



*Ministerio de Educación, Ciencia y Tecnología*  
*Secretaría de Políticas Universitarias*

**SIU**

# Manual SIU-Toba

Versión 1.2.0



<b>1. Introducción.....</b>	<b>4</b>
Flujo de trabajo.....	4
Esquema de Directorios.....	4
<b>2. Instalación de SIU-Toba.....</b>	<b>6</b>
Guía Rápida Windows.....	6
Guía Rápida GNU/Linux .....	6
<b>3. Editor Toba.....</b>	<b>8</b>
Primeros pasos.....	8
Previsualización.....	8
Árbol de Operaciones.....	9
<b>4. Asistentes: Construcción automática de ABMs.....</b>	<b>11</b>
Paso 1: Seleccionar Tipo de Operación.....	11
Paso 2: Definir la tabla a editar.....	11
Paso 3: Generar operación.....	12
<b>5. Componentes.....</b>	<b>13</b>
Tipos de Componentes.....	13
Extensión de Componentes.....	14
<b>6. Controlador de Interface.....</b>	<b>16</b>
Atención de eventos.....	16
Configuración.....	17
Propiedades en sesión.....	18
<b>7. Persistencia.....</b>	<b>20</b>
Transacción inmediata.....	20
Transacción a nivel operación.....	21
Carga de la relación.....	22
API Tabular de las tablas.....	23
Sincronización.....	25
<b>8. Formularios.....</b>	<b>27</b>
Tipos de Efs.....	28
Carga de opciones.....	29
<b>9. Cuadro.....</b>	<b>30</b>
Configuración y Eventos.....	30
Paginado.....	31
Ordenamiento.....	32
<b>10. Construcción manual de ABMs.....</b>	<b>33</b>
Programación de un ABM Simple.....	33
Programación de un ABM Multitabla.....	34
Extensión del Ci de Navegación/Selección.....	35
Extensión del Ci de Edición.....	37
<b>11. Extensiones Javascript.....</b>	<b>42</b>
Atrapando Eventos.....	42
Redefiniendo métodos.....	43
<b>12. Otras partes del API.....</b>	<b>44</b>
Fuentes de Datos.....	44
Vinculación entre Items.....	44
Memoria.....	45
Logger.....	46
Mensajes y Notificaciones.....	46
<b>13. Esquema de Ejecución.....</b>	<b>48</b>
<b>14. Administración Básica.....</b>	<b>49</b>
Metadatos.....	49
Pasaje Desarrollo - Producción.....	50
Actualización de Toba.....	51

Este material está basado en la documentación online del proyecto. Se puede encontrar más información sobre temas particulares en:

- Wiki: <http://desarrollos2.siu.edu.ar/trac/toba/wiki>

- Tutorial: [http://desarrollos2.siu.edu.ar/toba\\_referencia\\_trunk](http://desarrollos2.siu.edu.ar/toba_referencia_trunk)

# 1. Introducción

**SIU-Toba** es un ambiente de desarrollo Web creado por el SIU con el objetivo de brindar una herramienta de desarrollo rápido para construir aplicaciones transaccionales. Lo llamamos **Ambiente** porque es una suite de distintas utilidades:

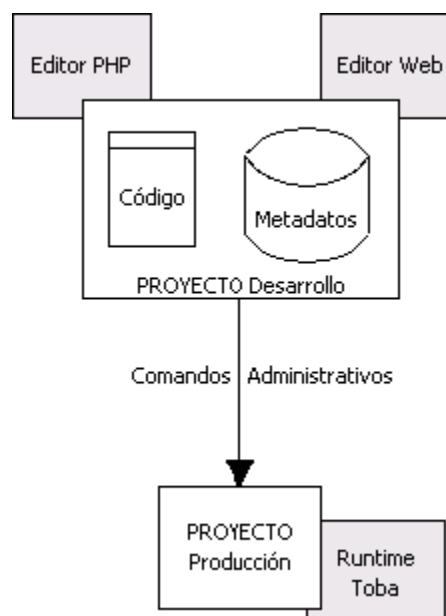
- Un conjunto de **librerías** que son consumidas en ejecución.
- Un **editor web** destinado a la definición/configuración del proyecto, creación de operaciones y definición de sus comportamientos.
- Un conjunto de **comandos de consola** destinados a administrar los proyectos creados con la herramienta.

Las definiciones realizadas en el editor web se las denomina **metadatos**, y junto a las definiciones en código conforman el comportamiento del proyecto creado. Durante el desarrollo estos metadatos son almacenados en una base de datos relacional denominada **instancia**.

## Flujo de trabajo

El flujo de desarrollo con la herramienta podría definirse así:

1. Se utiliza el **editor web** de toba para definir una operación, sus pantallas, sus componentes gráficos, tablas que se consumen, etc. Todo esto se almacena en metadatos en una base de datos.
2. Se utiliza un **editor PHP** para crear el código necesario para cubrir lógica particular de la operación.
3. Durante este proceso se va probando la operación desde el mismo **editor web** haciendo ajustes contextuales.
4. Una vez terminada se utilizan los **comandos administrativos** para exportar el proyecto desde el puesto de desarrollo e importarlo en el sistema en producción.
5. En el sistema en producción sólo necesita las **librerías** o *runtime* para ejecutar el proyecto (código + metadatos).



## Esquema de Directorios

La siguiente es una lista de los directorios más importantes de Toba y sus funcionalidades a alto nivel:

- **bin**: Contiene la puerta de entrada a los [comandos de consola](#)<sup>Wiki</sup>. Para poder ejecutarlos desde cualquier terminal/consola, a este directorio es necesario incluirlo en el PATH del sistema operativo.
- **doc**: Contiene documentación interna del proyecto. Para el desarrollador, la mejor documentación se encuentra en el [wiki](#)<sup>Wiki</sup> y en este tutorial.
- **instalacion**: Contiene toda la configuración local (bases de datos que se utilizan, proyectos que se editan, alias de apache, etc.) y los metadatos locales (logs, usuarios, etc.). Generalmente es un directorio que no se versiona ya que solo contiene información local de esta instalación.

- **php**
  - **3ros**: Librerías externas utilizadas en el proyecto.
  - **consola**: Código fuente de los comandos administrativos de consola.
  - **contrib**: Código contribuido por los proyectos, que aún no pertenecen al núcleo pero que esta bueno compartir.
  - **lib**: Clases sueltas propias comunes a todo el ambiente
  - **modelo**: Contiene una serie de clases que utilizan el editor y los comandos para editar metadatos y código. Forman una base útil para armar otras herramientas consumiendo una API de alto nivel. Por ejemplo si el proyecto determina que es necesario desarrollar un instalador con prestaciones extras, es un buen comienzo consumir estas clases.
  - **nucleo**: *Runtime* o conjunto de clases que se utilizan en la ejecución de un proyecto. La documentación de estas clases [se encuentra publicada](#).
- **proyectos**: Este directorio contiene los [proyectos propios](#)<sup>Wiki</sup> del ambiente y es el lugar sugerido para nuevos proyectos. Aunque pueden situarlos en cualquier directorio, si están aquí es más fácil configurarlos.
  - ...
  - **mi\_proyecto**:
    - **metadatos**: Contiene la última exportación de metadatos del proyecto.
    - **php**: Directorio que será parte del *include\_path* de PHP, se asume que el proyecto almacenará aquí sus extensiones y demás código.
    - **temp**: Directorio temporal no-navegable propio del proyecto
    - **www**: Directorio navegable que contiene los [puntos de acceso](#)<sup>Wiki</sup> a la aplicación.
      - **css**: Plantillas de estilos CSS del proyecto.
      - **img**: Imágenes propias del proyecto.
      - **temp**: Directorio temporal navegable del proyecto.
  - ...
- **temp**: Directorio temporal no-navegable común.
- **var**: Recursos internos a Toba.
- **www**: Directorio navegable que contiene recursos web que consumen el *runtime* y los proyectos.
  - **css**: Plantillas de estilos CSS disponibles.
  - **img**: Imágenes comunes que pueden utilizar los proyectos.
  - **js**: Clases javascript propias de toba y externas.
  - **temp**: Directorio temporal navegable común.

## 2. Instalación de SIU-Toba

Para más detalles de la instalación consultar el [wiki de toba](#)

### Guía Rápida Windows

1. Descarga instalador [Apache 2.2.9](#) y ejecutarlo con las opciones predeterminadas.
2. Bajar el instalador de [PHP versión 5.2.5](#) y ejecutarlo
  - o Seleccionar el módulo apache 2.2
  - o Indicar el directorio de configuración de Apache (generalmente C:\Archivos de Programa\Apache Software Foundation\Apache2.2\conf
  - o Seleccionar la extensión PDO PostgreSQL.
  - o En caso de que a futuro necesite instalar alguna otra extensión lo puede hacer desde Agregar/Quitar Programas del Panel de Control de Windows. No borrar el instalador .msi para poder reejecutarlo.
3. Reiniciar el servidor apache para terminar la instalación de php. Navegar hacia <http://localhost> verificando que apache funciona.
4. Instalar la última versión estable de [PostgreSQL 8.2.9](#), seleccionar opciones por defecto
5. Instalar cliente [Subversion 1.4.6](#), seleccionar opciones por defecto.
6. [Descomprimir o descargar](#) el código fuente de SIU-Toba por ejemplo en el path c:\proyectos\toba
7. En c:\proyectos\toba\bin, ejecutar instalar.bat y seguir las instrucciones.
8. Para editar el proyecto navegar hacia [http://localhost/toba\\_editor/1.2.0](http://localhost/toba_editor/1.2.0)
9. Para ejecutar el proyecto navegar hacia <http://localhost/miproyecto/1.0.0> (cambiar la url según el identificador que ha dado al proyecto).

### Guía Rápida GNU/Linux

Para distribuciones debian o derivados:

1. Ejecutar como superusuario:  

```
apt-get install apache2 libapache2-mod-php5 php5-cli php5-pgsql php5-gd
```
2. En caso de no encontrar los paquetes pdo o pdo\_pgsql instalarlos manualmente:  

```
apt-get install php5-dev php-pear postgresql-server-dev-8.1 build-essential  
pecl install pdo  
pecl install pdo_pgsql
```
3. Para instalar servidor y cliente de Postgresql ejecutar  

```
apt-get install postgresql-8.2 postgresql-client-8.2
```
4. Para instalar el cliente Subversion ejecutar  

```
apt-get install subversion
```
5. Para instalar graphviz ejecutar  

```
apt-get install graphviz
```
6. Editar el archivo /etc/php5/apache2/php.ini y /etc/php5/cli/php.ini:

```

#Mínimos
magic_quotes_gpc = Off
magic_quotes_runtime = Off
extension=php_pdo.so           #En caso de que se haya instalado
manualmente con PECL
extension=php_pdo_pgsql.so     #En caso de que se haya instalado
manualmente con PECL

#Recomendados
error_reporting = E_ALL        #Solo para desarrollo
memory_limit = 64M
post_max_size = 8 M
upload_max_filesize = 8 M

```

7. [Descomprimir o bajar](#) el código fuente de SIU-Toba por ejemplo en el path  
~/proyectos/toba
8. Ejecutar y seguir las instrucciones:

```
~/proyectos/toba/bin/instalar
```

9. Tener en cuenta el esquema de permisos de UNIX. Apache necesita acceso a las carpeta `www` de Toba y del Proyecto. Si se quieren guardar los logs (opción por defecto) también necesita acceso de escritura a la carpeta `instalacion`. Para facilitar esto en las nuevas versiones existe el siguiente comando que acomoda los permisos luego de la instalacion:

```
toba instalacion cambiar_permisos -u www-data -g admin
```

10. Para editar el proyecto navegar hacia [http://localhost/toba\\_editor/1.2.0](http://localhost/toba_editor/1.2.0)
11. Para ejecutar el proyecto navegar hacia <http://localhost/miproyecto/1.0.0> (cambiar la url según el identificador que ha dado al proyecto).
12. Finalmente ver notas sobre la [codificación local \(encoding\)](#)

### 3. Editor Toba

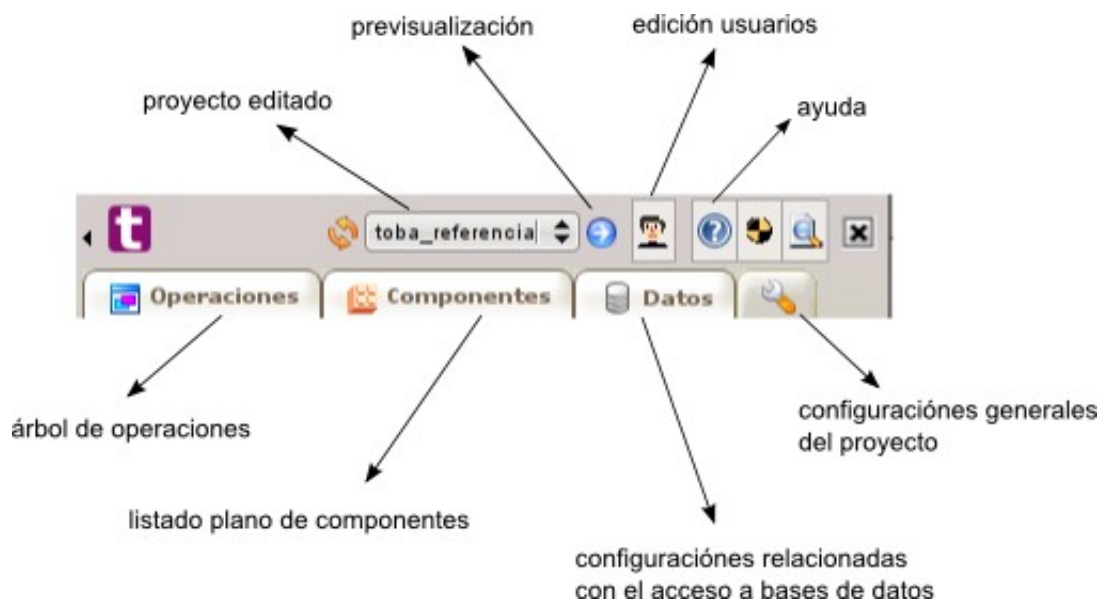
El editor Toba cubre distintos aspectos de la implementación de una aplicación:

- Configuración global del proyecto.
- Creación y edición de operaciones.
- Definición de las distintas fuentes de datos (base de datos) que se consumen.
- Previsualización de la aplicación. Con utilidades como edición contextual, visualización de logs o cronometraje de ejecución.

#### Primeros pasos

El editor es accesible navegando hacia [http://localhost/toba\\_editor/1.2.0](http://localhost/toba_editor/1.2.0)

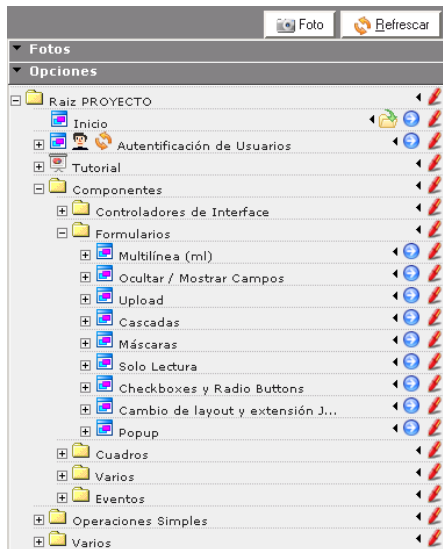
Allí se selecciona el proyecto a editar (toba\_referencia en este caso) y se encuentra con el menú principal del editor:



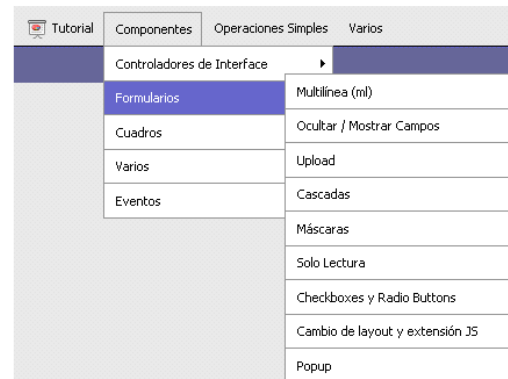
#### Previsualización

La previsualización permite ejecutar la aplicación desde el entorno de edición, a este proceso lo llamamos *instanciar* el proyecto. Al instanciar vemos la primera relación con la vista izquierda del editor. El menú de la aplicación lo vemos en forma vertical con una serie de utilidades contextuales.





Vista desde el editor



Menú de la aplicación en ejecución

Cuando se instancia el proyecto se cuenta con una barra de utilidades situada en la izq. Inferior derecha.



La Edición Contextual resalta en la pantalla aquellas partes que pueden ser editadas (el mismo efecto se logra presionando la tecla Control). Al clickear en alguno de estos enlaces podemos cambiar sus propiedades y al refrescar la operación.

## Árbol de Operaciones

Si se piensa la aplicación como un *Catálogo de operaciones*, cada una de estas operaciones se la puede pensar como un **ítem** de este catálogo. Así en la vista de ítems se puede ver tanto la imagen global del proyecto como la particular de una operación específica.

Al hacer clic sobre un ítem podemos ver que está formado de distintos **componentes**, desde el editor es posible editarlos, ver su contenido de código PHP o crear un nuevo componente hijo.



## 4. Asistentes: Construcción automática de ABMs

Por lo general la forma de construir una operación es ir definiendo los distintos componentes que la componen y programar la interacción entre los mismos. Cuando el tipo de operación es altamente repetitivo como el caso de un ABM (altas, bajas y modificaciones, también llamado CRUD) se puede utilizar un atajo llamado **Asistentes**. El asistente va a permitir definir la operación a alto nivel, construyendo tanto los componentes como su código. Veamos paso a paso un ejemplo de su uso

### Paso 1: Seleccionar Tipo de Operación

Dentro del menú *Operaciones* presionamos el botón *Crear* invocando así al asistente.



Una vez lanzado el asistente nos pregunta el tipo de operación a crear (en este caso ABM Simple) y la carpeta del menú donde será publicada.

Esta imagen muestra la interfaz de un asistente de configuración. Tiene tres campos de entrada: 'Carpeta (\*)' con un menú desplegable que muestra 'Raiz PROYECTO', 'Nombre Operación (\*)' con un campo de texto que contiene 'Juegos', y 'Tipo de Operación (\*)' con tres opciones de radio: 'ABM simple' (seleccionada), 'Grilla simple' y 'Importar Operación'. Debajo de estos campos hay un botón 'Ver demo' y un botón 'Siguiente' con una flecha azul. En la parte superior, hay un título 'Alta, baja y modificacion de registros de una tabla'.

### Paso 2: Definir la tabla a editar

Al presionar *Siguiente* se nos pregunta la carpeta del sistema de archivos donde será almacenado el código fuente creado automáticamente. Lo más importante de esta pantalla es la elección de la tabla a la cual se le va a construir un ABM, en este caso la tabla es *ref\_juegos*.

**Información básica** **Cuadro** **Formulario**

**Carpeta archivos (\*)** juegos

**Anexo nombre clases (\*)** juegos

**Fuente de Datos (\*)** Datos de prueba

**Tabla (\*)** ref\_juegos

Puede filtrar los datos ☐

**Definición de campos**

Campo	Etiqueta	Clave	Obligatorio	Tipo	Cuadro	Form.
id	Id	<input checked="" type="checkbox"/>	<input type="checkbox"/>	Entero	<input type="checkbox"/>	<input type="checkbox"/>
nombre	Nombre	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Cadena	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
descripcion	Descripción	<input type="checkbox"/>	<input type="checkbox"/>	Cadena	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Recargar

Anterior Generar

Una vez

### Paso 3: Generar operación

Una vez presionado el botón *Generar* se crea la operación

**Información**

La generación se realizó exitosamente

Aceptar

Si observamos el árbol de operaciones se encuentra el detalle de componentes que forman parte de esta nueva operación. Si la instanciamos ya tenemos el ABM de Juegos funcionando.

**Raiz PROYECTO** >

- Juegos
  - Juegos - CI
    - Pantalla
      - cuadro
        - formulario
          - datos

Nombre	Descripción	
Ajedrez		
Damas		
Go		
Reversi		

**Nombre (\*)** Reversi

Descripción

Modificar Eliminar Cancelar

## 5. Componentes

Una operación se construye a partir *componentes*. Estos elementos se definen en forma declarativa dentro del ambiente. El comportamiento particular de los componentes es determinado por:

- La definición usando el **editor web**.
- La extensión PHP del componente, utilizando algún **editor php**.
- La extensión Javascript del componente, usando algún editor de texto.

Los componentes son unidades o elementos que cubren distintos aspectos de una operación. Construir una operación en base a componentes permite que:

- Un componente pueda reutilizarse en distintas operaciones o en distintas partes de la misma operación.
- Los componentes se puedan componer o encastrar entre sí.
- Cada uno encapsule algún comportamiento complejo, que ya no es necesario programar.
- Al estar categorizados según su función, se logre en el sistema una separación en capas de forma transparente.



### Tipos de Componentes


Los componentes se categorizan según su función: interface, control o persistencia.

#### Interface

Los componentes o elementos de interface son controles gráficos o widgets. En su aspecto gráfico se basan en los elementos HTML aunque su comportamiento va más allá, tomando responsabilidades tanto en el cliente como en el servidor:

- En el servidor recibe un conjunto de datos a partir del control.
- Se grafica utilizando HTML.
- Tiene un comportamiento en el browser (usando javascript).
- Se comunica con su par en el servidor a través del POST.
- Se analizan los nuevos datos o acciones y se notifican al control.


 [ei\\_formulario](#): Formulario de edición simple y sus  [elementos de formulario \(efs\)](#).


 [ei\\_formulario\\_ml](#): Formulario de edición de múltiples líneas.


 [ei\\_filtro](#): Filtro de selección.

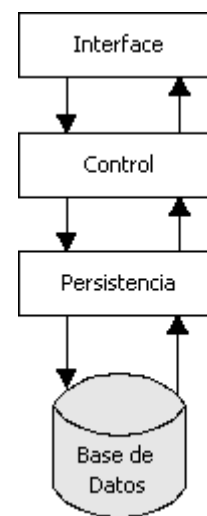
 [ei\\_cuadro](#): Grilla para mostrar y seleccionar datos.

 [ei\\_calendario](#): Calendario para visualizar contenidos diarios y seleccionar días o semanas.

 [ei\\_arbol](#): Visualiza una estructura de árbol en HTML.


 [ei\\_archivos](#): Permite navegar el sistema de archivos del servidor bajo una carpeta dada.


 [ei\\_esquema](#): Muestra un esquema utilizando [GraphViz](#).



#### Control

El componente llamado **controlador de interface** oficia como intermediario entre las capas de datos o negocio y la interface, formando la *plasticola* necesaria para construir una operación con componentes.


 **Controlador de Interface** (ci): Contienen otros componentes [configurándolos](#), escuchando sus [eventos](#), agrupándolos en pantallas y determinando la lógica de navegación entre las mismas. Son el nexo entre las capas de *persistencia* o *negocio* y la de *interface*.

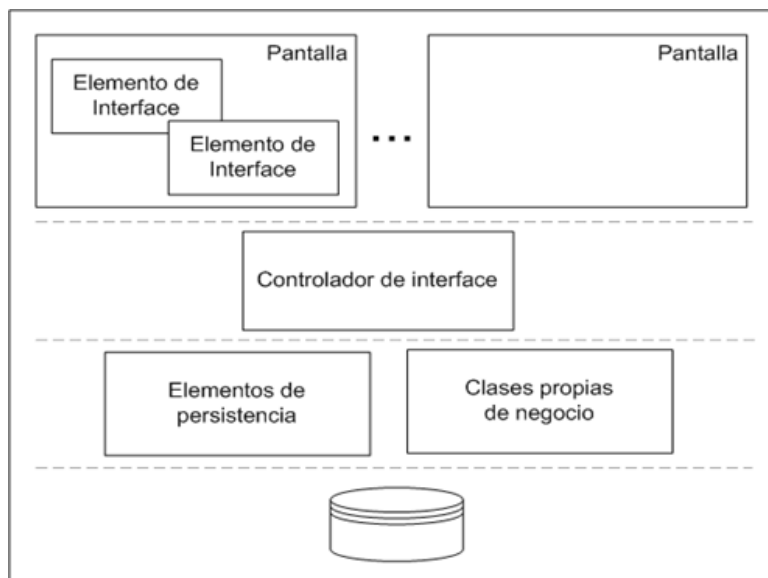
 **Controlador de Negocio** (cn): desacoplan la lógica de la interface de las clases de negocio propias o de los componentes de datos.

## Persistencia

Los componentes de persistencia permiten modelar tablas y registros de una base relacional o algún otro medio que maneje datos tabulares, brindando servicios transaccionales a las capas superiores. En estos componentes se describen las estructuras y asociaciones de las tablas involucradas en una operación y la forma en que los registros serán sincronizados al final de la misma.

 [datos\\_tabla](#): Abstracción de una tabla.


 [datos\\_relacion](#): Abstracción de una relación entre tablas.



## Extensión de Componentes

El comportamiento de un componente se basa en su definición y, en distinta medida según el tipo, su extensión en código.

La extensión en código se da a través de la herencia, creando una subclase del componente en cuestión y seleccionándola durante la definición del componente en el editor. Se podrían definir tres objetivos distintos a la hora de hacer una extensión de un componente:

- **Atender eventos:** El componente notifica *sucesos* y en la extensión se escuchan. A esta comunicación se la denomina *eventos* y se la ve más adelante en el tutorial.
- **Redefinir métodos:** En la [documentación API](#) los métodos recomendados para extender llevan a su lado un ícono de ventana . Otros métodos protegidos son extensibles también, pero si no poseen la ventana no se asegura que en futuras versiones del framework serán soportados, ya que lo que se está extendiendo es un método interno.

- **Extender el componente en Javascript:** Cada componente en PHP tiene su par en Javascript, por lo que en la extensión también es posible variar el comportamiento del componente en el cliente.

Para tener una idea de como 'lucen' una extensión, se presenta una extensión típica de un componente controlador. La idea no es entender en profundidad esta extensión sino es para tomar un primer contacto. Como se ve en los comentarios del código, en este caso se consumieron las formas de extensión vistas:


```
<?php
class ci_pago extends toba_ci
{
    /**
     * Atención de un evento "pagar" de un formulario
     */
    function evt__form_pago__pagar($datos)
    {
        $this->s_pago = $datos;
        $this->set_pantalla("pant_ubicacion");
    }


    /**
     * Redefinición de un método
     */
    function ini()
    {
        $this->valor_defecto = 0;
    }

    /**
     * Redefinición de un método para extender el componente en JS
     */
    function extender_objeto_js()
    {
        echo "
            /**
             * Atención del evento procesar de este componente
             */
            {$this->objeto_js}.evt__procesar = function() {
                return prompt(\"Desea Procesar?\");
            }
        ";
    }
}
```

?>

## 6. Controlador de Interface

El Controlador de interface o  CI es el componente raíz que necesitamos definir en nuestra operación, ya que tiene la capacidad de contener otros componentes, formando las distintas ramas del árbol de una operación.

Para organizar la operación, el CI tiene la capacidad de definir  **Pantallas** siendo responsable de la lógica de navegación entre las mismas y de los componentes que utiliza cada una. La forma más usual de navegación entre estas pantallas es usar solapas o tabs horizontales.



*Ejecución de la operación, las pantallas se ven como solapas horizontales*

### Atención de eventos

Un **Evento** representa la interacción del usuario. Al ser aplicaciones web, esta interacción surge en el cliente (navegador o browser) donde el usuario ha realizado acciones que deben ser atendidas en el lado servidor. En el servidor el lugar para atender esas acciones es la extensión del CI.

Una vez que definimos los componentes es hora de extender el CI y definir una subclase vacía. Lo primero que vamos a agregar a esta subclase es la atención de eventos del cuadro y del formulario. La forma de *atrapar* un evento es definir un método

```
function evt__causante__evento($parametros)
```

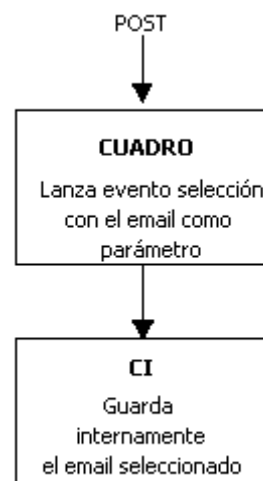
Donde *causante* es el id que toma el componente en el CI, y *evento* es el id del evento tal como se definio en el editor.

#### Evento Selección del cuadro

En el primer caso que vamos a tomar es el del **cuadro**. Cuando el usuario selecciona un elemento de la grilla, ese elemento debe ser guardado internamente para luego mostrar sus datos asociados en el formulario.

```
<?php
class ci_abm_direcciones extends toba_ci
{
    protected $actual;

    function evt__cuadro__seleccion($direccion)
    {
```





```

        $this->actual = $direccion["email"];
    }

    }
?>

```

### Evento Alta del formulario

El segundo caso de evento lo vamos a tomar del formulario, cuando presionamos el botón *Agregar*, viaja por el POST una nueva dirección de email que el formulario entrega con el evento *Alta*

```

<?php
//---Dentro de la subclase del CI

protected $direcciones;

/**
 * En el alta agrega la direccion al arreglo, indexado por email
 */
function evt__form__alta($nueva_dir)
{
    $email = $nueva_dir["email"];
    $this->direcciones[$email] = $nueva_dir;
}
?>

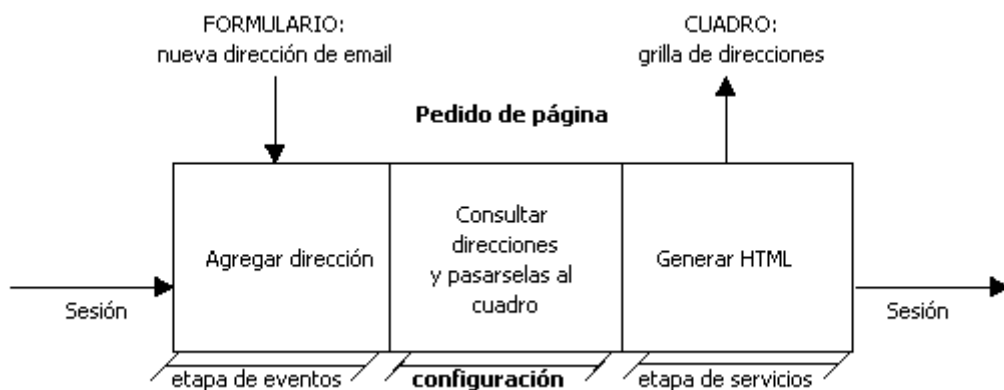
```



## Configuración

Una vez atendidas las acciones del usuario, la operación se dispone a construir una nueva interface a partir de sus componentes. Para ello primero se deben **configurar** los distintos componentes que formarán parte de la salida HTML. Para configurar un componente se debe definir un método *conf\_\_dependencia* donde *dependencia* es el id del componente en el CI.

En el siguiente gráfico podemos ver donde estamos parados en el pedido de página actual



### Configuración del Cuadro

Ya vimos como el formulario agregaba las direcciones en un arreglo, este arreglo es el que necesitará el cuadro para mostrar la grilla:

```
<?php
//---Dentro de la subclase del CI

function conf__cuadro(toba_ei_cuadro $cuadro)
{
    $cuadro->set_datos($this->direcciones);
}

?>
```

### Configuración del Formulario

Durante la configuración también vamos a cargar al formulario con datos, pero sólo cuando previamente se ha seleccionado algo desde el cuadro (así se edita lo que se seleccionó). En caso contrario no se cargarán datos y el formulario se graficará vacío.

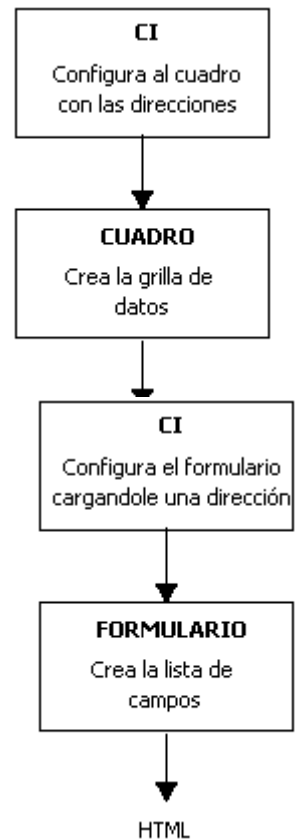
```
<?php
//---Dentro de la subclase del CI

function conf__form(toba_ei_formulario $formulario)
{
    if (isset($this->actual)) {
        $formulario->set_datos($this->direcciones[$this->actual]);
    }
}

?>
```

### Otras configuraciones

Además de componentes, el ci se puede configurar a sí mismo (definiendo el método *conf*) y a sus pantallas (*conf\_idpant*)



## Propiedades en sesión

Para cerrar el circuito eventos-configuración es necesario que el ci pueda **recordar** la información que va recolectando entre pedidos de página. Esto se logra gracias a las llamadas **variables de sesión**.

La forma de indicar al framework que una propiedad sea mantenida en sesión es prefijar su nombre con *s\_\_* (de sesión), en nuestro ejemplo mantendremos las direcciones y la selección actual en sesión:

```
<?php
```

```
class ci_abm_direcciones extends toba_ci
{
    protected $s__direcciones;
    protected $s__actual;
    ....
}
?>
```

## 7. Persistencia

En el capítulo anterior (CI) se dejó de lado un aspecto muy importante de una operación: la transacción con una base de datos. Las operaciones de Alta, Baja y Modificación (ABM) fueron impactando en un arreglo en memoria que se mantiene en sesión, pero en cuanto el usuario cierra la aplicación o navega hacia otra operación los cambios se pierden.

Para armar una operación 'real' se debe transaccionar con una fuente de datos, generalmente esta fuente es una base de datos relacional así que vamos a concentrarnos en ellas. Existen dos formas principales en que puede *transaccionar* una operación:

- inmediatamente producidos los eventos, prácticamente en cada pedido de página, o
- luego de una confirmación explícita del usuario.

### Transacción inmediata

Si los requisitos funcionales lo permiten, transaccionar inmediatamente cuando se produce el evento es la forma más fácil y directa de programar una operación. Simplemente en cada método que escucha un evento se ejecuta un comando SQL, y en las configuraciones se cargan los componentes directamente usando consultas SQL:

```
<?php
...

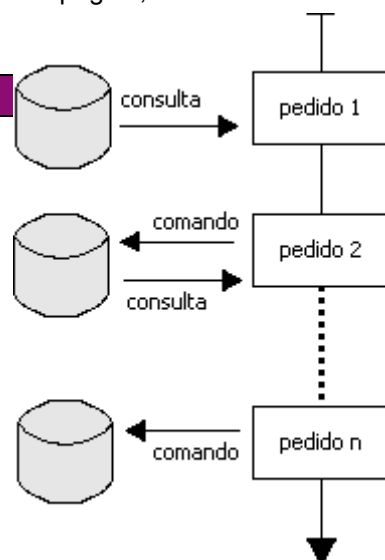
function evt__form__alta($datos)
{
    $sql = "INSERT INTO direcciones (email, nombre)
VALUES
    (\"$datos['email']\", \"$datos['nombre']\");
    toba::db()->ejecutar($sql);
}

function evt__form__modificacion($datos)
{
    $sql = "UPDATE direcciones SET nombre = \"$datos['nombre']\"
    WHERE email=\"$datos['email']\"";
    toba::db()->ejecutar($sql);
}

function evt__form__baja()
{
    $sql = "DELETE FROM direcciones WHERE email=\"$datos['email']\"";
    toba::db()->ejecutar($sql);
}

function conf__form(toba_ei_formulario $form)
{
    $sql = "SELECT email, nombre FROM direcciones WHERE email='{ $this->s__actual}'";
    $datos = toba::db()->consultar_fila($sql);
    $form->set_datos($datos);
}

function conf__cuadro(toba_ei_cuadro $cuadro)
```



```

{
    $sql = "SELECT email, nombre FROM direcciones";
    $datos = toba::db()->consultar($sql);
    $cuadro->set_datos($datos);
}
...
?>

```

## Transacción a nivel operación

El único problema con transaccionar continuamente es que muchas veces es un requisito funcional que la edición se maneje como una única transacción, que se cierra cuando el usuario decide presionar *Guardar*.

Este requisito se fundamenta en que si el usuario decide dejar la edición por la mitad, lo editado queda en un estado **inconsistente**. Esto depende en gran medida de lo que se está editando, por ejemplo si edito la información de una beca no tiene sentido que el usuario sólo llene la primer solapa si se cuenta con digamos 8 solapas más para llenar. En una aplicación web no es posible 'detectar' cuando el usuario cierra el navegador, cierra la ventana o navega hacia otra página, no es posible (o es muy difícil mejor dicho) *Deshacer* las inconsistencias.

La solución que se da a nivel Toba es confiar el manejo de la transacción en unos **componentes de persistencia**. Estos serán los encargados de:

1. Hacer las consultas para obtener los datos al inicio de la operación/transacción
2. Brindar una api para manejar los datos en sesión durante la operación
3. Analizar los cambios y sincronizarlo con la base de datos al fin de la operación

Estos tres pasos se pueden ver en la siguiente extensión de un CI:

La operación forma una única transacción a nivel lógico

```

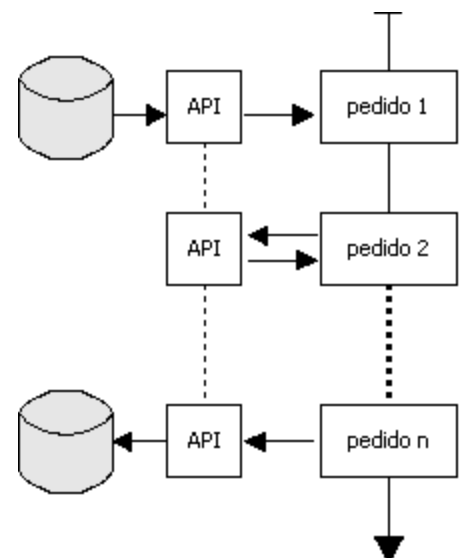
<?php
...

/**
 * Momento 1:
 * Carga inicial de datos, solo se da en el primer pedido de página
 */
function ini_operacion()
{
    $this->dep("direcciones")->cargar();
}

/**
 * Momento 2:
 * Eventos y configuración dialogan con la api
 */

function evt_form_alta($datos)
{
    $this->dep("direcciones")->nueva_fila($datos);
}

```



```

    }

    function evt__form__baja()
    {
        $this->dep("direcciones")->eliminar_filas($this->s__actual);
    }

    function evt__form__modificacion($datos)
    {
        $this->dep("direcciones")->modificar_filas($this->s__actual, $datos);
    }

    function conf__form(toba_ei_formulario $form)
    {
        $datos = $this->dep("direcciones")->get_filas($this->s__actual);
        $form->set_datos($datos);
    }

    function conf__cuadro(toba_ei_cuadro $cuadro)
    {
        $datos = $this->dep("direcciones")->get_filas();
        $cuadro->set_datos($datos);
    }

    /**
     * Momento 3:
     * Sincronización con la base de datos, solo se da cuando el usuario p
     * resiona GUARDAR
     */
    function evt__procesar()
    {
        $this->dep("direcciones")->sincronizar();
    }

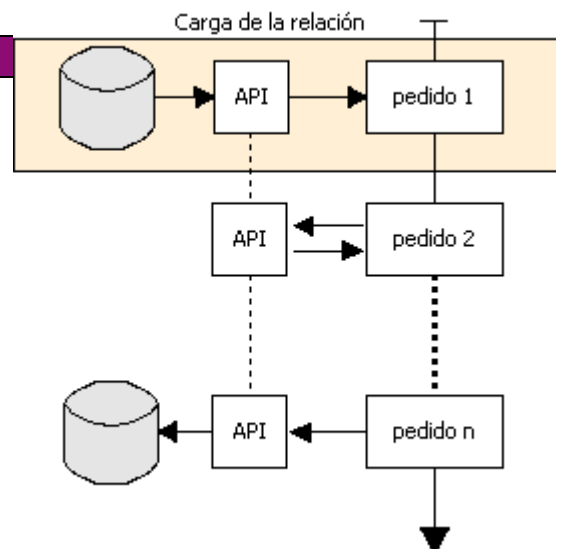
    ...
    ?>

```

## Carga de la relación


La transacción a nivel operación se inicia cuando el componente *datos\_relacion* se carga con datos. A partir de allí todo el trabajo de Altas, Bajas y Modificaciones debe impactar sobre los componentes *datos\_tabla* de esa relación y no se sincronizará con la base de datos hasta el final de la transacción.

La relación se carga como un todo, generalmente dando algún valor clave de las tablas raíces. En una relación una tabla raíz es aquella que no tiene padre, en este caso *ref\_persona*. La carga va formando las SQL de las tablas hijas en base a subselects de la tabla padre. También es posible pasarle otros criterios a la carga e incluso armar la consulta manualmente y pasarle directamente los datos a la relación, siendo estos casos menos comunes y no serán vistos en este capítulo.



Para el caso particular del ejemplo, la relación se carga cuando se selecciona una persona del cuadro, pasándole la clave de la selección:

```
<?php
function evt__cuadro_personas__seleccion($id)
{
    $this->dep("relacion")->cargar($id);
    $this->set_pantalla("edicion");
}
?>
```

Si utilizamos el  **visor del logger** durante la carga podemos ver las consultas que internamente utilizan los componentes para cargar la relación

## API Tabular de las tablas

Una vez cargada la relación es posible consumir el API para manipular los registros en memoria. Las operaciones clásicas sobre la tabla son:

- `get_fila($clave)`
- `nueva_fila($registro)`
- `modificar_fila($clave, $registro)`
- `eliminar_fila($clave)`

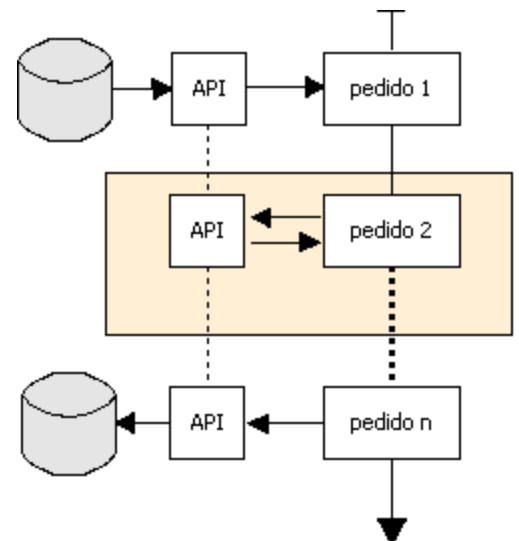
Cuando se conoce de antemano que una tabla en la transacción puede tener un único registro se dispone de dos métodos:

- `set($registro)`
- `get()`

Muchas de estas primitivas hacen referencia a una **\$clave**, vale notar que este valor no es la clave real del registro en la base sino un valor interno que maneja el `datos_tabla`. Por ejemplo al crear una `nueva_fila` se le brinda una clave para futuras referencias siendo que aún en la base aún no existe esta fila. Existe más información sobre las primitivas en la [documentación del `datos\_tabla`](#).

Volviendo al ejemplo, podemos ver el código que trabaja sobre los datos básicos y los deportes de una persona, tomado directamente de la [operación](#):

```
<?php
...
//-----
//--- Pantalla "persona"
// Se sabe de antemano que los datos de una persona es un único registro por lo
// que se trabaja con la api get/set
//-----
```



```

function conf__form_persona()
{
    return $this->get_relacion()->tabla("persona")->get();
}

function evt__form_persona_modificacion($registro)
{
    $this->get_relacion()->tabla("persona")->set($registro);
}

//-----
//--- Pantalla "deportes"
//-----

//-- Cuadro --

/**
 * El cuadro de deportes contiene todos los registros de deportes disponibles
 */
function conf__cuadro_deportes()
{
    return $this->get_relacion()->tabla("deportes")->get_filas();
}

function evt__cuadro_deportes_seleccion($seleccion) {
    $this->s__deporte = $seleccion;
}

//-- Formulario --

/**
 * Se carga al formulario con el deporte actualmente seleccionado
 */
function conf__form_deportes()
{
    if(isset($this->s__deporte)) {
        return $this->get_relacion()->tabla("deportes")->get_fila($this->s__deporte);
    }
}

/**
 * Se modifica el registro y se limpia el formulario
 */
function evt__form_deportes_modificacion($registro)
{
    if(isset($this->s__deporte)){
        $this->get_relacion()->tabla("deportes")->modificar_fila($this->s__deporte, $registro);
        $this->evt__form_deportes_cancelar();
    }
}

/**
 * Se borra el registro y se limpia el formulario
 */
function evt__form_deportes_baja()
{
    if(isset($this->s__deporte)){
        $this->get_relacion()->tabla("deportes")->eliminar_fila($this->s__deporte);
    }
}

```



```

s->s__deporte );
        $this->evt__form_deportes__cancelar();
    }
}

/**
 * Se crea una nueva fila
 */
function evt__form_deportes__alta($registro)
{
    $this->get_relacion()->tabla("deportes")->nueva_fila($registro);
}

function evt__form_deportes__cancelar()
{
    unset($this->s__deporte);
}
...
?>

```

## Sincronización

Finalmente la transacción finaliza sincronizando con el medio de persistencia, en este caso la base de datos. Durante este proceso se analizan todos los cambios que se produjeron desde la carga y se arma un plan de sincronización formando los comandos SQL necesarios.

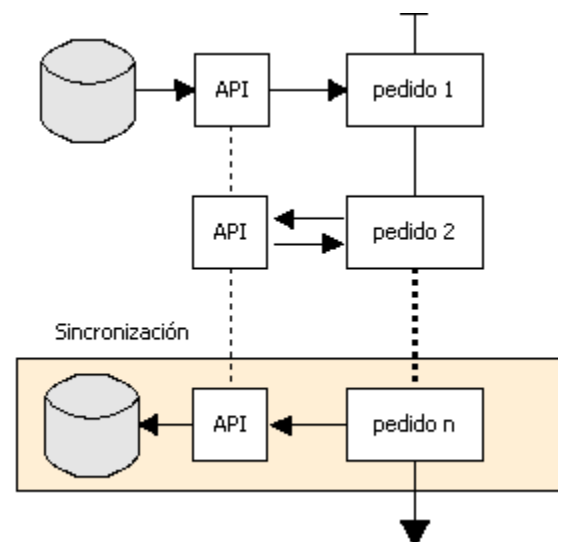
```

<?php
...
/**
 * Cuando presiona GUARDAR se sincroniza
 * a con la base, se resetea la relacion y se c
 *ambia de pantalla
 */
function evt__procesar()
{
    $this->get_relacion()->sincronizar();
    $this->get_relacion()->resetear();
    $this->set_pantalla("seleccion");
}


/**
 * Cuando presiona ELIMINAR se eliminan todos los registros de todas l
 *as tablas y
 * se sincroniza con la base (implicito), luego se cambia de pantalla
 */
function evt__eliminar()
{
    $this->get_relacion()->eliminar();
    $this->set_pantalla("seleccion");
}

/**
 * Cuando se presiona CANCELAR se descartan todos los cambio y se camb
 *ia de pantalla

```

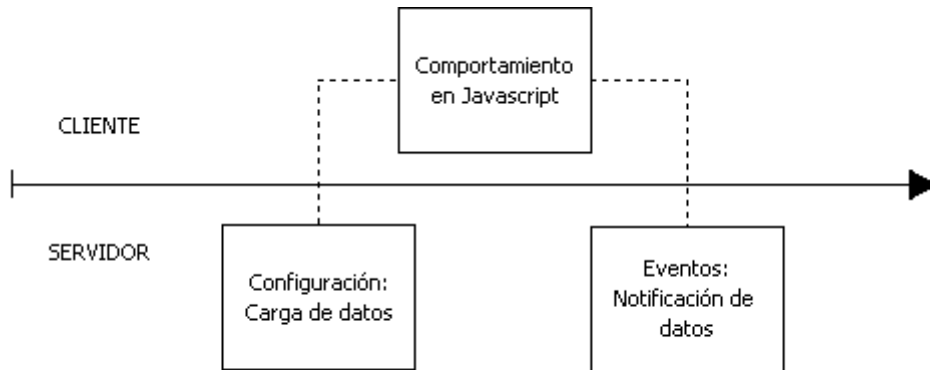


```
*/  
function evt__cancelar()  
{  
    $this->get_relacion()->resetear();  
    $this->set_pantalla("seleccion");  
}  
...  
?>
```

Si utilizamos el  **visor del logger** durante la sincronización podemos ver los comandos que generan los componentes para sincronizar la relación

## 8. Formularios

El formulario es un elemento de interface (ei) que permite incluir grillas de campos o elementos de formularios (efs). Durante la configuración se lo carga con un conjunto de datos y luego cuando vuelve al servidor informa a través de sus eventos el nuevo conjunto de datos editado por el usuario.



La forma de carga del formulario es un arreglo asociativo `id_ef=>estado`, se le dice *estado* al valor que toma el ef actualmente, independientemente de su formato. Por ejemplo para cargar el formulario de la imagen:

```
<?php
...
function conf__form(toba_ei_formulari
o $form)
{
    $datos = array(
        "fecha" => "2006-12-11",
        "editable" => "Texto",
        "moneda" => "234.23",
        "cuit" => "202806293",
        ....
    );
    $form->set_datos($datos);
}
...
?>
```

Fecha	11/12/2006
Editable	Texto
Moneda	\$ 234,23
CUIT/CUIL	20 - 28062902 - 3
Multilínea	
Número	343
Porcentaje	34 %

Los datos tienen el mismo formato cuando se disparan los eventos:

```
<?php
...
function evt__form__modificacion($datos)
{
    print_r($datos);
}
...
Array ( [fecha] => 2006-12-11 [editable] => Texto [moneda] => 234.23
[cuit] => 202806293 )
```

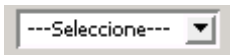
## Tipos de Efs

Los distintos tipos de elementos de formularios se pueden clasificar según la acción que el usuario realiza sobre ellos:

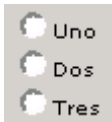
### El usuario selecciona un elemento



**ef\_checkbox:** Selección entre dos opciones, generalmente 0-1



**ef\_combo:** Selección entre varias opciones, pensado para conjuntos medianos de datos cuyos elementos son fáciles de encontrar por nombre.



**ef\_radio:** Selección entre varias opciones, pensado para conjuntos pequeños de datos, la elección es más explícita que en el combo, aunque ocupa mucho espacio como para poner muchas opciones. [Ver Ejemplo.](#)



**ef\_popup:** Al presionarlo, la elección entre las distintas opciones se realiza en una ventana aparte, en una operación separada. Pensado para conjuntos grandes con métodos de búsqueda complejos. La recomendación es usarlo sólo en casos justificados, ya que el combo o el radio brindan en general una mejor experiencia al usuario. [Ver Ejemplo.](#)

### El usuario selecciona varios elementos



**ef\_multi\_seleccion\_lista:** Selección usando el componente clásico HTML, difícil de entender para usuarios novatos ya que requiere presionar la tecla control o shift para hacer selecciones personalizadas.

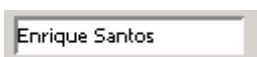


**ef\_multi\_seleccion\_doble:** Selecciona los elementos cruzandolo de un lado al otro.

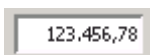


**ef\_multi\_seleccion\_check:** Selecciona los elementos tildando checkboxes. [Ver Ejemplo.](#)

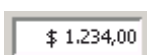
### El usuario edita



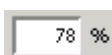
**ef\_editable:** El usuario edita texto libremente, respetando máximos.



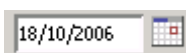
**ef\_editable\_numero:** El usuario edita un número, respetando límites mínimos y máximos.



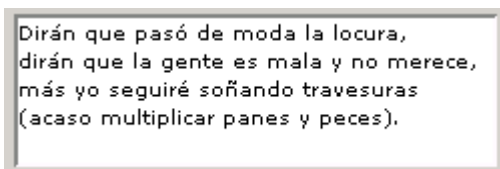
**ef\_editable\_moneda:** Igual al número, sólo que tiene una máscara que pone la moneda y tiene límites predefinidos.



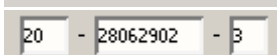
**ef\_editable\_numero\_porcentaje:** Número que representa un porcentaje.



**ef\_editable\_fecha:** El usuario ingresa una fecha, ayudado con un calendario.

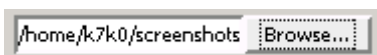


**ef\_editable\_textarea:** El usuario edita múltiples líneas de texto libremente, sin formato.



**ef\_cuit:** El usuario ingresa un número de CUIT/CUIL

### Otras acciones



**ef\_upload:** El usuario selecciona un archivo de su sistema para que esté disponible en el servidor. [Ver Ejemplo.](#)

## Carga de opciones

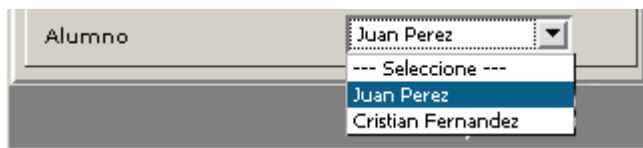
De los distintos tipos de efs disponibles existen los llamados de **selección**, estos permiten seleccionar su **estado** a partir de un conjunto de **opciones**.

La carga de los estados se vio anteriormente, se da durante la configuración del componente. La carga de opciones se puede realizar a partir distintos mecanismos, dependiendo de cada tipo de ef. Por ejemplo el ef\_combo posee los siguientes mecanismos:

- Lista de opciones: Las opciones son estáticas y se definen en el mismo editor.
- Consulta SQL: Las opciones provienen de una consulta que se especifica en el mismo editor.
- Método PHP: Las opciones surgen de la respuesta de un método de una clase PHP.

La forma recomendada es utilizar la carga por Método PHP, para esto se necesita:

- Definir un método que retorne el conjunto de opciones disponibles en el combo
- Indicar en el editor-web que las opciones del combo se cargan con este método



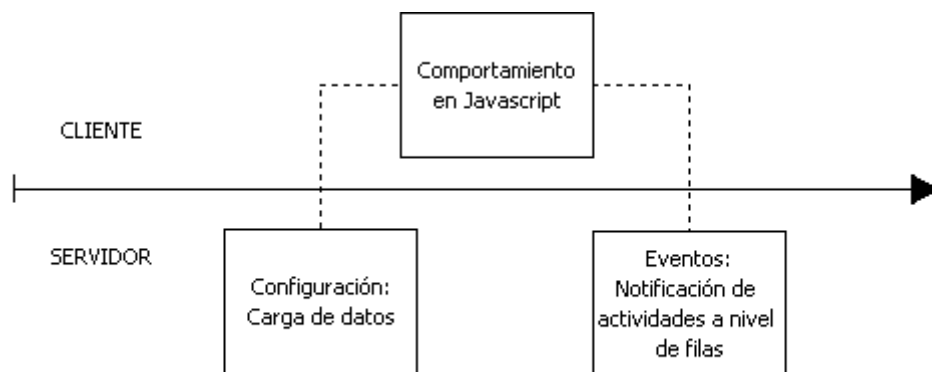
```
<?php
function alumnos_disponibles()
{
    return array(
        array('id' => 100, 'nombre' => 'Juan Perez'),
        array('id' => 142, 'nombre' => 'Cristian Fernandez'),
        .....
    );
?>
```

## 9. Cuadro

El cuadro es un elemento de interface (ei) que permite visualizar un conjunto de registros en forma de grilla. La grilla esta formada por una serie de filas o registros cada uno de ellos dividido en columnas.

Durante la configuración el cuadro se carga con un conjunto de datos y luego cuando vuelve al servidor informa si el usuario ha realizado alguna actividad sobre alguno de sus registros.

Fecha	Importe		
20/05/2004	\$ 12.500,00		
21/05/2004	\$ 22.200,00		
22/05/2004	\$ 4.500,00		
20/05/2005	\$ 12.500,00		
21/05/2005	\$ 22.200,00		
22/05/2005	\$ 4.500,00		
\$ 78.400,00			



### Configuración y Eventos

En la etapa de configuración es donde el cuadro necesita ser cargado con datos. Para esto requiere una estructura del tipo *recordset* que no es más que una matriz filas por columnas, el mismo formato que utiliza SQL en las respuestas de las consultas. un arreglo asociativo `id_ef=>estado`, se le dice *estado* al valor que toma el ef actualmente, independientemente de su formato.

Por ejemplo para cargar el formulario de la imagen:

Fecha	Importe		
20/05/2004	\$ 12.500,00		
21/05/2004	\$ 22.200,00		
22/05/2004	\$ 4.500,00		
20/05/2005	\$ 12.500,00		
21/05/2005	\$ 22.200,00		
22/05/2005	\$ 4.500,00		
\$ 78.400,00			

```
<?php
...
function conf__cuadro(toba_ei_cuadro $cuadro)
{
    $datos = array(
        array( "fecha" => "2004-05-20", "importe" => 12500),
        array( "fecha" => "2004-05-21", "importe" => 22200),
```

```

        array( "fecha" => "2004-05-22", "importe" => 4500),
        array( "fecha" => "2005-05-20", "importe" => 12500),
        array( "fecha" => "2005-05-21", "importe" => 22200),
        array( "fecha" => "2005-05-22", "importe" => 4500)
    );
    $cuadro->set_datos($datos);
}
...
?>

```

El cuadro tiene la capacidad de enviar eventos relacionados con una fila específica de la grilla, por ejemplo la selección con la *lupa*. En este caso el evento informa la clave de la fila seleccionada:





```

<?php
...
function evt__cuadro__seleccion($seleccion)
{
    print_r($seleccion);
}
...
Array ( [fecha] => 2005-05-21)
?>

```

#### . Paginado y ordenamiento

Fecha ▲ ▼	Importe ▲ ▼
1-03-2006	99
2-03-2006	98
3-03-2006	97
4-03-2006	96
5-03-2006	95
6-03-2006	94
7-03-2006	93
8-03-2006	92
9-03-2006	91
10-03-2006	90



 Página 1 de 4
 


### Paginado

Cuando la cantidad de registros a mostrar en el cuadro es muy grande existe la posibilidad de dividir la visualización en distintas páginas. Este paginado permite al usuario avanzar y retroceder entre conjuntos de estos registros.

La división en páginas la puede hacer el mismo cuadro o hacerla manualmente el programador:

- A cargo del cuadro: La división en páginas se produce cuando el cuadro recibe los datos en la configuración. En el editor este modo de paginado se lo conoce como *Propio*. Es la opción más adecuada cuando el conjunto de registros a mostrar es pequeño.

- Manualmente: La división en páginas la realiza el programador desde el CI contenedor. Al cuadro sólo llega los datos de la página actual y la cantidad total de registros. Esta forma es la más eficiente ya que sólo se consultan los registros a mostrar.

## Ordenamiento

El cuadro también ofrece al usuario la posibilidad de ordenar el conjunto de datos por sus columnas. Al igual que el paginado existen dos posibilidades de ordenamiento:

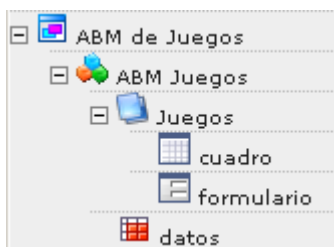
- El CI contenedor. Si así se decide se debe escuchar el evento `evt__idcuadro__ordenar` que recibe como parametro un arreglo conteniendo el sentido y la columna del orden. Por ejemplo: `array('sentido' => 'asc', 'columna' => 'importe');`. Estas opciones deberían incidir en el mecanismo de recepción de datos (típicamente el ORDER BY de una consulta SQL).
- El mismo cuadro: En caso que el evento no se escuche, el cuadro tomará la iniciativa de ordenar por sí mismo el set de datos. Para esto debe tener el conjunto completo de datos. Si por ejemplo el cuadro está paginado y sólo se carga la página actual, el cuadro sólo podrá ordenar esa página



## 10. Construcción manual de ABMs

Ya se han presentado los componentes principales en forma individual, resta mostrar cómo integrarlos para formar una operación completa. El tipo de operación que se va a usar de ejemplo es de los llamados ABMs (Altas-Bajas y Modificaciones de una o varias entidades), comenzando por los llamados **simples**

Un ABM simple contiene las operaciones de alta, baja y modificación de una única tabla. La idea es utilizar un **cuadro** para listar los datos existentes en la tabla y un **formulario** para poder agregar, modificar o eliminar los registros individualmente. Estos dos componentes se encontrarán en una **pantalla** de un **ci**. Finalmente para transaccionar con la base de datos se utilizará un **datos\_tabla**.



Árbol de componentes en el editor

Interfaz de usuario de un ABM de Juegos. La parte superior muestra un título 'ABM de Juegos'. Debajo hay una tabla con dos columnas: 'Nombre' y 'Descripción'. La tabla contiene tres registros: 'Ajedrez', 'Go' y 'Reversi'. Debajo de la tabla hay un formulario para editar un registro. El formulario tiene dos campos: 'Nombre (\*)' y 'Descripción'. El campo 'Nombre (\*)' contiene el texto 'Go' y el campo 'Descripción' contiene el texto 'Juego de estrategia creado en China hace más de 4.000 años.'. En la parte inferior derecha del formulario hay dos botones: 'Eliminar' (con una X roja) y 'Modificación'.

Nombre	Descripción
Ajedrez	Juego de mesa por turnos, que se desarrolla sobre un tablero cuadrado compuesto por 64 casillas.
Go	Juego de estrategia creado en China hace más de 4.000 años.
Reversi	El reversi es un juego entre dos personas, que comparten 64 fichas iguales, de caras distintas, que se van colocando por turnos en un tablero dividido en 64 escaques.

Nombre (*)	Go
Descripción	Juego de estrategia creado en China hace más de 4.000 años.

Eliminar Modificación

Captura de la operación en ejecución

### Programación de un ABM Simple

Una vez definidos los componentes resta programar la lógica de la operación. En este caso la lógica es bien simple, sólo es necesario atender los eventos y configurar el cuadro y formulario. En los eventos se interactúa con el **datos\_tabla** que es quien en definitiva hace las consultas y comandos SQL.

#### Manejo del Cuadro

- Para cargar el cuadro con datos se hace una consulta directa a la base.

```
function conf__cuadro()  
{  
    $sql = "SELECT id, nombre, descripcion FROM ref_juegos";
```

```

        return toba::db()->consultar($sql);
    }

```

- Cuando del cuadro se selecciona un elemento, el `datos_tabla` se carga con ese elemento, marcando que a partir de aquí las operaciones de ABM se harán sobre este registro. En esta operación el registro cargado del `datos_tabla` funciona como un **cursor** que representa la fila actualmente seleccionada, si no está cargado, no hay selección y viceversa.

```

function evt__cuadro__seleccion($seleccion)
{
    $this->dep("datos")->cargar($seleccion);
}

```

## Manejo del Formulario

- Cuando el `datos_tabla` está cargado, es señal que del cuadro algo se seleccionó, entonces se dispone a cargar con estos datos. Se usa el método `get()` del `datos_tabla` porque se sabe de antemano que se va a retornar un único registro, si la cantidad puede ser mayor se necesita llamar al método `get_filas()`

```

function conf__formulario()
{
    if ($this->dep("datos")->esta_cargada()) {
        return $this->dep("datos")->get();
    }
}

```

- Cuando el usuario selecciona una acción sobre el registro cargado en el formulario, es necesario indicar la acción al `datos_tabla`, sincronizarlo con la base de datos (ejecutando los comandos SQL) y lo resetea para limpiar la selección:

```

function evt__formulario__alta($datos)
{
    $this->dep("datos")->nueva_fila($datos);
    $this->dep("datos")->sincronizar();
    $this->dep("datos")->resetear();
}

function evt__formulario__modificacion($datos)
{
    $this->dep("datos")->set($datos);
    $this->dep("datos")->sincronizar();
    $this->dep("datos")->resetear();
}

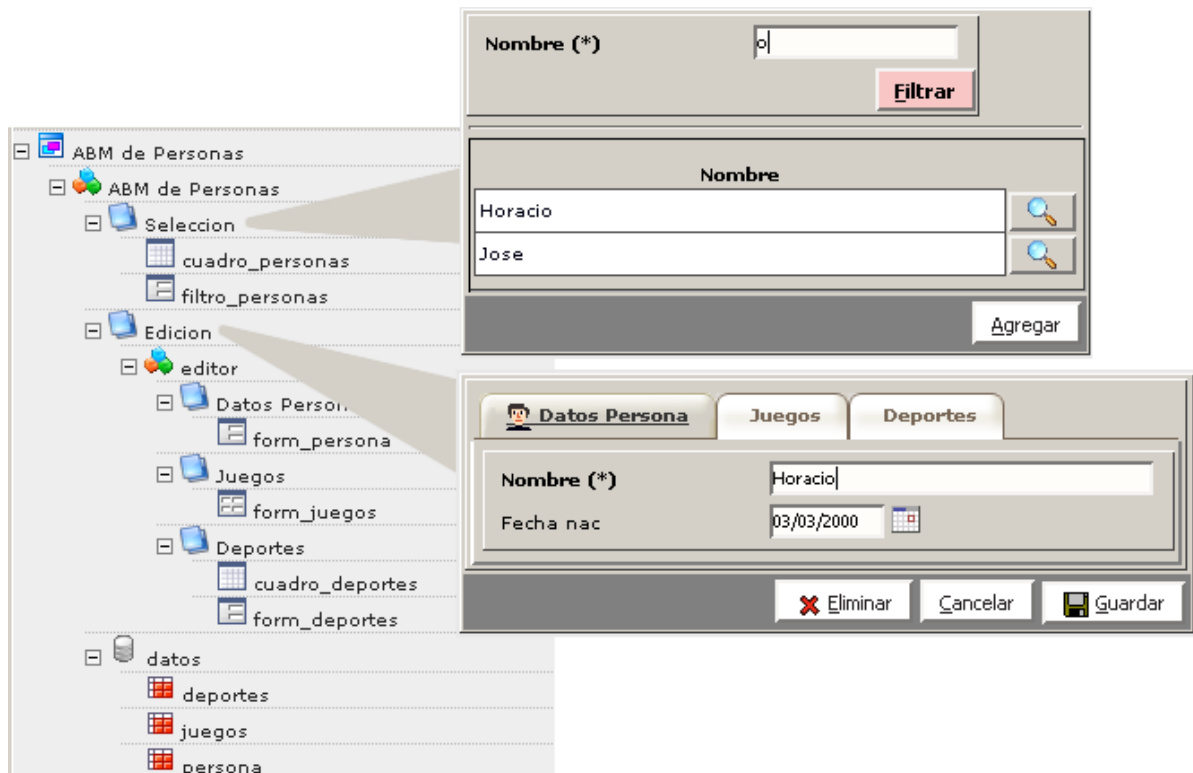
function evt__formulario__baja()
{
    $this->dep("datos")->eliminar_filas();
    $this->dep("datos")->sincronizar();
    $this->dep("datos")->resetear();
}

```

## Programación de un ABM Multitabla

Cuando la entidad a editar en el ABM se compone de más de una tabla, la operación se puede dividir en dos grandes etapas:

1. Elección entre editar un entidad existente o crear una nueva.
2. Edición de la entidad



Estas dos etapas se modelan como dos **pantallas** distintas del **CI** principal de la operación. La primer pantalla (la de *Navegación*) contiene sólo un **filtro** y un **cuadro** que permite al usuario seleccionar una entidad existente, pasando inmediatamente a modo edición. También se incluye un evento en esta pantalla que permite avanzar hacia el alta (Botón Agregar en el ejemplo).

A la segunda pantalla se le dice de *edición* y contiene en composición otro **CI** que tiene generalmente una pantalla por tabla involucrada en la entidad. Estas pantallas se muestran como solapas o tabs permitiendo al usuario navegar entre ellas e ir editando las distintas tablas que componen la entidad.

Además de la cantidad de componentes, la diferencia principal en el armado de esta operación es que no se transacciona con la base de datos hasta que el usuario en la pantalla de edición presiona el botón **Guardar**. Para soportar este requisito se va a usar una **Transacción a nivel operación**, vista en el capítulo de Persistencia. Las modificaciones, altas y bajas son mantenidas en memoria (sesión) hasta que el usuario presiona Guardar, donde se sincronizan con la base de datos.

### Extensión del Ci de Navegación/Selección

Una vez definidos los componentes se necesita programar la lógica del CI principal, es decir el que maneja la navegación y la transacción a alto nivel. Lo más interesante en este CI es atrapar los distintos eventos:

**Primera Pantalla**

**Segunda Pantalla**



Posibles eventos:

- Ingresar una condición al filtro, reduciendo el conjunto de datos que muestra el cuadro
- Seleccionar un elemento del cuadro, pasando a editar el elemento seleccionado
- Decidir Agregar un nuevo elemento, pasando a editar un elemento vacío inicialmente

Posibles eventos:

- Eliminar completo la entidad
- Cancelar la edición y volver a la pantalla anterior
- Guardar los cambios a la base de datos
- Cambiar de solapas y cambiar los datos, esto se delega a un CI anidado que se ve luego.

...(parte de la extensión del CI principal)...

//----- Pantalla seleccion -----//

```
function evt__filtro_personas__filtrar($datos)
```

```
{
```

```
    //Guardar las condiciones en una variable de sesion
```

```
    //para poder usarla en la configuracion del cuadro
```

```
    $this->s__filtro = $datos;
```

```
}
```

```
function evt__cuadro_personas__seleccion($id)
```

```
{
```

```
    $this->dep("datos")->cargar($id);    //Carga el datos_relacion
```

```
    $this->set_pantalla("edicion");    //Cambia a la pantalla de edición
```

```
}
```

```
function evt__agregar()
```

```
{
```

```
    $this->set_pantalla("edicion");    //Cambia a la pantalla de edición
```

```
}
```

//----- Pantalla edicion -----//

```
function evt__eliminar()
```

```
{
```

```
    //Elimina TODOS los datos de la relación y sincroniza
```

```
    $this->dep("datos")->eliminar();
```

```
    //Cambia a la pantalla de selección o navegación
```

```
    $this->set_pantalla("seleccion");
```

```
}
```

```

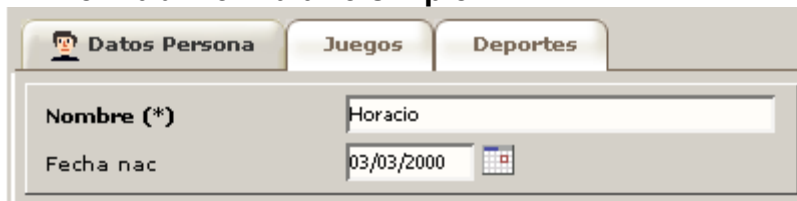
function evt__cancelar()
{
    //Limpia al CI anidado de edición
    $this->dep("editor")->disparar_limpieza_memoria();
    //Descarta los cambios en el datos_relacion
    $this->dep("datos")->resetear();
    //Cambia a la pantalla de selección o navegación
    $this->set_pantalla("seleccion");
}

function evt__procesar()
{
    //Limpia al CI anidado de edición
    $this->dep("editor")->disparar_limpieza_memoria();
    //Sincroniza los cambios del datos_relacion con la base
    $this->dep("datos")->sincronizar();
    $this->dep("datos")->resetear();
    //Cambia a la pantalla de selección o navegación
    $this->set_pantalla("seleccion");
}

```

## Extensión del Ci de Edición

### Primer Tab: Formulario simple



En esta solapa se encuentra un formulario con un evento implícito *modificacion*. Ya que se está editando la tabla cabecera de la relación (en este caso persona) sólo es posible que exista un único registro de esta tabla en la relación.

El método *set* del *datos\_tabla* está preparado para estos casos, si no existe el registro lo crea y si existe lo modifica. La carga en la configuración también es sencilla, con el método *get* se piden los datos del único registro, en caso de no existir este método retorna *null* mostrando el formulario vacío.

```

function conf__form_persona()
{
    return $this->get_relacion()->tabla("persona")->get();
}

function evt__form_persona__modificacion($registro)
{
    $this->get_relacion()->tabla("persona")->set($registro);
}

```

### Segundo Tab: Formulario ml

Juego(*)	Dia semana(*)	Hora inicio(*)	Hora fin(*)
Ajedrez	Lunes	17	19
Go	Jueves	05	14


En esta solapa, un formulario ML maneja las tres acciones (ABM) sobre una tabla de la relación (juegos de una persona). Lo interesante del formulario ML es que las acciones se realizan en javascript, informandolas al servidor como un bloque. El formato de la información que recibe el servidor es una matriz, donde por cada fila se informa su estado (A, B o M) junto con su nuevo valor (exceptuando la baja).

En lugar de recorrer esta estructura manualmente y con un case derivar cada acción de una fila a un método del `datos_tabla`, esta clase contiene un método *procesar\_filas* que lo hace internamente. Para la carga se utiliza el método *get\_filas* con su segundo parámetro en verdadero, indicando que las filas se retornen en una matriz asociativa cuya clave sea la clave interna de la fila. Esto permite que el ML y el `datos_tabla` mantengan los mismos valores de claves de las filas.

```
function conf__form_juegos()
{
    return $this->get_relacion()->tabla("juegos")->get_filas(null,true);
}

function evt__form_juegos_modificacion($datos)
{
    $this->get_relacion()->tabla("juegos")->procesar_filas($datos);
}
?>
```

### Tercer Tab: Cuadro y Formulario


**Datos Persona**

**Juegos**

**Deportes**

Deporte	Dia semana	Hora inicio	Hora fin	
Tenis	Jueves	17	19	
Basquet	Martes	17	19	

**Deporte (\*)**

Tenis

**Dia semana (\*)**

Jueves

**Hora inicio (\*)**

17

**Hora fin. (\*)**

19


**Eliminar**

**Modificacion**

**Cancelar**

La última solapa tiene un cuadro y un formulario que maneja las acciones sobre una tabla de la relación (deportes de una persona). La estrategia aquí es manejar la interface de una forma clásica, en donde en el cuadro se muestran las filas disponibles y al seleccionarlase pueden editar o borrar.

La primera forma de encararlo es mantener en sesión la fila seleccionada, si no existe tal fila indica que el formulario se debe mostrar vacío dando lugar a una alta. Cuando un atributo de la clase comienza con *s\_\_* indica que será mantenido en sesión, en este caso ese atributo es *\$s\_\_deporte*. Las operaciones de baja y modificación utilizan esta fila seleccionada como parámetro para los métodos del *datos\_tabla*.

```
protected $s__deporte;

function conf__cuadro_deportes()
{
    return $this->get_relacion()->tabla("deportes")->get_filas();
}

function evt__cuadro_deportes_seleccion($seleccion) {
    $this->s__deporte = $seleccion;
}

function conf__form_deportes()
{
    if(isset($this->s__deporte)) {
        return $this->get_relacion()->tabla("deportes")->get_fila($this->s__deporte);
    }
}

function evt__form_deportes_modificacion($registro)
{
}
```

```

        if(isset($this->s__deporte)){
            $this->get_relacion()->tabla("deportes")->modificar_fila($this->s__deporte, $registro);
            $this->evt__form_deportes__cancelar();
        }
    }

function evt__form_deportes__baja()
{
    if(isset($this->s__deporte)){
        $this->get_relacion()->tabla("deportes")->eliminar_fila( $this->s__deporte );
        $this->evt__form_deportes__cancelar();
    }
}

function evt__form_deportes__alta($registro)
{
    $this->get_relacion()->tabla("deportes")->nueva_fila($registro);
}

function evt__form_deportes__cancelar()
{
    unset($this->s__deporte);
}

```

### Tercer Tab: Forma alternativa

La alternativa a mantener la selección en una variable de sesión y luego usar la API del `datos_tabla` sobre esta fila (para obtener sus valores, modificarla o borrarla) es usar una utilidad del `datos_tabla` llamada **cursor**. El cursor permite apuntar a una fila particular, haciendo que ciertas operaciones utilicen esa fila como predeterminada. Por ejemplo si se fija el cursor en la fila 8 y luego, sea en el mismo u otro pedido de página, se hace un `set($datos)` en esa tabla, los datos afectan a esta fila.

El cursor es de suma utilidad cuando se trabaja con relaciones más complejas ya que permite fijar valores en ciertas tablas y operar en forma reiterada sobre registros relacionados de otras tablas, esto se verá más adelante.

```

function conf__cuadro_deportes()
{
    return $this->get_relacion()->tabla("deportes")->get_filas();
}

function evt__cuadro_deportes__seleccion($seleccion) {
    $this->get_relacion()->tabla("deportes")->set_cursor($seleccion);
}

function conf__form_deportes()
{
    if ($this->get_relacion()->tabla("deportes")->hay_cursor()) {
        return $this->get_relacion()->tabla("deportes")->get();
    }
}

function evt__form_deportes__modificacion($registro)
{
    $this->get_relacion()->tabla("deportes")->set($registro);
}

```



```
        $this->evt__form_deportes__cancelar();
    }

function evt__form_deportes__baja()
{
    $this->get_relacion()->tabla("deportes")->set(null);
    $this->evt__form_deportes__cancelar();
}

function evt__form_deportes__alta($registro)
{
    $this->get_relacion()->tabla("deportes")->nueva_fila($registro);
}

function evt__form_deportes__cancelar()
{
    $this->get_relacion()->tabla("deportes")->resetear_cursor();
}
```

## 11. Extensiones Javascript

En el capítulo de componentes se utilizó la extensión PHP para personalizar su comportamiento. Dentro de la extensión en PHP es posible modificar el comportamiento del componente en el cliente utilizando javascript. En este capítulo se va trabajar exclusivamente con la parte Javascript de los componentes, para esto Toba cuenta con una jerarquía de clases similar a la que existe en PHP, para profundizar sobre la API está disponible la [documentación javascript](#).

Es importante tener en cuenta la forma en la cual se extiende un componente en javascript. A continuación se muestra un código muy simple que agrega una confirmación en el cliente cuando el usuario clickea Guardar:

```
class ci_X extends toba_ci
{
    function extender_objeto_js()
    {
        echo "
            {$this->objeto_js}.evt_guardar = function() {
                return prompt(\"Desea Guardar?\");
            }
        ";
    }
}
```

El código muestra que el método PHP a extender es **extender\_objeto\_js()** dentro del cual es necesario insertar el código Javascript. Este lenguaje no soporta clases en la forma convencional de los lenguajes Java o PHP, por lo cual no se *hereda* del componente sino que directamente se lo cambia, esto es por ejemplo si se quiere agregar un método a un objeto *mi\_componente* se hace definiendo *mi\_componente.tal\_metodo = function() { var i = 20; ...}*.

Entonces en la extensión PHP, se extiende la **clase** (por ejemplo *toba\_ei\_formulario*) mientras que en la de Javascript se extiende el **objeto** puntual. El nombre de este objeto es desconocido al programador (se compone del id del componente) por lo que es necesario pedirselo a la clase PHP por eso se hace *{ \$this->objeto\_js }.metodo = ....*

Finalmente cabe recalcar que las extensiones javascript se hacen dentro del mismo componente por motivos de orden y modularidad, pero no es necesariamente la única forma ya que el javascript en definitiva forma parte del HTML resultante de la operación, si se mira el código fuente de la página HTML se podrá ver la extensión y su entorno.

### Atrapando Eventos

Así como el evento en PHP significa la interacción del usuario con el servidor, en Javascript existe el mismo criterio, sólo que es la previa de esta interacción. Un evento antes de viajar al servidor escucha un *listener* en javascript. Por ejemplo si un formulario dispara un evento *modificacion*, en la extensión del mismo formulario se puede atrapar el método *evt\_modificacion* y retornar *true/false* para permitir o no la ejecución del evento (entre otras cosas que se pueden hacer).

Un ejemplo real es el siguiente javascript perteneciente al catalogo de items del editor, la idea es que cuando el usuario presiona el botón *Sacar Foto* se le pregunte en Javascript el nombre que toma la foto y luego se tome la foto en el servidor. En this.\_parametros se guarda el valor del parámetro que termina en el servidor (en este caso el método evt\_\_X\_\_sacar\_foto(\$nombre\_foto))

```
{ $this->objeto_js }.evt__sacar_foto = function() {  
    this._parametros = prompt("Nombre de la foto","nombre de la foto");  
    if (this._parametros != "" && this._parametros != null) {  
        return true;  
    }  
    return false;  
}
```

## Redefiniendo métodos

La opción a atrapar eventos predefinidos es redefinir el comportamiento de métodos ya existentes. En general no es una metodología recomendada pero a veces es necesaria para casos no contemplados.

Al no existir la herencia clásica, la redefinición del método tiene que simularla manualmente, esto es guardar el método viejo y definir el nuevo llamando cuando sea necesario al viejo. Vemos un ejemplo:

```
{ $this->objeto_js }.iniciar_viejo = { $this->objeto_js }.iniciar;  
  
{ $this->objeto_js }.iniciar = function() {  
    //Extensión  
    this.iniciar_viejo(); //Llamada al original  
    //Extensión  
}
```

## 12. Otras partes del API

### Fuentes de Datos

Cuando se utilizan los componentes de datos (`datos_tabla` y `datos_relacion`) se hace uso implícito de la base de negocios definida en el editor. A esta base se la denomina **fente de datos** y puede ser accedida a través de una clase de toba.

Para obtener el objeto que representa la conexión con la base de datos se utiliza el método `toba::db($fuente)`, si no se brinda el nombre de la fuente se utiliza la predeterminada del proyecto. Veamos un ejemplo de una consulta ad-hoc, donde el formato del resultado es una matriz filas x columnas (también llamado RecordSet):

```
$sql = "SELECT id, nombre FROM tabla..."
$rs = toba::db()->consultar($sql);
if (! empty($rs)) {
    foreach ($rs as $fila) {
        echo "{$fila['id']} es {$fila['nombre']} ";
    }
} else {
    echo "No hay datos!"
}
//--- Si la consulta falla (por ej. no existe la tabla), tira una excepcion toba_error_db
```

Para los comandos SQL lo que se retorna es el número de registros afectados:

```
$sql = "UPDATE tabla SET nombre = id";

toba::db()->abrir_transaccion();
$cant = toba::db()->ejecutar($sql);
....
toba::db()->cerrar_transaccion();

echo "Se modificaron $cant registros";
//--- Si el ejecutar falla (por ej. una restricción de clave foránea), tira una excepcion toba_error_db
```

### Más info

- [Fuentes de Datos](#)<sup>Wiki</sup>
- [toba\\_db](#)

### Vinculación entre Items

Existe una clase que permite crear links entre items incluso de distintos proyectos, esta clase recibe el nombre de **vinculador**. La utilidad es poder navegar hacia otras operaciones pasando parámetros, o hacia la misma operación accediendo de otra forma.

Mostramos algunos ejemplos:

```
//Forma general
toba::vinculador()->crear_vinculo($proyecto, $item, $parametros, $opcione
```

```

s);

//Crea un vínculo al ítem 23421 de este proyecto con un dato como parámetro
o
$vinculo = toba::vinculador()->crear_vinculo(null, 23421, array("moneda" => "dolar"));

//Crea un vínculo al ítem actual
$vinculo = toba::vinculador()->crear_autovinculo($parametros, $opciones);

echo "<a href='{$vinculo}'>Navegar</a>";

```

La interfaz completa de la API está publicada [toba\\_vinculador](#)

## Memoria

Sabemos que las aplicaciones desarrolladas PHP en general no mantienen información en memoria entre dos pedidos de página. La forma de recordar información de cada usuario es guardarla en un arreglo global de nombre `$_SESSION`, garantizando que todo lo que allí se guarda estará disponible a los pedidos siguientes.

Otra forma de compartir información entre pedidos es a través de la URL o datos de formularios, estos dos conceptos también tienen un arreglo global, `$_GET` para los datos que provienen de URL y `$_POST` para los de formularios. La diferencia principal de estos con respecto a la sesión es que son datos no-seguros (pueden haber sido modificados o leídos por terceros) y que sólo perduran un pedido de página.

Lo que propone Toba son mecanismos de almacenamiento de información más sofisticados para la aplicación. La idea es dividir los datos según su alcance, serían tres niveles donde se tiene garantizado que la información es segura tanto que no se puede modificar ni leer por terceros:

- Datos **globales a la aplicación**, que por lo general se mantienen durante toda la sesión del usuario, como por ejemplo la Universidad a la cual pertenece.
- Datos **globales a una operación**, información que sólo interesa dentro de la operación actual, que al cambiar de ítem será descartada.
- Datos que sólo se almacenan para el **siguiente pedido** de página.

Generalmente en las operaciones el manejo de los datos se hace en forma interna, por ejemplo no tenemos que guardar manualmente cual es la solapa actualmente seleccionada sino que el componente es quien lo guarda y consulta internamente. La API para almacenar información personalizada tiene razón de ser cuando esta información es del propio dominio de la aplicación. Para cubrir estos tres niveles la clase [toba\\_memoria](#) brinda estas primitivas:

```

//--- Guardar el par $clave=>$valor por lo que resta de la sesión
toba::memoria()->set_dato_aplicacion($clave, $valor);

//--- Guardar el par $clave=>$valor por lo que resta de la operación
toba::memoria()->set_dato_operacion($clave, $valor);

//-- Para estos tipos de almacenamiento se utiliza este método para consultarlos posteriormente
toba::memoria()->get_dato($clave);

//--- Guardar el par $clave=>$valor para el siguiente pedido de página

```

```
toba::memoria()->set_dato_sincronizado($clave, $valor);
//--- Recuperar este valor
toba::memoria()->get_dato_sincronizado($clave);
```

## Logger


Toba cuenta con una clase que va recolectando información interna y se almacena en un archivo común de logs del proyecto ubicado en *\$toba\_dir/instalacion/i\_\_X/p\_\_Y/logs*, donde X es la instancia e Y es el proyecto. Los programadores también pueden utilizar este log para guardar información de debug del sistema. Para esto se consume la clase [toba\\_logger](#):

```
//--- Guardar un mensaje de debug
toba::logger()->debug($mensaje);

//--- Guardar un error
toba::logger()->error($mensaje);

//--- Guardar el valor de una variable
toba::logger()->var_dump($variable);

//--- Guardar una traza completa de llamadas
toba::logger()->trace();
```

El archivo de logs generado puede ser analizado con una operación del editor creada para ayudar al desarrollo. Este analizador puede ser accedido a través del ícono .

## Mensajes y Notificaciones

Para centralizar el manejo de mensajes y permitir su posterior personalización Toba brinda la posibilidad de definir los mensajes en el mismo editor web y posteriormente instanciarlos y notificarlos usando la API.

Una vez creados los mensajes en el editor es posible recuperarlos en ejecución usando la clase [toba\\_mensajes](#):

```
//Suponiendo que el mensaje ingresado es: 'Esta es la %1% instancia de un
mensaje global de Toba. Fecha de hoy: %2%.'
$mensaje = toba::mensajes()->get("indice", array("primera", date("d/M/Y")));
echo $mensaje;
//La salida es: 'Esta es la primera instancia de un mensaje global de Toba
. Fecha de hoy: 01/02/2007.'
```

En lugar de mostrar el mensaje con un simple *echo* es posible notificarlo utilizando la clase [toba\\_notificacion](#):

```
toba::notificacion()->agregar($mensaje);
toba::notificacion()->agregar($mensaje, "info");
```

**Se han encontrado los siguientes problemas:**

---



Esta es la primer instancia de un mensaje global de Toba. Fecha de hoy: 14/Feb/2007.



Esta es la primer instancia de un mensaje global de Toba. Fecha de hoy: 14/Feb/2007.

---

Aceptar

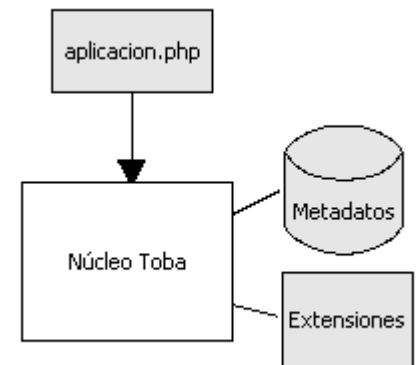
**Más info**

- [Mensajes y Notificaciones](#)<sup>Wiki</sup>
- [Ejemplo](#)

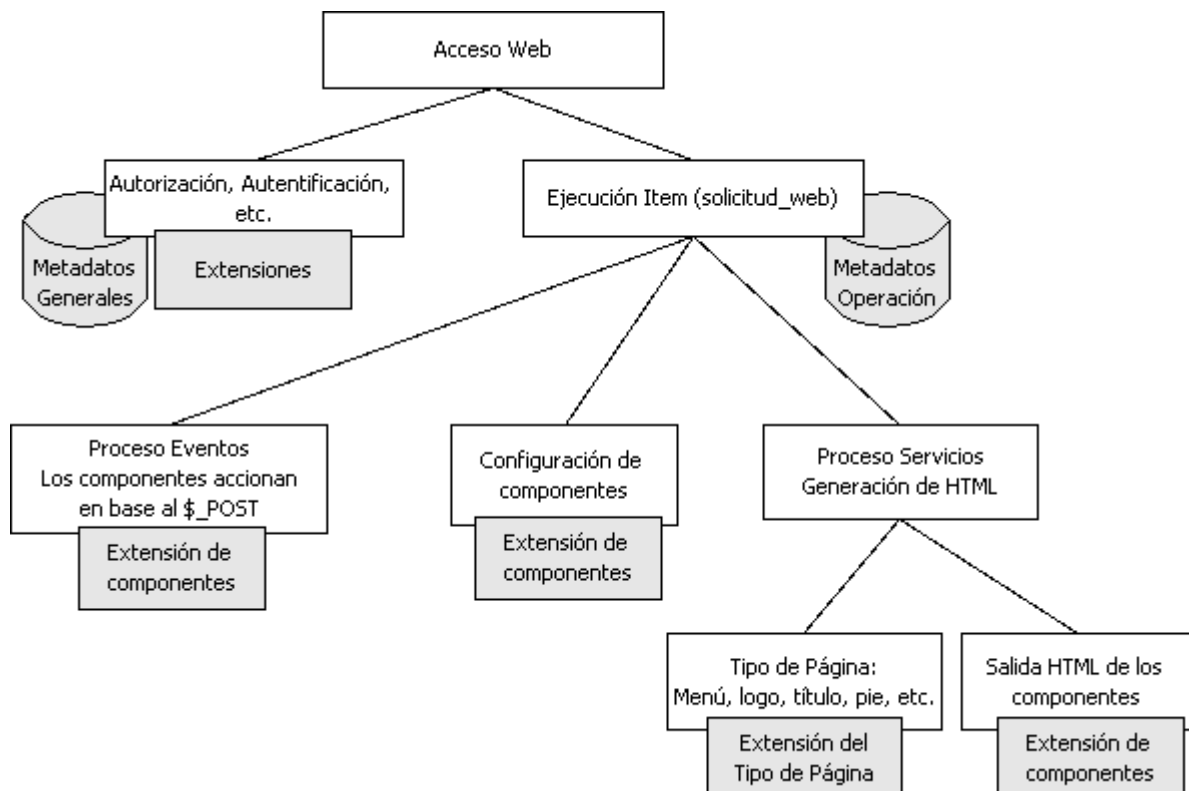
## 13. Esquema de Ejecución

En la introducción vimos que existe un *núcleo* o *Runtime* encargado de la ejecución. A diferencia de una librería clásica, no existe el concepto de procedimiento principal o *main* en el cual el programador incluye las librerías y las consume. En Toba la situación es distinta:

- El proyecto brinda un *punto de acceso* en donde se incluye al núcleo de toba (generalmente es el archivo `www/aplicacion.php`).
- A partir de allí el núcleo analiza los **metadatos** del proyecto y de la operación puntual que se ejecuta, activando los componentes acordes.
- Si alguna clase del runtime o algún componente se encuentra extendido por el proyecto, recién allí el programador puede incluir código propio, siempre enmarcado en un 'plan maestro' ya delineado.



Lo más interesante para mostrar en este tutorial es cómo el proyecto puede variar el comportamiento en ejecución. En el siguiente gráfico se muestra un mayor detalle de la ejecución resaltando en gris los puntos donde el proyecto tiene el control de la ejecución, ya sea con **metadatos** o con **extensión** de código.





## 14. Administración Básica

Para utilizar toba se necesita una **Instalación** que es ni más ni menos el sistema de archivos conteniendo una versión de Toba. Dentro de este sistema de archivos se encuentra una serie de **Proyectos** en sus respectivas versiones. Tanto la instalación de toba como los proyectos además de contener el código fuente contienen los metadatos de las operaciones, de esta forma se puede distribuir un proyecto llevando tanto el código como los metadatos a un sistema de producción.

Para modificar estos metadatos en forma grupal y a través de un editor es necesario cargar estos metadatos desde el sistema de archivos hacia una base de datos, esta base se denomina una **Instancia Toba**. Así el sistema puede tener distintas instancias como producción, desarrollo, testing, pruebas\_varias, etc.

Durante la ejecución, el usuario ingresa una URL en el navegador, allí se liga a una instalación, instancia y opciones específicas de ejecución. Esta unión entre URL y entorno de ejecución se lo conoce como Punto de Acceso.

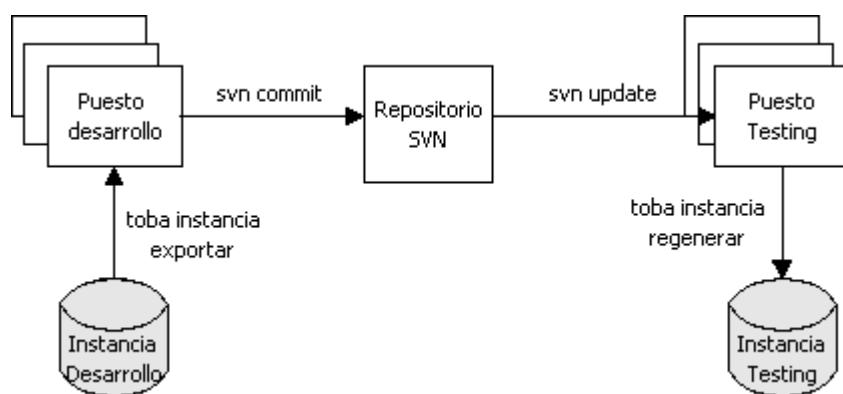
### Metadatos

Cuando utilizamos el **editor web** de Toba, estamos definiendo un proyecto en base a **metadatos**, almacenados en una base de datos definida durante la instalación.

Lo positivo de esto es que, al estar centralizada, es posible que un grupo de desarrollo localizado en la misma red pueda desarrollar sobre esta base en forma simultánea. Además se puede utilizar SQL tanto para manipular como para obtener los metadatos.

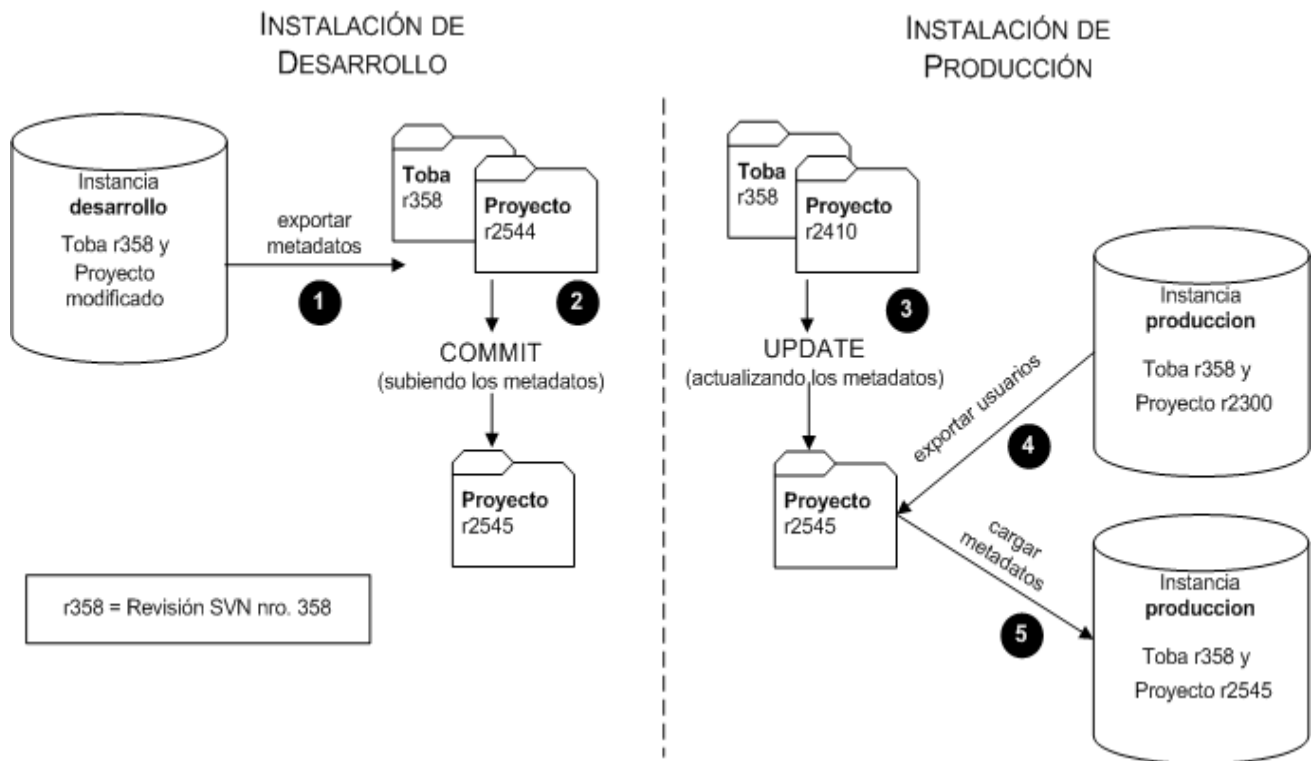
Lo negativo es que mientras estos metadatos no sean exportados al sistema de archivos no podrán ser compartidos con otros grupos de trabajo o dentro de un mismo grupo geográficamente distante. Esta necesidad de importar - exportar metadatos se cubre usando los **comandos de consola**. Como introducción a estos comandos necesitamos presentar dos:

- toba instancia exportar: Exporta desde la base hacia el sistema de archivos
- toba instancia regenerar: Importa desde el sistema de archivos hacia la base



## Pasaje Desarrollo - Producción

El escenario más común es tener una única instancia de desarrollo y actualizar otra instancia (generalmente Producción o Testing) de manera periódica. Este escenario puede describirse en cinco pasos:



1. **Desarrollo - Exportación de metadata:** Siempre en desarrollo los últimos metadata se encuentran en la base de datos (la instancia) y no en el sistema de archivos, esto es lógico porque los desarrolladores estuvieron utilizando el Editor en forma conjunta con esta instancia. La exportación se hace de la siguiente forma:

```
toba proyecto exportar -p mi_proyecto
```

2. **Desarrollo - Commit de los nuevos archivos:** El resultado de la exportación es la actualización de los archivos ubicados en la carpeta *metadata* del proyecto. Junto a las modificaciones de código existentes se suben las modificaciones (commit) creando una nueva revisión, almacenada en un repositorio central.
3. **Producción - Actualización de la nueva revisión:** Suponiendo que se toma la decisión de actualizar el sistema en producción, el primer paso es actualizar el proyecto en cuestión, esto traerá los metadata y el código actualizado.
4. **Producción - Exportación de información local:** Lo único que se necesita de la instancia en producción son sus usuarios, logs y demás metadata considerados locales o propios de la instancia, para el resto de los metadata valen los de desarrollo. Entonces es necesario exportar los datos locales de esta instancia almacenándose en el sistema de archivos:

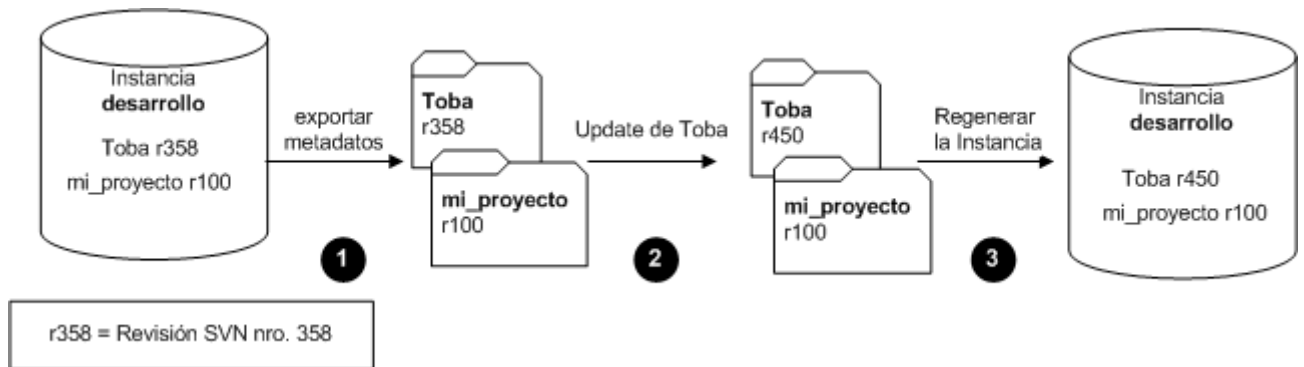
```
toba instancia exportar_local
```

5. **Producción - Regenerar el proyecto:** Ahora que tanto los metadata de desarrollo como los de los usuarios se encuentran en el sistema de archivos es posible regenerar el proyecto:

```
toba proyecto regenerar -p mi_proyecto
```

## Actualización de Toba

Muchas veces es necesario actualizar la versión de toba con la que se está desarrollando (arreglo de bugs, nuevas funcionalidades deseadas, etc.), para esto hay que tener el cuidado de exportar los metadatos previo a la actualización, ya que actualizar toba muchas veces (no todas) implica regenerar la instancia.



1. Exportar Metadatos del proyecto: Como los últimos metadatos se encuentran en la base de datos (la instancia) es necesario llevarlas al sistema de archivos y **versionarlos**. Para cada proyecto propio ubicado en esta instancia se debe hacer:

```
toba proyecto exportar -p mi_proyecto
```

2. Actualizar Toba: Se actualiza la nueva revisión de Toba desde el repositorio SVN.
3. Actualización de metadatos: Es posible que la nueva versión requiera alguna migración automática (ver [Versiones Toba](#)), para ello ejecutar:

```
toba instalacion migrar
```

4. Por si la migración afecta algún metadato del proyecto es necesario volver a exportar al sistema de archivos:

```
toba proyecto exportar -p mi_proyecto
```

5. Con los nuevos metadatos de toba y los metadatos exportados del proyecto en el sistema de archivos es necesario volver a cargar la instancia:

```
toba instancia regenerar
```

Cabe notar que instancia regenerar como primer paso borra el contenido de la base, por lo que si la exportación no se efectuó o se efectuó con errores se perderán los datos del proyecto y sólo será posible recuperarlos desde alguna revisión del SVN del proyecto o algún backup manual. Hay una razón por la cual la exportación se debe hacer antes que la actualización de código. Es posible que el nuevo código de toba haga referencia a nuevas columnas o nuevos campos no presentes en la instancia actual del proyecto (porque fue creada usando un toba más viejo). Si se quiere hacer la exportación luego de la actualización, este nuevo código va a querer exportar esta estructura de base de datos nueva no presente en la instancia actual, imposibilitando la extracción de los metadatos hacia el sistema de archivos. Hay una solución a este error, siempre y cuando no se haya ejecutado el tercer paso por la razón explicada en el párrafo anterior. La solución es volver atrás la revisión de toba a la misma con la que se generó la instancia y reiniciar el proceso.



