA pueden, en ciertos casos, llegar a ser insuficientes. Por ello siempre quedará la posibilidad de realizar una implementación propia, aunque se recomiendan algunos aspectos a tener en cuenta:

1. Si se desea realizar una implementación similar a las existentes pero con código propio se deberán seguir los siguientes pasos:
   a. Crear una clase con las siguiente estructura:

```
import org.springframework.web.servlet.view.AbstractView;
...
public class XXXView extends AbstractView{
    @Override
    protected void renderMergedOutputModel(Map<String, Object> model,
        HttpServletRequest request, HttpServletResponse response) throws Exception {
            ...

    }

}
```

   b. Definir la clase al repositorio de clases para generar informes (**reports-config.xml**):

```
<bean id="myReport" class="com.ejie.xxx.reports.XXXView" />
```

   c. En el *Controller* correspondiente devolver el literal definido para el informe propio:

```
return new ModelAndView("myReport ", modelMap);
```

2. Si se utiliza el componente **rup-report** se deberá crear la cookie que indica que el informe se ha generado.

```
Cookie cookie = new Cookie("fileDownload", "true");
cookie.setPath("/");
response.addCookie(cookie);
```

3. Cabeceras y objetos para la salida de datos utilizados en los informes generados por UDA:
   a. CSV

```
//Tipo y Nombre Fichero
response.setHeader("Content-type", "application/octet-stream; charset=ISO-8859-1");
response.setHeader("Content-Disposition", "attachment; filename=XXX.csv");

BufferedWriter writer = new BufferedWriter(response.getWriter());
```

   b. XLS

```
//Tipo y Nombre Fichero
response.setHeader("Content-type", " application/vnd.ms-excel");
response.setHeader("Content-Disposition", "attachment; filename=XXX.xls");

org.apache.poi.hssf.usermodel.HSSFWorkbook;
```

   c. XLSX

```
//Tipo y Nombre Fichero
response.setHeader("Content-type", " application/vnd.openxmlformats-
officedocument.spreadsheetml.sheet");
response.setHeader("Content-Disposition", "attachment; filename=XXX.xlsx");

org.apache.poi.xssf.usermodel.XSSFWorkbook
```

   d. ODS

```
//Tipo y Nombre Fichero
response.setHeader("Content-type", " application/vnd.oasis.opendocument.spreadsheet");
response.setHeader("Content-Disposition", "attachment; filename=XXX.ods");

org.odftoolkit.odfdom.doc.OdfSpreadsheetDocument
```