MANUAL

Programación en Cividas

Caso práctico

DIRIGIDO A

Desarrolladores

FECHA

3 de Marzo de 2012

Caso practico de programación en Cividas

Inscripción Registro Mercantil de Pontevedra, Tomo 3116, Libro 3116, Folio 30, Hoja PO-38276 – C.I.F.: B-36.499.960

imatia
your innovation partner

Autor	**Carlos Fernández Villar**
Fecha creación	03 Marzo 2012

Versiones			
1.0	03/03/2012	Carlos Fernández	Versión inicial del documento.

Índice

Creación de un proyecto Cividas en Eclipse

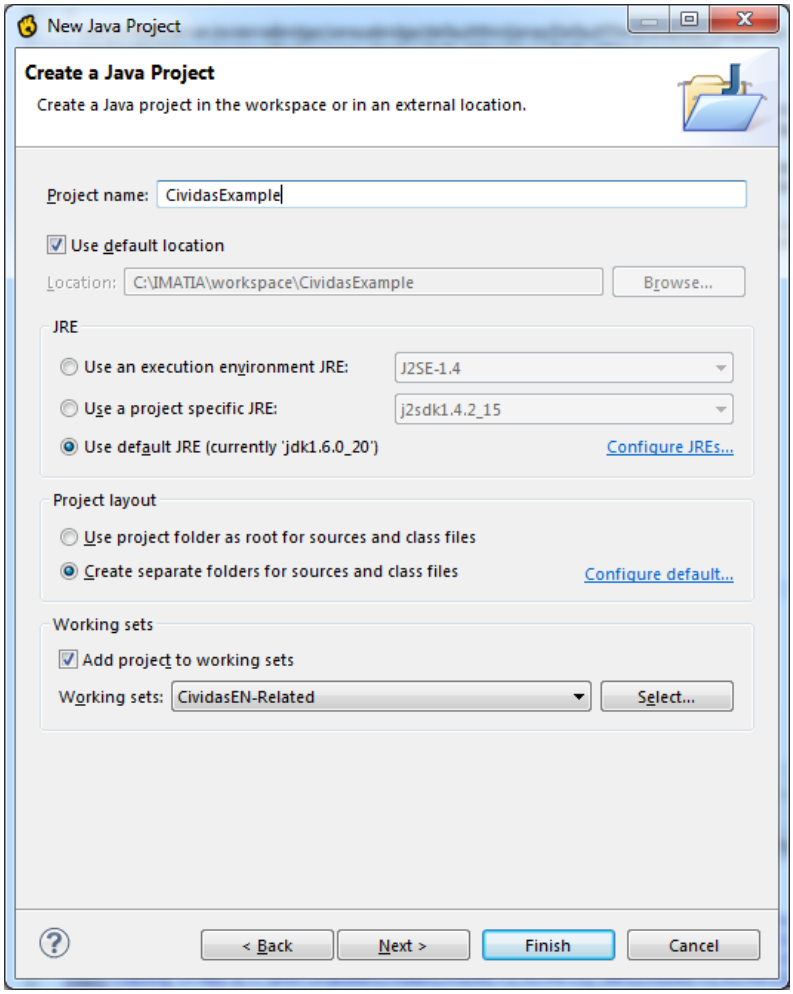
Base de datos

Creamos la base de datos. Se importa la versión actual de la misma.

Proyecto Eclipse

Nuevo proyecto

Creamos un nuevo proyecto Java:



Archivos de propiedades

Incorporamos los archivos de propiedades. Las rutas y nombres de los mismos no tienen por que ser estos, siempre y cuando se configuren correctamente (el lanzador del servidor apunta al server.properties y luego unos se llaman a otros).

Archivos básicos

license.dat

com/cividas/server/conf/server.properties
com/cividas/server/conf/locator.properties
com/cividas/server/conf/database.properties
com/cividas/server/conf/remotereferences.xml

Otros

com/cividas/server/externalbridge/
com/cividas/server/registryintegration/prop/
com/cividas/server/registryintegration/prop/mapping/

Librerías / Proyectos asociados

Se incorporan las librerías:

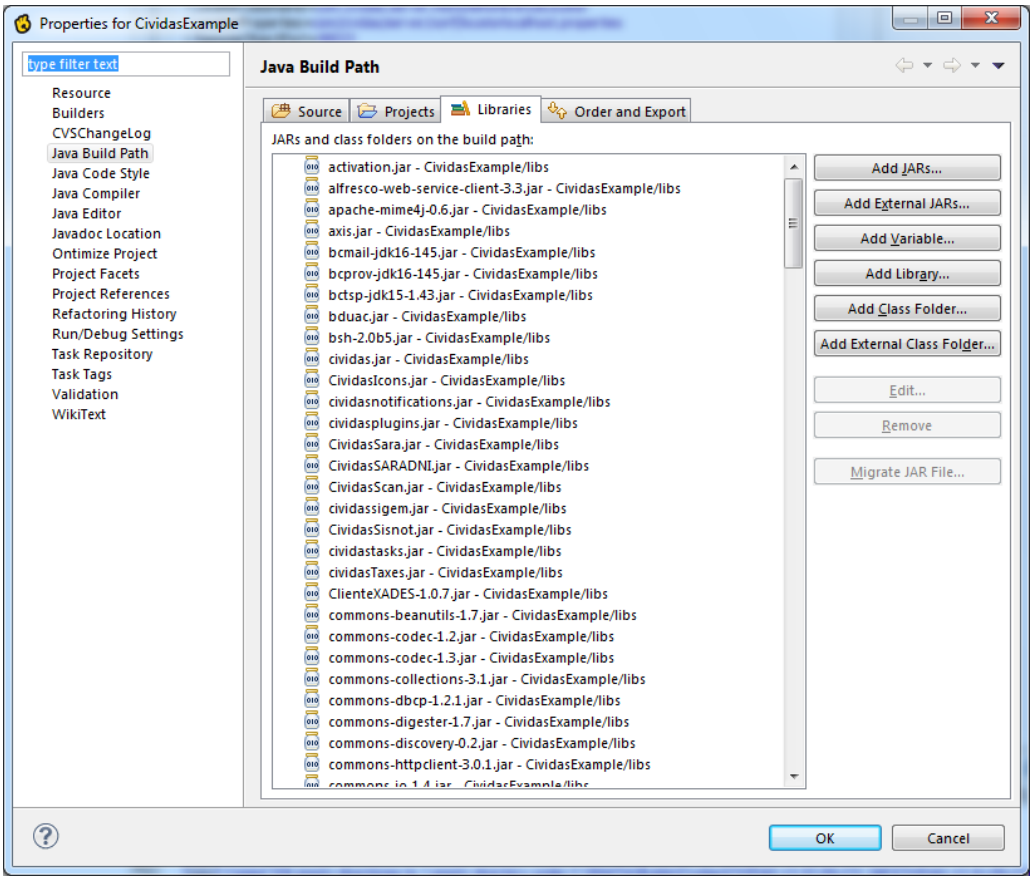
- ¿Librerías cliente VS Librerías servidor?

En la creación del proyecto no es imprescindible hacer la separación de librerías de cliente y de servidor, aunque no sean siempre las mismas. Desde el desarrollo **de** Cividas si se hace sobre todo por el hecho de que el número de librerías es bastante elevado y se hace necesario limitar al máximo las librerías que se va a descargar el cliente.

En un entorno de desarrollo **sobre** Cividas no es tan necesario ya que, a no ser que se aporte un número considerable de librerías a mayores de las de plataforma, será la propia plataforma la que marque que va en cliente y que va en servidor.

- ¿Estructuración en carpetas?

Al igual que con respecto al caso anterior, no es imprescindible la estructuración en carpetas de las librerías. En el caso de desarrollo de Cividas se hace por temas de organización, pero en cualquier otro caso no es necesario.



En algunos casos, puede ser necesario crear proyectos paralelos en lugar de incorporar librerías. Es recomendable, p.e. en procedimientos de integración independizar el conector en sí del uso que se hace del mismo en la plataforma.

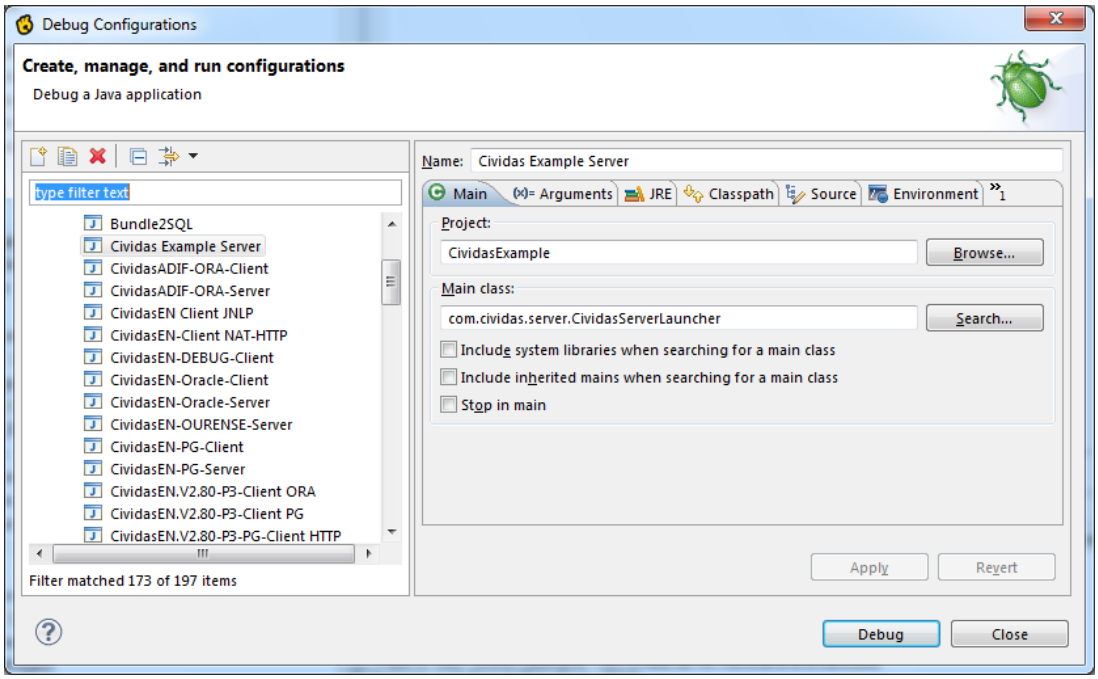
Lanzadores

Es conveniente guardar los lanzadores en el propio proyecto, de forma que se puedan compartir entre los distintos desarrolladores. Para ello en la pestaña Common tenemos la opción de Local file / Shared file. Eligiremos la segunda.

Existen por otro lado unas pequeñas diferencias entre el lanzador del servidor y el del cliente.

Lanzador del servidor

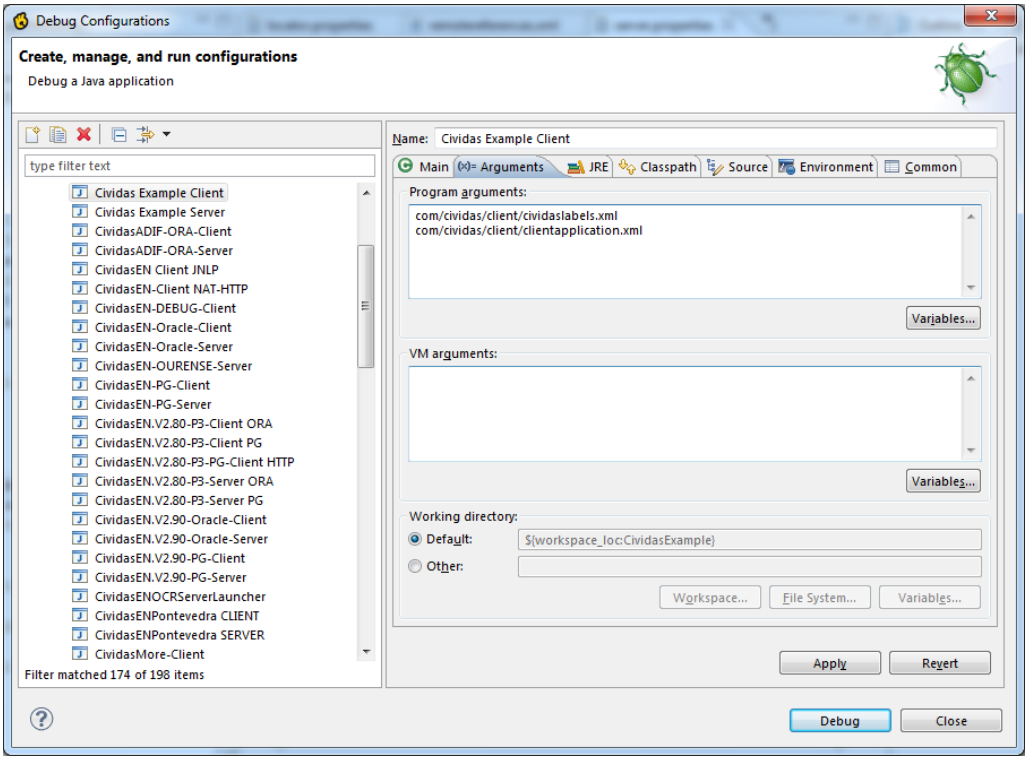
El lanzador del servidor utiliza como parámetro la ruta del server.properties utilizado. Las librerías utilizadas pueden ser directamente las del proyecto sin más, ya que el orden de las mismas no debería ser determinante (no siempre en las instalaciones podremos asegurar un orden en las librerías del servidor



Lanzador del cliente

Los parámetros de lanzamiento del cliente son los siguientes:

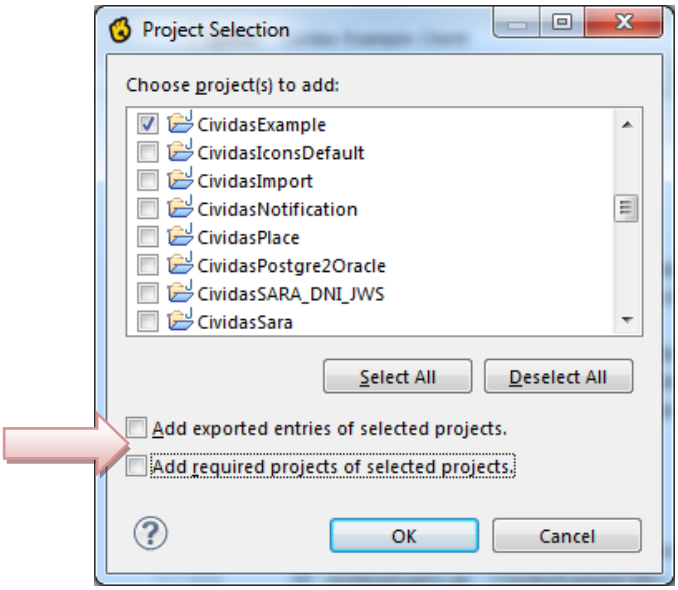
- Ruta del XML de etiquetas (com/cividas/client/cividaslabels.xml)
- Ruta del XML de arquitectura de formularios (com/cividas/client/clientapplication.xml)



En el caso del cliente, el orden de las librerías sí que tiene una cierta importancia y, de hecho, en las instalaciones se puede especificar (el orden es el indicado en el JNLP). Para simular esta circunstancia, las librerías deben ser añadidas UNA a UNA.

Por tanto desde la pestaña Classpath haremos lo siguiente:

- Eliminar la entrada por defecto (que apunta al proyecto completo)
- Añadir el proyecto pero desactivando todas las opciones



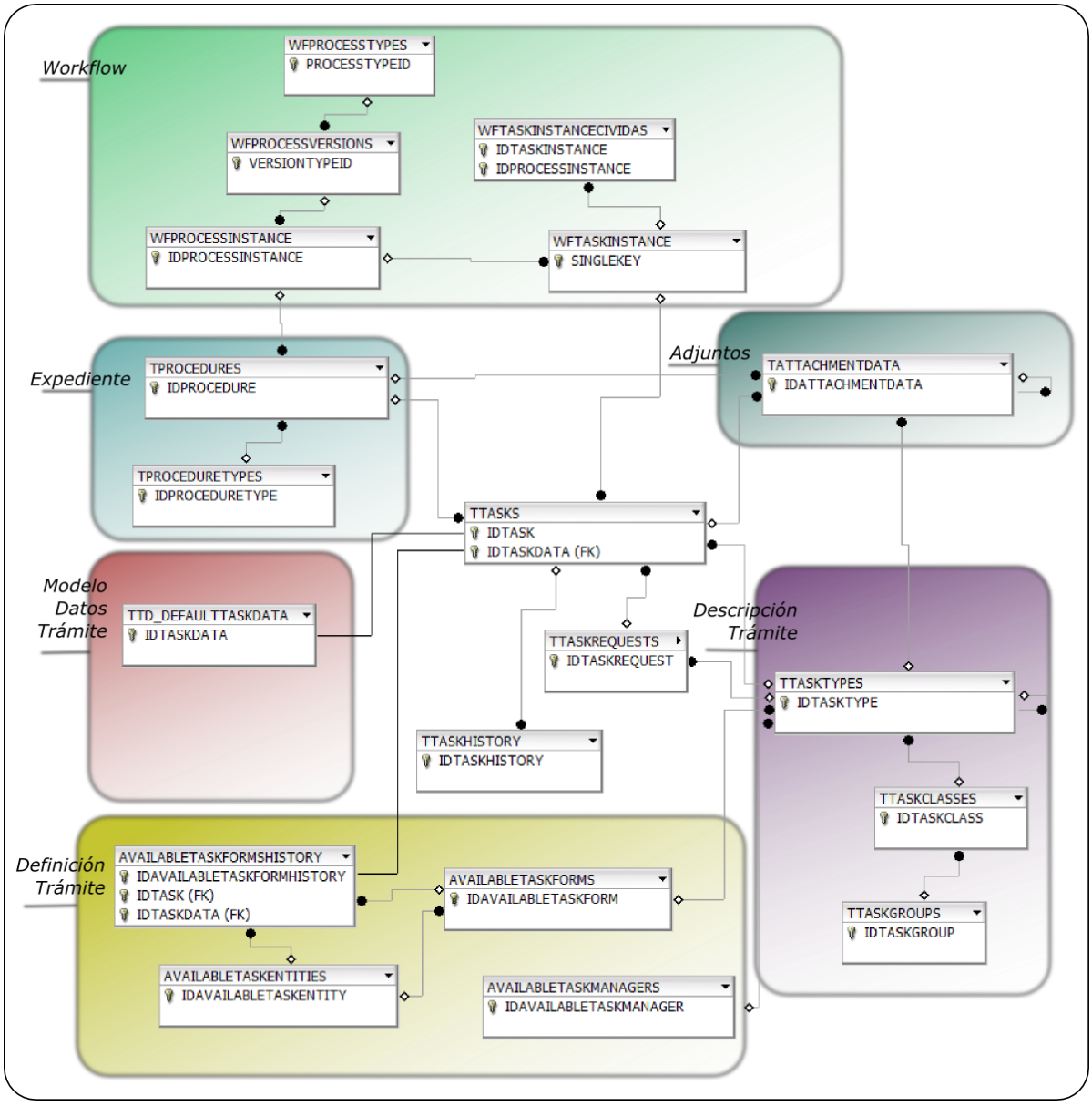
- Añadir el conjunto de librerías de forma independiente. Esto nos permite particularizar el orden de las mismas. Debería ser el mismo que el indicado en el JNLP. Por otro lado la referencia al proyecto debería estar ARRIBA de todo, ya que se supone que completará lo proporcionado por la plataforma, por lo que su código debería tener prioridad sobre el de la plataforma.

Creación de un nuevo expediente simple

En primer lugar crearemos un nuevo tipo de expediente muy sencillo desde la propia aplicación. Posteriormente se irá complicando.

Sin embargo, antes de entrar de lleno en ello, haremos un breve resumen sobre la arquitectura del proceso de tramitación en Cividas.

Introducción: Modelo de datos



Este esquema es simplemente un resumen de las tablas principales que intervienen en el proceso de tramitación. Se han ignorado tanto las relaciones con otras tablas del sistema, así como todas las posibles tablas que formarían parte del modelo de datos de los distintos tipos de trámites (hemos puesto exclusivamente la tabla de datos por defecto.

Trámites

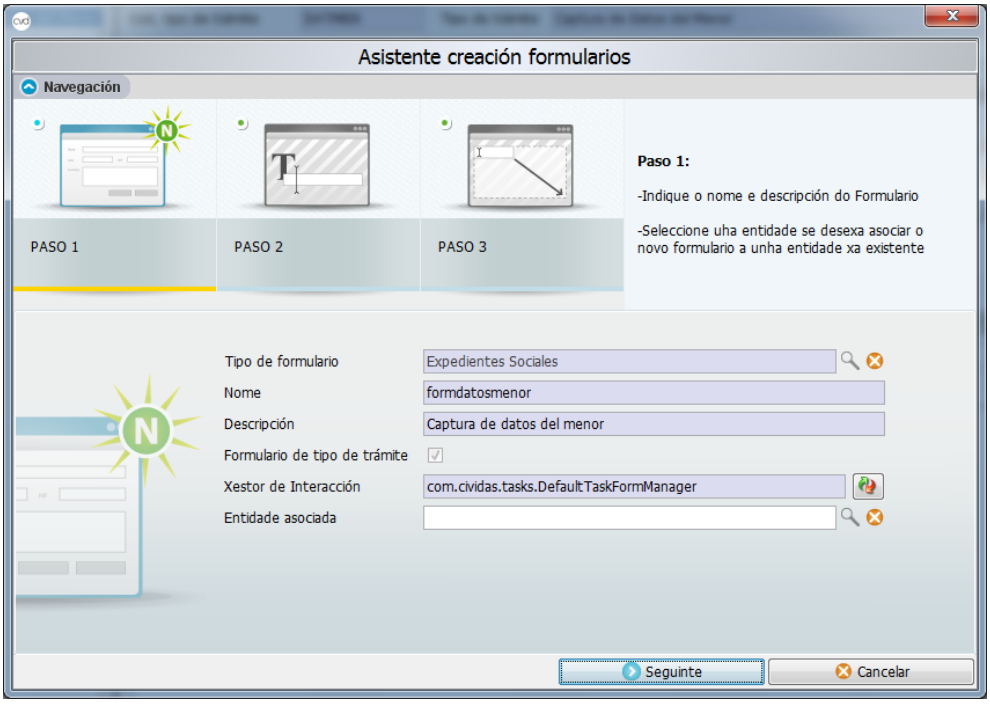
Antes de nada, hay que definir los trámites que van a formar parte de nuestro expediente. En principio sólo tendrá un trámite, por lo que lo damos de alta.

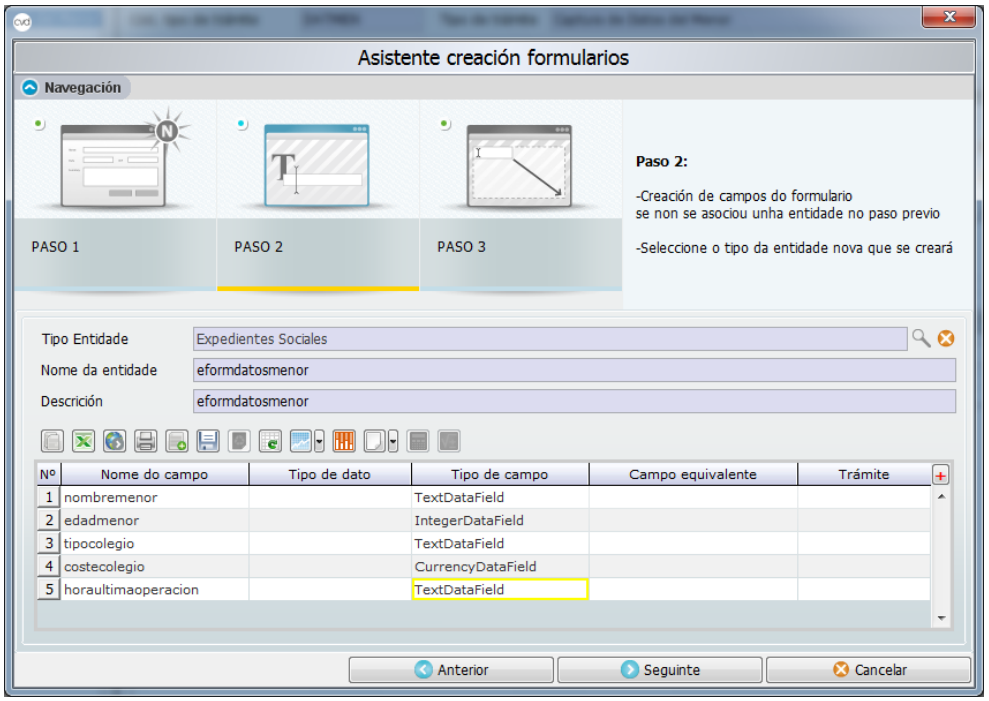
imatia your innovation partner

En el proceso de alta del trámite, tendremos que definir su formulario y su entidad asociada. En este primer ejemplo no tendremos ningún tipo de lógica adicional por lo que en la mayor parte de configuración dejaremos las opciones por defecto.

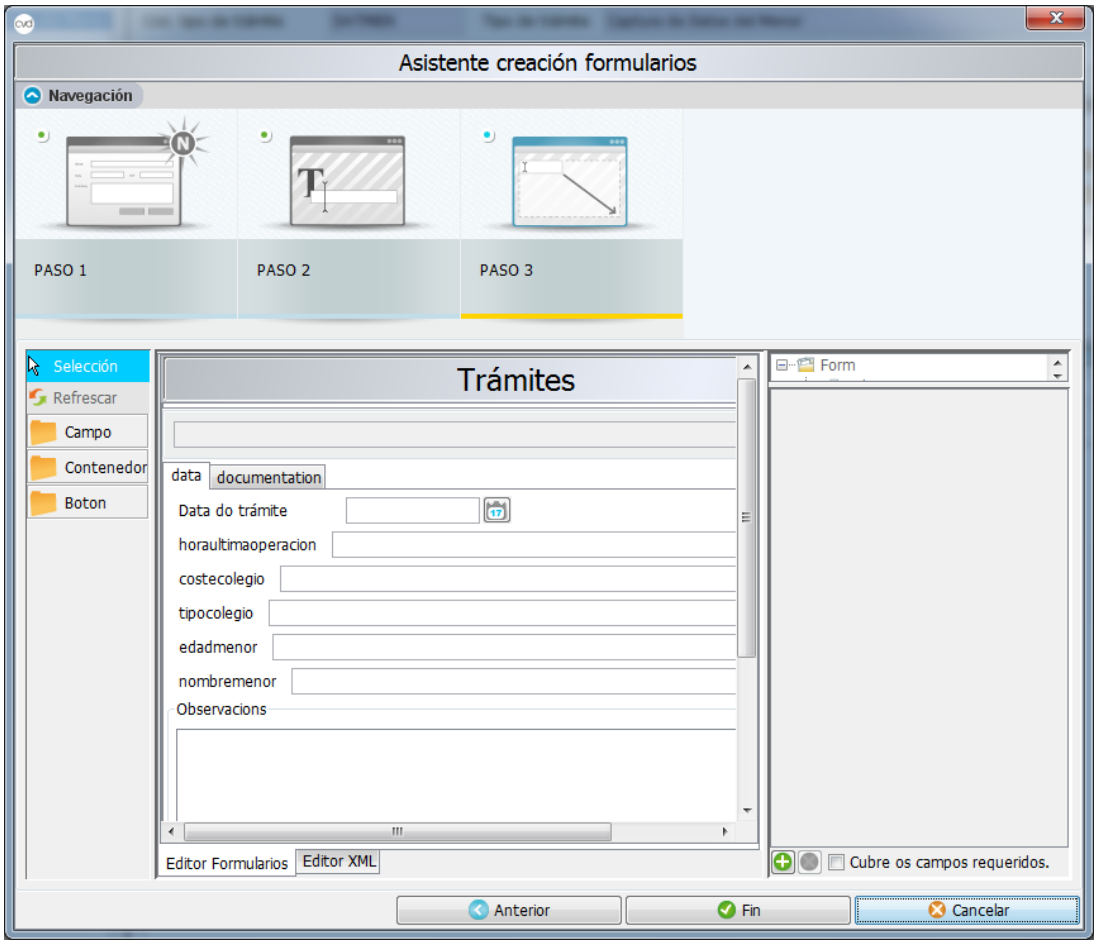
Definimos su entidad con cinco campos:

- Nombre del menor de tipo texto
- Edad de tipo entero
- Tipo de colegio de tipo texto
- Coste del colegio de tipo moneda
- Hora de la última operación de tipo texto



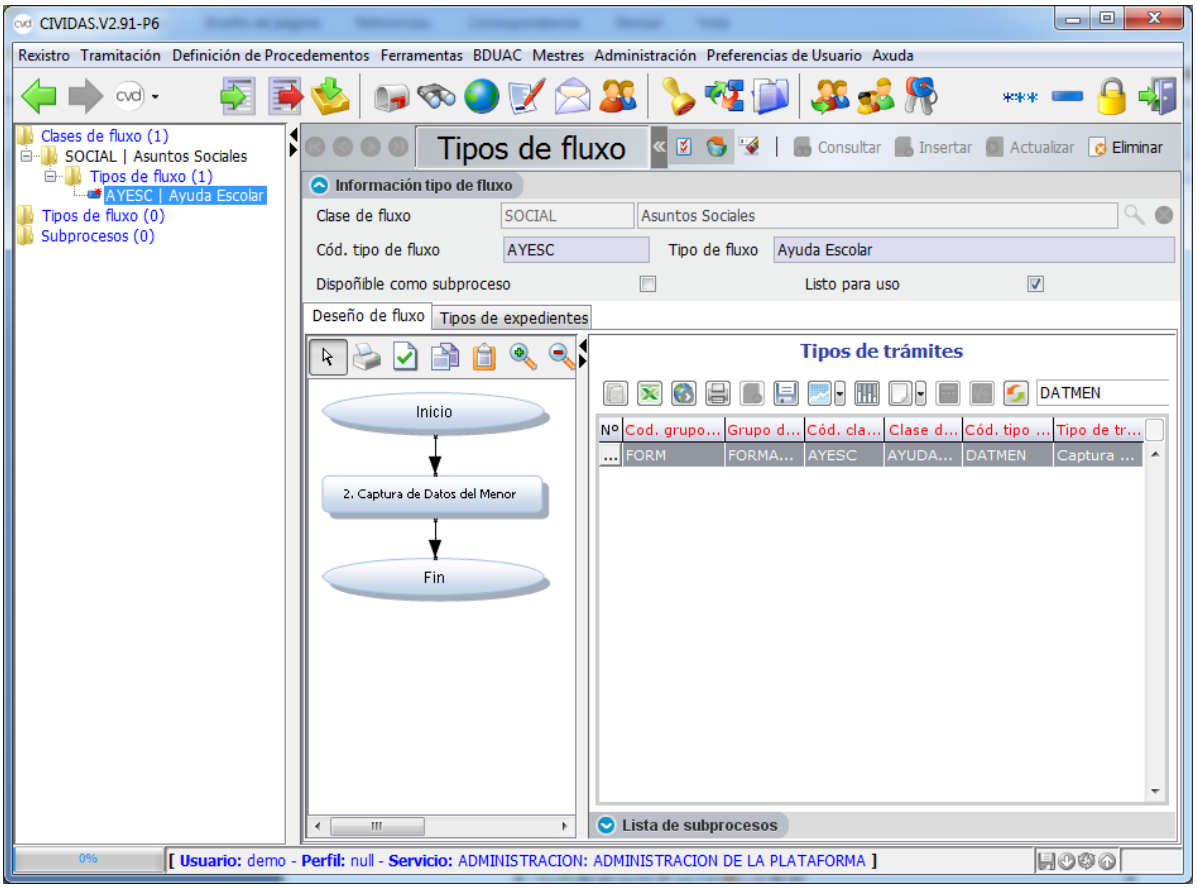


De momento dejaremos los campos tal cual nos lo propone el editor.



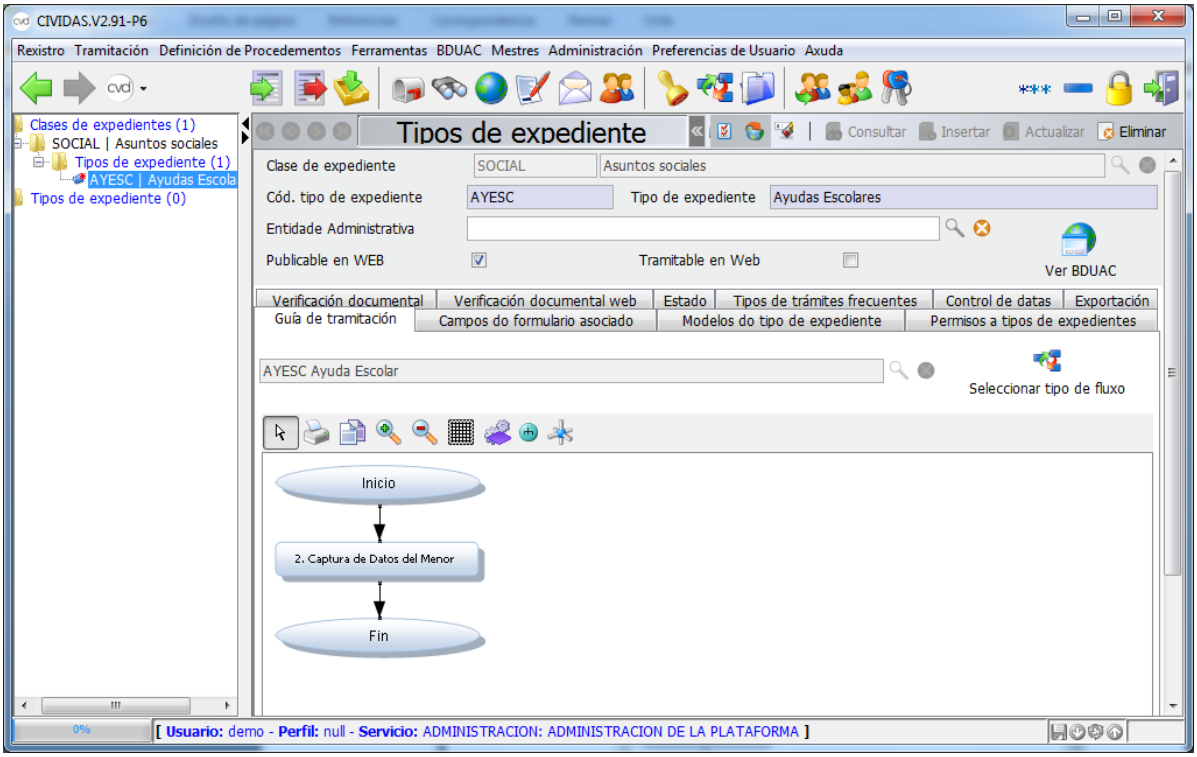
Flujo de Tramitación

Con este trámite ya creado y con su formulario correctamente asociado, definimos el flujo de tramitación. Como estamos ante un ejemplo simple, este flujo tendrá unicamente el trámite que acabamos de definir.



Tipo de Expediente

Utilizaremos este flujo para nuestro tipo de expediente:

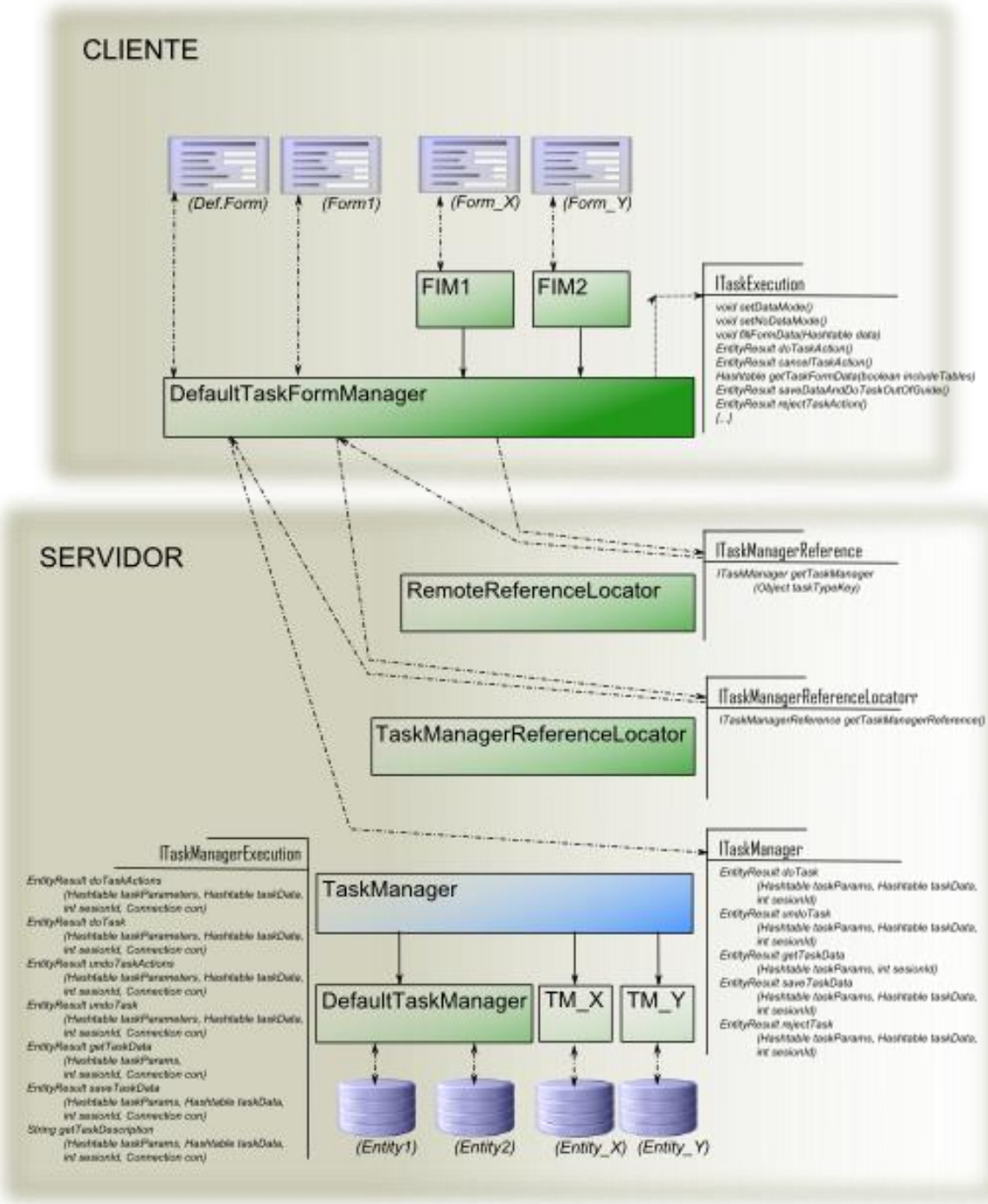


imatia your innovation partner

Incorporación de un trámite programado simple

Introducción: Arquitectura de tramitación

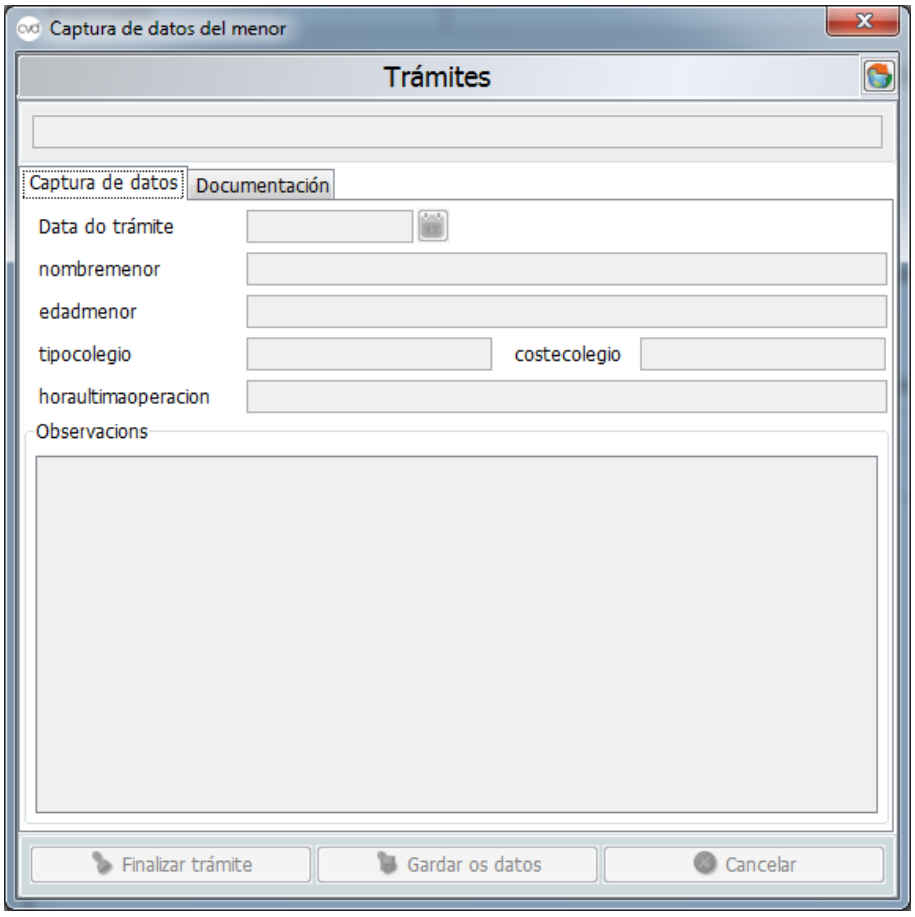
Antes de meternos de lleno en la programación de los elementos que forman parte del trámite, haremos un breve resumen sobre la arquitectura del proceso de tramitación:



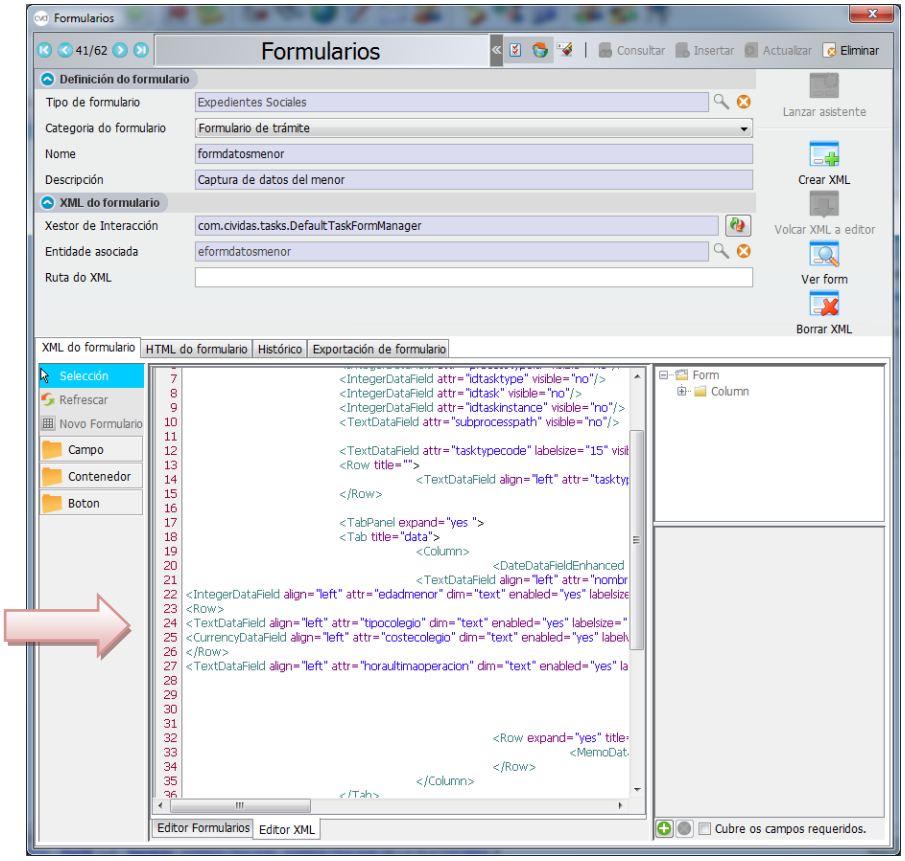
Formulario

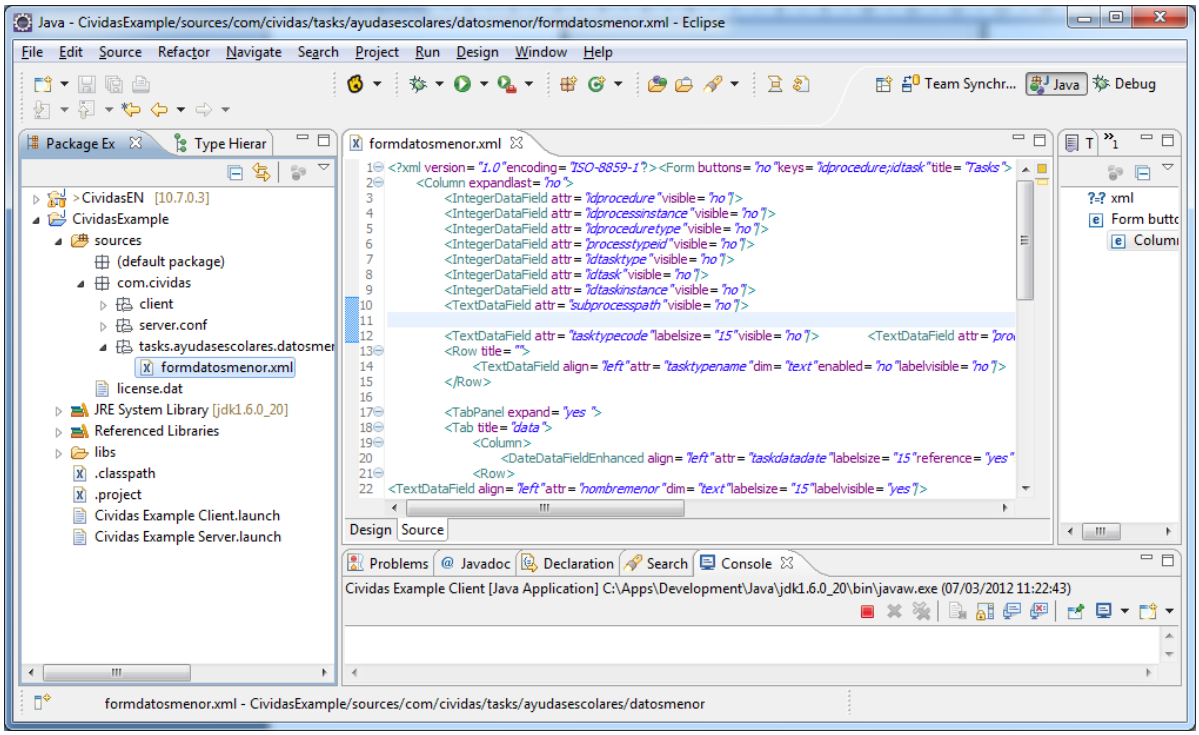
Lo primero que haremos será colocar los campos del formulario. Podemos hacerlo desde la aplicación utilizando para ello el editor:

imatia your innovation partner

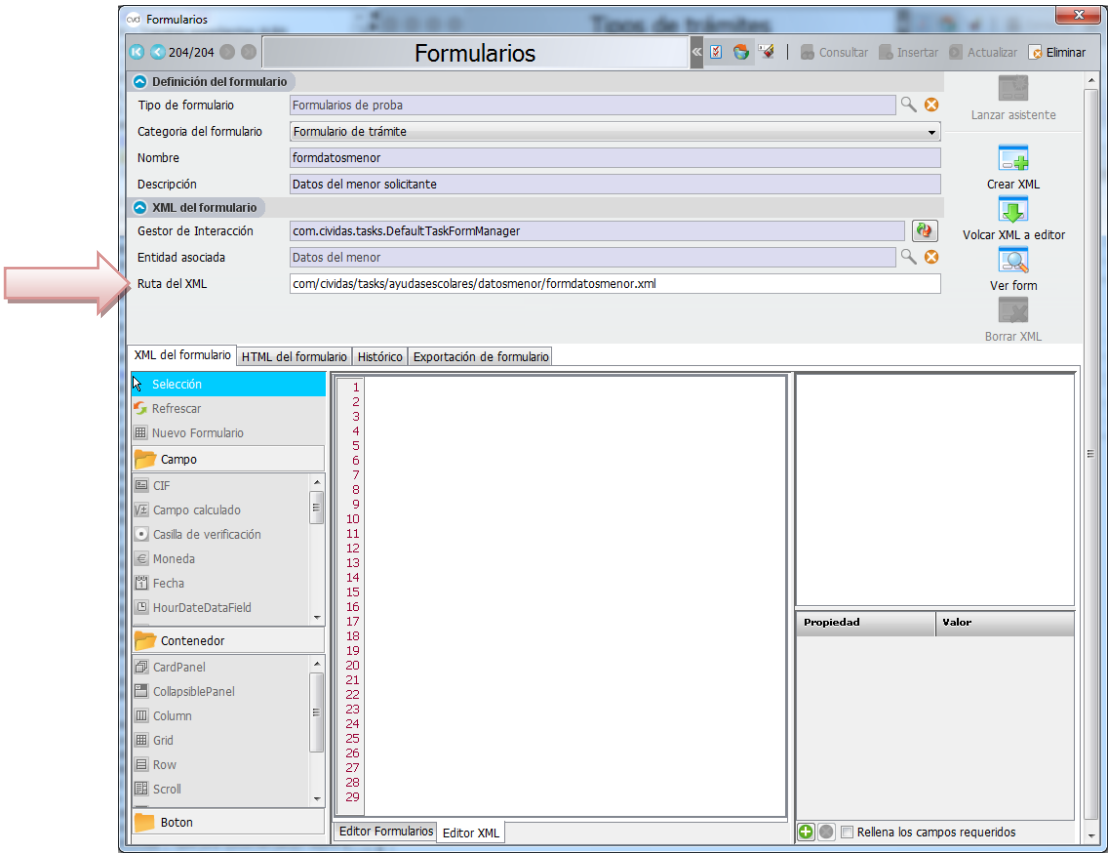


Una vez que lo tengamos podemos "sacarlo" de la base de datos y llevarlo al código. Para ello copiamos el XML de definición del formulario y lo llevamos a nuestro proyecto:





Tendremos que tener en cuenta que ahora la definición del formulario cambia. Dejará de ser el XML en base de datos para ser el XML en código. Para indicar esto, debemos desde la aplicación Borrar el XML e indicar la ruta del mismo en el código:

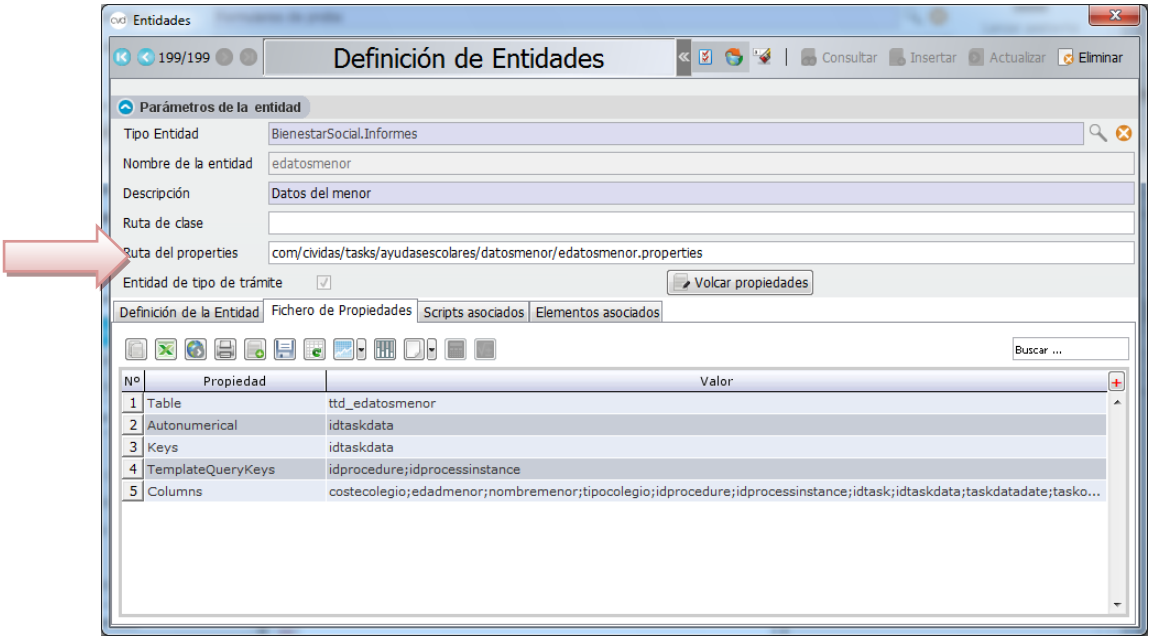


imatia your innovation partner

Entidad

La entidad también nos la vamos a "traer" al código. Para ello el mecanismo será similar:

- Creamos un properties en código.
- Llevamos los atributos que originalmente estaban en base de datos.
- Indicamos la ruta del properties de la entidad en código.



Lógica del trámite: Gestor de Interacción VS Gestor de Tramitación

Tenemos en el trámite dos puntos en los que añadir lógica particularizada: El Gestor de Interacción y el Gestor de Tramitación. ¿Cuándo añadir la lógica en un punto o en otro? Depende.

A grandes rasgos el Gestor de Interacción incorpora la lógica de cliente, que debe ser lo mínima posible y limitarse, digamos a cuestiones de interfaz gráfica, esto es:

- Operaciones básicas sobre los campos: Activar/Desactivar, Mostrar/Ocultar, comprobar valores, volcar valores, etc.
- Otras funcionalidades del trámite que requieran la intervención del usuario, p.e. nuevos botones

El Gestor de Tramitación incorpora lógica de negocio del lado del servidor. Todas las operaciones con cierta enjundia deberían implementarse en este punto, ya que generalmente los equipos servidores son más potentes y están mejor dotados para realizar operaciones más pesadas. Las integraciones son un buen ejemplo de funcionalidades que, a toda costa, deberían implementarse en el lado del servidor.

Gestor de Interacción. Descripción

Como se comentó anteriormente, este primer trámite hasta el momento sólo tiene la lógica por defecto. Lo que vamos a hacer ahora es incorporar algunas comprobaciones de cliente.

Para ello crearemos en código nuestro gestor de interacción. Es imprescindible que este gestor extienda del gestor base:

com.cividas.tasks.DefaultTaskFormManager

para no perder nada de la lógica estándar del trámite.

Este gestor base implementa una serie de interfaces con un importante conjunto de métodos para distintas funcionalides. No creemos adecuado en este punto empezar a describir uno por uno todos estos métodos, dado que seguramente sea más gráfico y comprensible analizar la sucesión de

acontecimientos en las operaciones básicas del trámite: Abrir el trámite, Finalizar el trámite, Guardar datos, Cancelar:

Abrir el trámite

Un trámite puede tener distintos orígenes: Una tarea del WF, la tabla de trámites pendientes, el histórico, la tabla de trámites libres. Cada uno de estos origenes proporciona una información determinada distinta que, entre otras cosas, nos ayudará a saber como tenemos que abrirlo.

Lo que se hace en origen para abrir un trámite es llamar al método:

```
public void loadFormManager(Hashtable taskFormParameters, IMassiveDoTask parentClass,
                            Component parentComponent) throws Exception
```

por lo que este es el primer punto de entrada a la lógica de cliente del trámite. Dentro de este método se hace lo siguiente:

1/ Inicialización de parámetros.

Limpiar los parámetros para reniciarlos:

```
public void clearTaskParameters();
```

2/ Carga del contexto

Se carga el contexto del trámite con los parámetros que nos llegan directamente del origen de la llamada.

```
public void loadTaskContext (taskFormParameters);
```

Este método a su vez realiza una llamada remota que recupera la información básica del trámite (si ya ha sido ejecutado, si el usuario tiene permisos sobre él, etc., así como los datos del mismo.

3/ Modo inicial

Se establece el modo inicial de apertura del trámite

```
public void setInitialMode()
```

3.1/ Discrimina si el trámite es nuevo o ya tiene datos

3.1a/ Si ya tiene datos, se llama a:

```
public void setDataMode()
```

que se encarga de rellenar el formulario con los valores por defecto procedentes de la parametrización particular y con los procedentes del contexto antes mencionado.

3.1b/ Si no tiene datos se llama a:

```
public void setNoDataMode()
```

que se encarga de borrar el formulario y rellenarlo con los valores por defecto.

3.2/ Comprueba si tenemos ID de trámite o no y en función de eso llama a sendos métodos que efectúa las acciones implementadas para ambos casos

```
public void checkTaskIDActions()
```

3.2a/ Método llamado en la primera ejecución del trámite:

```
public void setStatusFirstExecution () throws Exception
```

3.2b/ Método llamado en las siguientes ejecuciones del trámite:

```
public void setStatusAlreadyExecuted () throws Exception
```

3.3/ Refresca todas las tablas del formulario. Si se quiere excluir alguna de ellas habría que sobrecargar el método getExcludedTablesToGenericRefresh() que devuelve un Vector con los atributos de las tablas que NO se tienen que refrescar por defecto.

3.4/ Deshabilita los campos procedentes del modelo del expediente. En la consulta de datos del trámite hay un conjunto de valores que no son del trámite sino del expediente, y que se devuelven por comodidad a la hora de mostrarlos en un formulario de trámite. Estos campos tienen el prefijo "procedure$nombre_del_campo" y NO son modificables bajo ninguna circunstancia desde el trámite.

3.5/ Comprueba si se trata de una ejecución masiva

4/ Apertura en modo lectura

Se realiza la comprobación de si el formulario debe ser abierto en modo lectura y se efectúan las modificaciones necesarias en el comportamiento del mismo si procede:

```
public boolean openInReadOnlyMode()
```

En la versión Cividas.V2.91, la única condición para abrir el formulario en modo sólo lectura era que el expediente estuviese archivado. En la versión Cividas.V2.95 ya se permite especificar permisos de lectura y escritura por grupo de usuario. En cualquier caso, se trata de un método que se podría extender para:

- Establecer condiciones alternativas para determinar si el formulario se abre en modo sólo lectura o no
- Implementar acciones alternativas en el caso de que el formulario esté en modo sólo lectura.

5/ Captura del token

Para evitar el acceso simultáneo a un mismo trámite, el usuario que lo abre captura un "token" que provoca que el trámite esté deshabilitado para cualquier tipo de acceso mientras el usuario lo tenga abierto.

```
public Object catchTaskToken(Object idtask, Object singlekey, Object idprocedure) throws
Exception
```

6/ Visibilidad del formulario

A continuación se hace visible el formulario. Se trata de una operación que, por sus características, no es extensible.

7/ Liberación del token

Una vez se han realizado las operaciones necesarias con el trámite se regresa a este punto para liberar el "token" capturado previamente:

```
public void freeTaskToken (Object idtasktoken) throws Exception
```

Finalizar el trámite

La finalización del trámite se realiza a través de un botón con una clave determinada que tiene un escuchador particular que acaba llamando a un método concreto del gestor de interacción básico. Cualquiera de estos elementos pueden ser sustituídos o extendidos. Desgranándolo con más detalle, tenemos:

Botón de finalización del trámite:

```
Button bDoTask = managedForm.getButton(BUTTON_DO_TASK);
```

Escuchador de finalización del trámite:

```
DoTaskListener doTaskListener = new DoTaskListener(managedForm);
```

Acción que se ejecuta desde el escuchador:

```
public EntityResult doTaskAction() throws Exception
```

En general, a la hora de extender la finalización del trámite lo conveniente es centrarse exclusivamente en este último elemento, ya que hacerlo con el botón o el escuchador complica un poco más la lógica y no aporta demasiado.

Dentro de la extensión del método hay 4 puntos fundamentales:

1/ Previo a la llamada remota

En este punto se puede hacer uso del método:

```
public void appendToTaskParameters(Object key, Object value, boolean overwrite)
```

para añadir nuevos parámetros

y también de:

```
public Hashtable getTaskFormData(boolean includeTables) throws Exception
```

para añadir nuevos valores a los datos del trámite

2/ Llamada remota

La extensión en este punto se llevaría al servidor y se explicará en el siguiente apartado

3/ Posterior a una operación correcta

Se puede extender el método:

```
public void postTaskDoneActions()
```

para indicar las acciones a tomar tras una finalización de trámite correcta

4/ Posterior a una operación independientemente del resultado

Si lo que se desea es efectuar una operación tras la finalización del trámite, sea cual sea el resultado del mismo, el método a extender sería:

```
public void postTaskActions() throws Exception
```

Guardar datos

La operación de guardar datos es totalmente equivalente a la de finalizar el trámite salvo por el detalle de que el botón, el escuchador y el método son distintos, lógicamente. Así, tenemos:

```
Button bSaveData = managedForm.getButton(BUTTON_SAVE_DATA);
```

```
SaveTaskDataActionListener saveDataActionListener = new SaveTaskDataActionListener
        (managedForm);
```

y el método a extender:

```
public EntityResult saveDataAndDoTaskOutOfGuide() throws Exception
```

Los putnos de extensión son los mismos que en el caso anterior, con la salvedad de que no hay un método propio para el retorno tras una operación correcta (serían, en todo caso, los ya mencionados loadTaskContext y setInitialMode para hacer un "refresco" del formulario. Así tenemos:

1/ Previo a la llamada remota

En este punto se puede hacer uso del método:

```
public void appendToTaskParameters(Object key, Object value, boolean overwrite)
```

para añadir nuevos parámetros

y también de:

```
public Hashtable getTaskFormData(boolean includeTables) throws Exception
```

para añadir nuevos valores a los datos del trámite

2/ Llamada remota

La extensión en este punto se llevaría al servidor y se explicará en el siguiente apartado

3/ Posterior a una operación independientemente del resultado

Si lo que se desea es efectuar una operación tras guardar los datos del trámite, teniendo en cuenta que previamente ya se ha "reseteado" el formulario, tendríamos que sobrecargar lo siguiente:

```
public void postTaskSaveActions() throws Exception
```

Cancelar

Por el momento no entramos en este punto, ya que su importancia es menor.

Gestor de Interacción. Ejemplo

Problema 1

El tipo de colegio debe ser un combo y no un valor de texto libre

Solución 1

[…]

Problema 2

El ejercicio consiste en crear una etiqueta que se muestra cuando sólo cuando se ejecuta el trámite por primera vez.

Solución 2

[…]

Gestor de Tramitación. Descripción

Como en el caso de los formularios, para la implementación de la lógica de servidor también se dispone de una clase base de la se recomienda extender, no ya sólo para disponer de la funcionalidad básica de tramitación sino para tener implementaciones por defecto de los métodos impuestos por las interfaces necesarias.

Esta clase es:

```
com.cividas.tasks.TaskManager
```

Al igual que en el caso anterior, se estima que es más adecuado describir el proceso de tramitación que cada uno de los métodos de las distintas interfaces implementadas. Cabe resaltar que la mayor parte de los métodos que se van a mencionar de aquí en adelante tienen su versión con parámetro de conexión y sin él. En caso de sobrecarga, se recomienda siempre hacerlo con el método con conexión, aunque, claro está, para todo hay excepciones.

Recuperar el contexto / los datos del trámite:

Hay un método fundamental en este sentido:

```
public TaskContext getTaskContext(Hashtable taskParameters, int sessionID,Connection con)
        throws Exception
```

Ya se ha hablado de este método en el apartado anterior. Lo que hace en su interior es lo siguiente:

1/ Comprobación de permisos

Comprobar si el usuario tiene permisos de ejecución (Cividas.V2.91 o de ejecución y lectura (Cividas.V2.95):

- Método deprecado (en Cividas.V2.95) de comprobación de permisos de ejecución:

```
protected boolean hasExecutePermission(Object idtasktype, int sessionID,Connection con)
        throws Exception
```

- Método de comprobación de permisos generales (lectura y escritura):

protected Hashtable getAccessConfiguration(Object idtasktype, int sessionID, Connection con) throws Exception

2/ Datos de la instancia del trámite

Obtener los datos a nivel de instancia del trámite:

public Hashtable getTaskInfo (Hashtable tParameters, int sessionID,Connection con) throws Exception

es decir, IDs de relación con otros elementos, así como los datos de la instancia del trámite (si es que existe).

3/ ID de la tabla de datos

Obtener el ID de los datos del trámite en la tabla correspondiente:

public Object getIDTaskData (Hashtable parameters, int sessionID,Connection con) throws Exception

4/ Datos del trámite

Obtener los datos del trámite en dicha tabla

public EntityResult getTaskData(Hashtable taskParams, int sessionID, Connection con) throws Exception

Realmente se hacen tres consultas para alimentar el único EntityResult que se devuelve:

- Contra la tabla de datos del trámite
- Contra la tabla de datos básicos del expediente (agregando el prefijo "procedure$" para idenficarlos).
- Contra la tabla del modelo de datos del expediente (agregando el prefijo "procedure$" para idenficarlos).

Finalizar un trámite

El método principal es:

public EntityResult doTask (Hashtable taskParameters, Hashtable taskData, int sessionID, Connection con) throws Exception

sin embargo, NO es este el método que se recomienda extender para realizar operaciones alternativas en un trámite. En el proceso de descripción de su funcionamiento interno se intentará mostrar por qué y cuál es el mejor punto para extender.

En este método se hace lo siguiente:

1/ Comprobación de ejecución a través del WF

Se efectúan las comprobaciones y modificaciones necesarias en los parámetros de entrada para preparar la posterior interacción del trámite con el WF.

public void manageTaskIntoGuide(Hashtable tParams, int sessionId,Connection con) throws Exception

2 / Comprobación de los permisos de lectura/ejecución:

…con las consideraciones hechas sobre las modificaciones en la versión Cividas.V2.95

protected Hashtable getAccessConfiguration(Object idtasktype, int sessionID, Connection con) throws Exception

Puede considerarse este método redundante con respecto a la llamada que se hace en la obtención del contexto. Sin embargo hay que tener en cuenta que los trámite se pueden ejecutar ad hoc sin necesidad de pasar por sus formularios, por lo que es necesario realizar la comprobación de permisos siempre que llegue una petición de ejecución

3/ Preparación de los parámetros del trámite

Aunque el grueso de parámetros nos llega en la propia petición, existen una serie de campos que se añaden o modifican tras la petición. Por ejemplo:

- El usuario que ha ejecutado el trámite
- La fecha de ejecución

- El ID dentro del histórico de formularios al que va a quedar asociada la instancia actual.
- etc.

public void setTaskParameters (Hashtable taskParams, int sessionID, Connection con) throws Exception

4/ Gestión de la interacción con las tablas de tramitación

Se efectúan las operaciones necesarias sobre las tablas de tramitación:

- Tabla de instancias de trámites
- Tabla de histórico de ejecuciones.

public EntityResult manageTaskTablesActions(Hashtable taskParams, Hashtable taskData, int sessionID, Connection con) throws Exception

5/ Almacenamiento de los datos del trámite

Teniendo en cuenta si se trata de una inserción de nuevos datos o una actualización de los datos ya existentes.

public EntityResult manageTaskData (Hashtable taskParams, Hashtable taskFormData, int sessionID, Connection con) throws Exception

6/ Ejecución de las tareas propias del trámite cuando éste se finaliza

Este es el método principal a la hora de sobrecargar un trámite. El 80% de la lógica particular de los trámites debería estar localizada aquí.

public EntityResult doTaskActions(Hashtable taskParameters, Hashtable taskData, int sessionID, Connection con) throws Exception

Este método no tiene ninguna implementación base, es decir, por defecto "no hace nada", por lo que se deja total libertad al programador para que monte en este punto las funcionalidades que considere oportunas.

Por otro la localización del método dentro del proceso de finalización del trámite, aparentemente "en medio de todo" tiene su razón de ser:

- ¿Por qué no se incluye antes? Porque es en este punto en el que disponemos de toda la información necesaria sobre el trámite ejecutado de las tablas de base de datos (su ID de la tabla de trámites, de la tabla del histórico, de la tabla de datos, etc)
- ¿Por qué no se incluye después? Como la mayor parte de la lógica pesada estará aquí localizada, parece lógico pensar que la mayor parte de los errores (deseados o no deseados) se producirán en esta parte. No tendría sentido, pues, ejecutar todas las acciones restantes (que en la inmensa mayoría de los casos no producirán ningún error) antes de ésta.

7/ Ejecución de las tareas propias del trámite cuando éste no se finaliza

La condición para que un trámite conste como finalizado es que entre los parámetros de invocación se encuentre la clave "taskend" y que su valor sea 1. Esta condición ya se incorpora por defecto en las llamadas estándar que se hacen desde el cliente y se incorpora a todas las llamadas del servidor que no indique explícitamente lo contrario.

Es precisamente en estos casos ("taskend"!=1) en los que NO se va a ejecutar el método mencionado anteriormente, ya que, a efectos funcionales, no se "está finalizando" el trámite, sino que se está dejando un registro de ejecución histórico,

7/ Gestión de las solicitudes de trámites

Cuando se cumplimenta una solicitud de trámite se debe marcar dicha solicitud como finalizada.

public EntityResult manageTaskRequests (Hashtable taskParams, int sesionId, Connection con) throws Exception

8/ Gestión de la interacción con el trámite de solicitud

No es el mismo caso que el anterior, aunque pueda parecerlo. En este método se comprueba si el trámite que estamos ejecutando procede del trámite de solicitud de trámite y en dicho caso, actualiza el estado o los datos del mismo.

public EntityResult manageRequestTask(Hashtable taskParams, Hashtable taskFormData, int sessionID, Connection con) throws Exception

9/ Gestión de los cambios de estado del expediente derivados del trámite

Se tiene para ello en cuenta la definición del tipo de trámite (a que estado debe evolucionar el expediente tras su finalización) además de si hay que dejar constancia de este cambio en el histórico de tramitación con un trámite de cambio de estado.

```
public EntityResult executeProcedureStateChange (Hashtable taskParameters, Hashtable
        taskData, int sessionID, Connection con) throws Exception
```

10/ Gestión de la interacción con el WF

Se efectúan las operaciones de interacción con el WF: evolucionarlo, marcar una tarea como ejecutada fuera de orden, etc.

```
public void manageTaskGuideActions (Hashtable taskGuideParameters , int sessionID,
        Connection con) throws Exception
```

11/ Obtención y actualización del campo de descripción de la instancia del trámite.

Al tratarse de un valor que puede depender de los datos almacenados en la tabla del trámite, no se puede obtener antes. Es el valor que se mostrará en la tabla de trámites de la pantalla del expediente. Por defecto, se muestra la columna especificada en la definición del tipo de trámite, pero podría extenderse para que esta descripción pudiese tener el origen que se desee.

```
public void manageTaskDescription (Hashtable taskParameters, Hashtable taskData, int
        sessionID, Connection con) throws Exception
```

12/ Envío de avisos.

Tras la ejecución de un trámite, si hay algún destinatario configuraro a tal efecto, se envía un aviso a dicho destinatario a través de este método:

```
public EntityResult manageTaskNotices (Hashtable taskparameters, Hashtable taskFormData,
        int sessionID, Connection con) throws Exception
```

Sin embargo, la mejor forma de particularizar el envío de un aviso no es sobrecargando dicho método sino haciendo uso de los siguientes:

```
public Object getidserviceNoticeDestiny(Hashtable taskparameters,Hashtable taskFormData,int
        sessionID, Connection con)throws Exception
```

Con este método podemos configurar la obtención del ID de la unidad de servicio (usuario) al que se va a enviar el aviso una vez finalizado el trámite.

```
public Hashtable builtMessageNoticeData(Hashtable taskparameters, Hashtable taskFormData,
        int sessionID, Connection con) throws Exception
```

Con este método se pueden configurar que información se le va a enviar al sistema de avisos para que construya el aviso.

Guardar los datos del trámite

El método a través del que se guardan los datos del trámite es el siguiente:

```
public EntityResult saveTaskData(Hashtable saveTaskParams, Hashtable taskData, int
        sessionID, Connection con) throws Exception
```

En este caso se trata de un proceso mucho más sencillo que el de finalización ya que toda la interacción con tablas, WF, etc se ignora. Simplemente se insertan o actualizan los datos en la entidad asociada al trámite.

Generalmente si es necesario sobrecargar algo en este proceso sería recomendable hacerlo en la entidad correspondiente más que en el gestor de tramitación.

Gestor de Tramitación. Ejemplo

Problema

El ejercicio a realizar para ilustrar un ejemplo de programación del gestor de tramitación se divide en tres partes:

1. Cuando se finalice el trámite (y sólo cuando se finalice), agregar al asunto del expediente el texto: "Se han dado de alta los datos del menor". Si se finaliza n veces, el texto aparecerá n veces.
2. El mismo caso que el anterior pero no sólo cuando se finalice sino cada vez que se graben los datos.
3. Si el trámite se hace a través de la guía de tramitación, lo indicaremos en las observaciones del propio trámite, en cualquier caso. El texto será "Se ha guardado a través de la guía".

Solución apdo. 1

[…]

Solución apdo 2.

[…]

Solución apdo. 3

[…]

¿Código o BBDD? Recomendaciones

En general no existen unas condiciones fijas y claras por las que un elemento se deba almacenar en base de datos o en código. A modo de recomendaciones se podría comentar lo siguiente, aplicado tanto a formularios como entidades:

- Si el trámite es muy variable: Es mejor guardarlo en BBDD, ya que nos permite realizar modificaciones sobre el de forma rápida y sin necesidad de redespliegues.
- Si el trámite es complejo y forma parte de un módulo más completo: Es mejor almacenarlo en código ya que nos permite un control más estricto sobre el mismo.
- Hay que tener en cuenta que una vez que se elige una de las dos opciones, no estamos obligados a continuar con ella hasta el final. Por ejemplo:
    - Estamos en fase de consultoría definiendo un trámite que sabemos que va a tener bastante lógica. Para poder tener una interacción flexible con el usuario cliente en esta fase de definición creamos el formulario en BBDD y lo vamos modificando en caliente. Así podemos enseñarle como se va adaptando a sus necesidades (al menos en términos de aspecto) de forma rápida. Una vez cerrada o semi-cerrada esta fase de definición ya nos podríamos llevar el XML de definición al código y seguir desde ese punto.
    - Hemos cerrado la definición de un trámite complejo en código. Podemos hacer pequeñas particularizaciones en BBDD partiendo del XML definido en código.

Gestor de interacción o Gestor de tramitación? Recomendaciones

Vamos a debatir sobre una serie de casos para analizar que sería lo más conveniente, si incorporar la lógica al cliente o al servidor. Lógicamente, en la mayor parte de las ocasiones habrá que programar en ambos puntos, pero ahora se trata de comentar casos muy sencillos y puntuales:

Problema 1:

Necesitamos un nuevo campo que le aplica un porcentaje al coste del colegio para obtener dos valores:

- Lo que tiene que pagar el Ayuntamiento: 30% del coste del colegio
- Lo que tiene que pagar el Gobierno Autónomo: 70% del coste total

Estos campos deben mostrarse en el formulario pero no ser editables. También deberían guardarse en BBDD para alimentar plantillas e informes

Problema 2:

Necesitamos una opción para validar la letra de un nuevo campo que va a contener un DNI.

Problema 3:

Necesitamos evitar ingresar edades mayores de 16 años

Problema 4:

Necesitamos cambiar el tipo de expediente en función de la edad del menor:

- Si es menor de 10 años: Ayuda Infantil
- Si es mayor de 10 años: Ayuda juvenil

Incorporación de un trámite programado avanzado

Integraciones como parte del trámite

Problema

Vamos a completar ahora el trámite con la llamada a un WebService público sencillo. Por ejemplo, tenemos este disponible que nos devuelve la hora:

http://dotnet.jku.at/buch/samples/7/simple/TimeService1.asmx?WSDL

El ejercicio que vamos a hacer ahora es actualizar la hora de última ejecución del trámite con la hora proporcionada por ese servicio cada vez que se guardan cambios.

Para ello no nos vamos a lanzar a sobrecargar el método doTaskActions con la lógica de llamada al WebService sino que lo vamos a intentar generalizar un poco más incluyendo un nuevo elemento: Las referencias remotas.

Referencias Remotas

Las referencias remotas son una herramienta que nos proporciona Ontimize para disponer de "servicios" configurables en el servidor. Es un elemento similar a las entidades pero que no tiene por que tener ninguna relación con la Base de Datos y dispone de una configuración mucho más potente que las entidades.

En la versión actual Cividas.V2.91 las referencias remotas aún se tienen que definir al "modo Ontimize" es decir, editando un archivo de propiedades en la carpeta de instalación del servidor, más concretamente:

com/cividas/server/conf/remotereferences.xml

aunque la ruta de este archivo es parametrizable.

En la próxima versión, Cividas.V2.95, la gestión de las referencias remotas se comparte entre el archivo antes mencionado y la base de datos, de forma que en la aplicación existirá un maestro de referencias remotas que se podrá configurar totalmente en Base de Datos (excepto la lógica de negocio asociada, claro está).

Se dispone de un manual de utilización de las referencias remotas, tanto desde el punto de vista de Ontimize como teniendo en cuenta los agregados que proporciona Cividas, encaminados a una más fácil parametrización dinámica.

Problema

Vamos a generalizar la llamada al WebService anterior para que sea una referencia remota que pueda ser utilizada de forma cómoda en cualquier otro punto de la aplicación.

Solución

[…]

Uso de las parametrizaciones Cividas

Problema

Vamos a parametrizar la llamada al WebService de forma que:

- Se parametrice la URL de invocación del WS.
- Se parametrice el prefijo a añadir delante de la hora.
- Se parametrice si se aplica el prefijo o no.

Solución

[…]

Envío de avisos y correos

Problema

Vamos a enviar dos avisos indicando que se ha ejecutado la tarea:

- Un aviso interno a la persona que tiene el expediente en su buzón.
- Un correo electrónico a una dirección especificada.

Solución

[…]

Otras utilidades de la plataforma

Extensiones de elementos: Ventajas y riesgos

Al estar la plataforma Cividas desarrollada sobre el framework Ontimize, una de las utilidades de las que se dispone es la extensión de elementos, esto es:

- Formularios
- Fichero de definición del menú de la aplicación
- Fichero de definición de la barra de botones de la aplicación
- Fichero de definición de la aplicación cliente
- Ficheros de propiedades de las entidades

Su modo de empleo (detallado para cada uno de los elementos extendibles) se define en un manual centrado en este aspecto que ofrece Ontimize.

Ahora bien, al ser Cividas, no sólo una plataforma sino también un producto que evoluciona con el tiempo, hay que tener un especial cuidado a la hora de hacer cierto tipo de extensiones en el siguiente sentido:

- En una determinada versión la plataforma proporciona un menú determinado, una botonera determinada y un conjunto de formularios determinado
- Un desarrollador modifica, p.e. uno de los formularios para añadir un campo. Se preocupará por que el nuevo formulario tenga un aspecto apropiado en función de la versión que la plataforma ofrece.
- En la siguiente versión la plataforma ha hecho evolucionar dicho formulario para incorporar nuevas funcionalidades. Esto probablemente implique alteraciones en el aspecto.

- El formulario extendido por el desarrollador puede, en este punto, verse alterado de forma significativa y no siempre positivamente. Un problema mayor se encontraría si, p.e. el desarrollador agrega un campo y la plataforma en una versión posterior agrega ese mismo campo.
- Por tanto, el desarrollador en cada actualización de la plataforma tendría que revisar TODAS sus extensiones para comprobar que ninguna de ellas haya sido alterada.
- Desde la plataforma NO se van a tener en cuenta las extensiones que los desarrolladores apliquen a partir de lo ofrecido por la misma. Se garantiza, eso sí, la compatibilidad hacia atrás del código desarrollado, pero si es necesario, pongamos por caso, reordenar la pantalla de registro de entrada, NO se va a tener en cuenta si en algún proyecto se ha extendido dicha pantalla.

Siguiendo en esta línea argumental, en los casos en que se necesite alguna modificación sobre alguna de las pantallas de la aplicación se recomienda reportar esa necesidas para evaluar la opción de incorporarla al producto. SÓLO EN CASOS MUY PARTICULARES se deberían extender formularios de producto (se entiende por formularios de producto todos los que no están relacionados con los trámites).

Extensiones de menús y pantallas

A pesar de todo lo comentado en el apartado anterior, vamos a hacer un pequeño ejemplo que incluya la extensión del menú y de un formulario. Lo que vamos a hacer es partir de un formulario sencillo y añadirle un botón que consulte la hora del WS que definimos anteriormente. Utilizaremos el formulario de edición de las entidades administrativas para tal efecto.

Problema

En las entidades administrativas queremos un botón que vuelque la hora obtenida del WS al campo de observaciones. Podríamos pensar en: "lo razonable sería meterlo en un campo nuevo y extender la entidad también". Sin embargo, en el caso de entidades, la extensión es más crítica si cabe. Pongamos que añadimos un nuevo campo. Tendríamos que:

- Añadirlo a la tabla en BBDD
- Añadirlo a la vista en BBDD
- Añadirlo al properties de la entidad mediante extensión

Ahora pongamos que en la siguiente versión Cividas añade otro campo (distinto). Tendremos:

- Script de agregado del campo a la tabla en BBDD: Ningún problema.
- Script de modificación de la vista asociada: Se pisará el campo agregado en la extensión.
- Properties modificado: Ningún problema.

Por otro lado, queremos que esté localizado en una posición distinta dentro del menú.

Solución

[…]

Integraciones con Terceros

Las integraciones con terceros se plantean en Cividas SIEMPRE como un extensión de los orígenes de datos diponibles y NUNCA como una extensión del modelo de datos de interesados de la plataforma, por lo comentado precisamente en el apartado anterior.

Debido a que la implementación de un caso práctico resulta ligeramente compleja por la infraestructura necesaria para que el ejemplo sea los suficientemente significativo, se realizará una descripción semi-teórica partiendo de la documentación disponible sobre la forma de realizar una integración con terceros.

Acciones: Usos en Crons y Controles de Fechas

En este apartado vamos a describir el funcionamiento de los crons y las acciones asociadas a través de un ejemplo.

Definición del cron

Un cron es, a fin de cuentas, equivalente a un proceso que se tiene que despertar en un momento determinado y ejecutar una operación.

Como en otros casos tenemos una parte de configuración y definición dentro de la propia aplicación y otra de implementación del código que ejecute las tareas que necesitemos en cada caso.

En Cividas existe una clase para implementar la acción de Cron por defecto:

com.cividas.server.utils.actions.DefaultCividasCron

esta clase extiende la abstracta proporcionada por Ontimize:

com.ontimize.util.quartz.AbstractTask

y además implementa la interfaz:

com.cividas.server.utils.actions.ICividasCron

que proporciona algún método extra para completar la funcionalidad dada por Ontimize. Vamos a continuación, pues, definir un pequeño problema que resolveremos haciendo uso de estas herramientas.

Problema 1

Queremos crear un proceso, que se levante cada 20 segundos y que actualice las observaciones de TODAS las entidades administrativas con el resultado de la llamada a la referencia remota creada anteriormente.

Solución 1

[…]

Ejecución de acciones

Siguiendo con el ejemplo actual, ahora nos centraremos en el combo que nos permite definir la acción a realizar en caso de error durante la ejecución del proceso. Este tipo de acciones también son configurables desde la aplicación pero su implementación difiere ligeramente sobre la de ejecución de procesos. Cabe señalar, en todo caso, que estas acciones estarán disponibles tanto en la definición de los procesos a ejecutar como en la definición de controles y plazos sobre fechas de los trámites, ya que en ambos casos la interfaz a implementar es la misma y el combo que alimenta a la selección también es el mismo.

En definitiva, se trata de clases que deben implementar la siguiente interfaz:

com.cividas.server.utils.actions.ICividasActions

Esta interfaz impone sobre todo dos métodos:

```
public EntityResult executeAction(Hashtable parameters, EntityReferenceLocator refLocator)
                    throws Exception;
public EntityResult executeAction(Hashtable parameters, EntityReferenceLocator refLocator,
                    int sessionID,Connection con) throws Exception ;
```

Realmente ambos métodos son versiones de la misma utilidad, una teniendo en cuenta que disponemos de un ID de sesión y un conexión contra la BBDD y otra teniendo en cuenta que la acción se puede lanzar desde cualquier punto sin tener los valores anteriores. La recomendación es estructurar nuestra acción de una forma similar a lo comentado en la Referencia Remota: El grueso de la lógica irá incluído en el método sin conexión siendo la implementación del otro una mera capa que provee de ID de sesión y establece una conexión contra la BBDD para acabar llamando con esa información al primer método.

Problema 2

En nuestro caso vamos a definir una acción para que en caso de error deje constancia del mismo en las observaciones del cron.

Solución 2

[…]

Subprocesos VS Procesos secundarios

En este punto se debatirá la conveniencia o no de utilizar subprocesos del WF o bien los procesos secundarios.

Informes Jasper

En este apartado se tratará de añadir un botón que permita el rellenado de un informe Jasper configurable a través de la parametrización de Cividas.