PHP Connector

DHTMLX connector is a helper server-side library that enables access to external datasources and backend systems.
It supports all DHTMLX components operated on data.
The library itself provides wide server-side functionality, such as data loading (static or dynamic), saving, filtration, validation etc. Cooperation with DHTMLX Dataprocessor allows you to manage insert/delete/update operations in addition.

Release notes

new in version 1.5

Security:

- Protection from XSS attacks;
- Preventing CSRF and XSRF attacks.

Flexibility:

- New methods for rendering data (support for stored procedures calls, loading from data arrays);
- The possibility to set userdata with all types of connectors;
- The possibility to rename the query status parameter '!nativeeditor_status';
- Better OOP support (support for data model and data behavior classes);
- Better Touch Support (improved data formats, adding extra tags/sections to data);
- Support for PHP frameworks (CodeIgniter, YII, CakePHP).

new in version 1.2

- SQLite support
- DataConnector, JSONDataConnector
- onDBError event

server-side requirements

- Fully supported DataBases: MySQL, PostgreSQL, Oracle, MSSQL, FileSystem
- Partially supported DataBases: SQLite, any PDO|ODBC compliant (MSAccess, Excel, SQLLite, DB2, FireBird, etc.)
- PHP version: 5.x

supported functionality

- Data loading for Grid, TreeGrid, Tree, Combo, Scheduler, DataView, Chart, Form
- Data saving for Grid, TreeGrid, Tree, Scheduler, DataView, Form
- Server-side sorting and filtering for Grid, Treegrid
- Dynamic loading (paging) for Grid, Treegrid, Tree, Scheduler, DataView, Combo

migration from older versions

0.9 → 0.96

- *beforeFilter* and *beforeSorting* events provide object as parameter instead of plain data
- Order of parameters for render_* methods of TreeGrid and Tree connectors was changed.

```
//0.9
render_table("table","id","fields","parent id")
//1.0
render_table("table","id","fields","extra fields","paren
```

0.96 → 1.0

- *beforeOutput* event has different parameters

1.0 → 1.5

- new version filter out html content before saving, use the next code if you want to revert logic of connector to the previous behavior

```
    ConnectorSecurity::$xss = DHX_SECURITY_TRUSTED;
```

GUIDES

Please note, throughout the guides you'll see examples that relate to grid. If you use a different component that supports mentioned functionality, just replace GridConnector with the appropriate type. See a list of connector types here.

General idea of dhtmlxConnector

dhtmlxConnector is a server-side library that lets you access data sources. It consists of individual component-specific connectors and a set of additional ones providing advanced functionality.

A dhtmlxConnector's job is to provide necessary data exchange conditions so that you do not have to deal with the technical details of working with various data stores, systems or services. As such, each type of dhtmlxConnector is designed to use a specific API.

As the library is server-side it provides different manipulations just on server backend:

- loading data (statically or dynamically) from database, File System, Excel file.
- server-side filtration, sorting and validation.

When you need to deal with client-side as well, i.e. pass data back to server, you should use DataProcessor additionally (for more details of this topic, see the chapter 'Client-side requirement - dataProcessor').

general list of connectors

Component-specific:

- **ChartConnector** (relates to dhtmlxChart)
- **ComboConnector** (relates to dhtmlxCombo)
- **DataViewConnector** (relates to dhtmlxDataView)
- **FormConnector** (relates to dhtmlxForm)
- **GridConnector** (relates to dhtmlxGrid)
- **SchedulerConnector** (relates to dhtmlxScheduler)
- **TreeConnector** (relates to dhtmlxTree)
- **TreeGridConnector** (relates to dhtmlxTreeGrid)

Additional (most important):

- JSONDataConnector/DataConnector (relates to **touch-oriented libraries**: DHTMLX Touch, DataTable, dhtmlXDataStore)
- KeyGridConnector
- SelectOptionsConnector
- TreeGroupConnector/TreeGridGroupConnector
- TreeMultitableConnector

Throughout the documentation you'll see examples that use 'GridConnector'. Subject to using component, replace it with the needed connector.

JSONDataConnector/DataConnector

Common

There are 2 connectors you can use for touch-oriented DHTMLX libraries (such as DHTMLX Touch, DataTable, DataStore):

- **DataConnector**
- **JSONDataConnector**

They both work the same way, but differ in type of returned data.

```php
<?php
require_once("../../connector/data_connector.php");
require_once("../../connector/db_sqlite.php");

if (!$db = sqlite_open('db', 0777, $sqliteerror)) {
        die($sqliteerror);
}
$data = new JSONDataConnector($db,"SQLite");
$data->render_table("users", "id", "name,age,city");
?>
```

DataConnector

DataConnector generates 'XML' data feed as in:

```xml
<data>
   <item id="1" title="War and Peace"  author="Leo Tolstoy"
   <item id="2" title="Hamlet"         author="William Shak
   <item id="3" title="Madame Bovary"  author="Gustave Flau
</data>
```

JSONDataConnector

JSONDataConnector generates 'JSON' data feed as in:

```
[
   { id:"1", title:"War and Peace",    author:"Leo Tolstoy"
   { id:"2", title:"Hamlet",           author:"Shakespeare"
   { id:"3", title:"Madame Bovary",    author:"Gustave Flau
]
```

Adding tags/sections to data

Starting from version 1.5, there is a possibility to add the first child tags, sections to data returned by the connector. To add the tag(section), use method add_section.

DataConnector

For example, if you call add_section as follows:

```
$data = new JSONDataConnector($db,"SQLite");
$data->add_section("config", "some_data");
$data->add_section("config2", "<column>data1</column><colum
```

DataConnector produces the next data:

```
<data>
    <config>some_data</config>
    <config2><column>data1</column><column>data2</column></
    <item id="1" attr1="qwe" attr2="asd" />
    <item id="2" attr1="qwe" attr2="asd" />
</data>
```

JSONDataConnector

If you call add_section as follows:

```
$data = new JSONDataConnector($db,"SQLite");
$data->add_section("config", "'some_data'");
$data->add_section("config2", "{ column1: 'data1'}");
```

JSONDataConnector produces the next data:

```
{
    data: [
      { 'id': '1', 'attr1':'qwe', 'attr2':'asd'},
      { 'id': '2', 'attr1':'qwe', 'attr2':'asd'}
    ],
    config: 'some_data',
    config2: { 'column1':'data1' },
}
```

Simple Data Transfer

JSONDataConnector/DataConnector supports simplified protocol for CRUD requests.

- get all data

```
GET  data.php?action=get
```

server response - collection of json (xml) objects

- get info for some specific record

```
GET   data.php?action=get&id=123
```

response - json(xml) object

- delete

```
GET  data.php?action=delete
POST id=123
```

server response - 'true'/'false'

- insert

```
GET  data.php?action=insert
POST id=123&some=value&other=value
```

response on success

```
true
<new_id>
```

or

```
false
```

on failed insert operation

- update

```
GET  data.php?action=delete
POST id=123&some=value&other=value
```

response - 'true'/'false'

SelectOptionsConnector

SelectOptionsConnector is used by the dhtmlxForm component to fill a list of options for the 'select' item.
Don't confuse SelectOptionsConnector with OptionsConnector. The second one is an auxiliary connector and used just in the context of dhtmlxGrid and dhtmlxScheduler components. SelectOptionsConnector is an independent connector able to generate output XML data.

So, to define options of the **select** form's item you should make the following:

1. On client-side you should specify the parameter 'connector':

```
var formData = [
                {type: "select", label: "Categories",
                {type: "button", value: "Proceed"}
];
myForm = new dhtmlXForm("form_container", formData);
```

2. On server-side your code will look like:

```
<?php
require_once("../codebase/connector/options_connector
$res=mysql_connect("localhost","root","");
mysql_select_db("tasks");

$data = new SelectOptionsConnector($res, "MySQL");
$data->render_table("categories","id","value, label")

?>
```

Render_table method takes the following parameters:

- 'categories' - the name of table.
- 'id' - the id column
- 'value' - the column that will be used as values of options.
- 'label' - the column that will be used as labels of options.

Initialization

To use functionality of DHTMLX Connector, first you should initialize it. Generally, both client- and server-sides are concerned in it.

client-side

if you need just load data from database (with or without data preprocessing)

1. Specify connector file in load (or loadXML) method of a component.

```
//index.html
myGrid = new dhtmlXGridObject("someContainer");// ini
...
myGrid.load("my_connector.php");
```

if you need to perform any update operations

1. Include connector.js file (located in dhtmlxConnector_php/codebase).
2. Specify connector file in load (or loadXML) method of a component.
3. Initialize dhtmlxDataProcessor on client-side (read more about it here).

```
//index.html
<script src="codebase/connector/connector.js"></scrip
myGrid = new dhtmlXGridObject("someContainer");// ini
...
myGrid.load("my_connector.php");
myDP = new dataProcessor("myconnector.php");// initia
myDP.init(myGrid);
```

Samples of client-side initialization for all components

server-side

1. Include appropriate connector file into the page.
2. Create Database connection.
3. Instantiate connector object. Linking variable is a mandatory parameter in all constructors. The second parameter(database type)- optional. By default “MySQL”. Other possible variants see here.
4. The last step is data configuration.

```
//my_connector.php
require("connector/grid_connector.php");// connector
$res=mysql_connect("localhost","root","");// db conne
mysql_select_db("myDatabase"); // db connection
$gridConn = new GridConnector($res,"MySQL"); // conne
$gridConn->render_table("mytable","item_id","item_nm,
```

Samples of server-side initialization for all components

Summarized components functionality

- Chart
- Combo
- DataStore
- DataView
- Form
- Grid
- Tree
- TreeGrid
- Scheduler

- DHTXML Touch Components

CHART

Functionality supported by dhtmlxConnector:

- Data loading
- Formatting/changing data before loading
- Dynamic loading
- Server-side logging

COMBO

Functionality supported by dhtmlxConnector:

- Dynamic loading
- Formatting/changing data before loading
- Server-side filtration
- Server-side logging
- Server-side sorting
- Static loading from database tables, File System, Excel files

Functionality supported by dhtmlConnector & dataProcessor:

- Client-side logging
- Client-side validation
- Elementary DB actions with dhtmlxConnector
- Making complex queries
- Making simple queries
- Multi-user sync
- Server-side validation

DATASTORE

Functionality supported by dhtmlxConnector:

- Dynamic loading
- Formatting/changing data before loading
- Server-side filtration
- Server-side logging
- Server-side sorting
- Static loading from database tables, File System, Excel files

Functionality supported by dhtmlConnector & dataProcessor:

- Client-side logging
- Client-side validation
- Elementary DB actions with dhtmlxConnector
- Making complex queries
- Making simple queries
- Server-side validation

DATAVIEW

Functionality supported by dhtmlxConnector:

- Dynamic loading
- Formatting/changing data before loading
- Server-side filtration
- Server-side logging
- Server-side sorting
- Static loading from database tables, File System, Excel files

Functionality supported by dhtmlConnector & dataProcessor:

- Client-side logging
- Client-side validation
- Elementary DB actions with dhtmlxConnector
- Making complex queries
- Making simple queries
- Multi-user sync
- Server-side validation

FORM

Functionality supported by dhtmlxConnector:

- Dynamic loading
- Formatting/changing data before loading
- Server-side filtration
- Server-side logging
- Server-side sorting
- Static loading from database tables, File System, Excel files

Functionality supported by dhtmlConnector & dataProcessor:

- Client-side logging
- Client-side validation
- Elementary DB actions with dhtmlxConnector
- Making complex queries
- Making simple queries
- Multi-user sync
- Server-side validation

GRID

Functionality supported by dhtmlxConnector:

- Defining grid structure on server-side
- Dynamic loading
- Formatting/changing data before loading
- Loading from tables that don't contain identity field
- Select/combo box columns in Grid
- Server-side data export to PDF, Excel
- Server-side filtration
- Server-side logging
- Server-side sorting
- Static loading from database tables, File System, Excel files

Functionality supported by dhtmlConnector & dataProcessor:

- Client-side logging
- Client-side validation
- Elementary DB actions with dhtmlxConnector
- Making complex queries
- Making simple queries
- Multi-user sync
- Server-side validation
- Transactions

DEFINING GRID STRUCTURE ON SERVER-SIDE

APPLICABLE TO: Grid

Starting from version 1.0, connectors can be used to define header of grid.
It can be done in 2 ways:

- automatic
- manual

automatic defining

In case automatic defining, grid will use names of table's fields as labels for the columns.

```
require("../../codebase/grid_connector.php");
$grid = new GridConnector($res);

$grid->set_config(new GridConfiguration());
$grid->render_table("grid50000","item_id","item_nm,item_cd"
```

If you want to apply automatic server-side sorting and filtration for specified columns, you should specify *true* inside *GridConfiguration*:

```
$grid->set_config(new GridConfiguration(true));
$grid->render_table("grid50000","item_id","item_nm,item_cd"
```

manual defining

In manual mode, headers and their parameters are defined by php command. Names of commands mimic names of *js* commands with similar functionality.

```
$config = new GridConfiguration();
$config->setHeader(array("column 1","column 2"));
$config->setColTypes(array("ro","ed"));

$grid->set_config($config);

$grid->render_table("grid50000","item_id","item_nm,item_cd"
```

available commands

All commands below, get as input parameter an array of values or comma-separated string, delimited by 'header delimiter' (default value - ',').

```
// column labels
$config->setHeader($names);
// column types
$config->setColTypes($typeStr);
// column IDs
$config->setColIds($idsStr);
// column width, int values, will be processed as size in p
$config->setInitWidths($widths);
// column width, int value, will be threated as size in perc
$config->setInitWidthsP($widths);
// column align
$config->setColAlign($alStr);
// column sorting type
$config->setColSorting($sortStr);
// column color
```

```
$config->setColColor($colorStr);
// visibility of column
$config->setColHidden($hidStr);//'true' if column must be h
```

header and footer

- To attach header to the grid you should use attachHeader() method:

  ```
  $config->attachHeader($values, $styles = null);
  ```

  Parameters:

  - array or string of header names. In case of string, names are delimited by 'header delimiter' (default value -',')
  - array or string of header styles. In case of string, styles are delimited by 'header delimiter' (default value - ',')

- To attach footer to the grid you should use attachFooter() method:

  ```
  $config->attachFooter($values, $styles = null);
  ```

  Parameters:

  - array or string of footer names. In case of string, names are delimited by 'header delimiter' (default value - ',')
  - array or string of footer styles. In case of string, styles are delimited by 'header delimiter' (default value - ',')

header delimiter

Header delimeter sets symbol or several symbols which will be used as delimiter in string arguments.

```
$config->setHeaderDelimiter($headerDelimiter);
```

Related samples

- defining grid structure on server-side

KeyGridConnector

Normal grid connector expects that target table has some autogenerated ID field, but, in some cases, data needs to be loaded and saved from database table which doesn't have such autofilled ID.
Normal grid connector can be used for data loading from such tables, but won't be able to save data back. KeyGridConnector can resolve such problem (use it instead of GridConnector):

```
$grid = new KeyGridConnector($res);
$grid->render_table("mytable","name","name,address,phone");
```

So, there are two differences from the normal GridConnector:

- name of connector class (GridConnector→KeyGridConnector).
- one of data fields is used as ID

SELECT/COMBO BOX COLUMNS IN GRID

When on client-side you specify co/coro/combo columns (combo-box/select) in grid, grid will automatically request data for them in the same manner as for filtration options.

By default, grid will use DISTINCT select against related field, and fetch all possible options.
If you need to define custom list of options you can use one of 2 ways:

- **hardcoded list**

```
$grid->set_options("item_nm",array("1" => "one", "2"=>"two"
$grid->render_table("grid50","item_id","item_nm,item_cd");
```

- **list created on base of different table**

```
$options = new OptionsConnector($res);
$options->render_table("countries","country_id","country_id
$grid->set_options("item_nm",$options);

$grid->render_table("grid50","item_id","item_nm,item_cd");
```

co/combo type

- when using predefined list of data, both values and labels need to be provided
- when using connector, there must be two fields selected, one as (value), second as (label)

coro type:

- only labels need to be provided

- connector may define only label parameter

TREE

Functionality supported by dhtmlxConnector:

- Dynamic loading
- Formatting/changing data before loading
- Loading tree data from multiple tables
- Loading tree data from tables that don't contain relation field
- Marking an item as a leaf or branch
- Server-side filtration
- Server-side logging
- Server-side sorting
- Static loading from database tables, File System, Excel files

Functionality supported by dhtmlConnector & dataProcessor:

- Client-side logging
- Client-side validation
- Elementary DB actions with dhtmlxConnector
- Making complex queries
- Making simple queries
- Multi-user sync
- Server-side validation

TREEGRID

Functionality supported by dhtmlxConnector:

- Dynamic loading
- Formatting/changing data before loading
- Loading tree data from multiple tables
- Loading tree data from tables that don't contain relation field
- Marking an item as a leaf or branch
- Server-side filtration
- Server-side logging
- Server-side sorting
- Static loading from database tables, File System, Excel files

Functionality supported by dhtmlConnector & dataProcessor:

- Client-side logging
- Client-side validation
- Elementary DB actions with dhtmlxConnector
- Making complex queries
- Making simple queries
- Multi-user sync
- Server-side validation

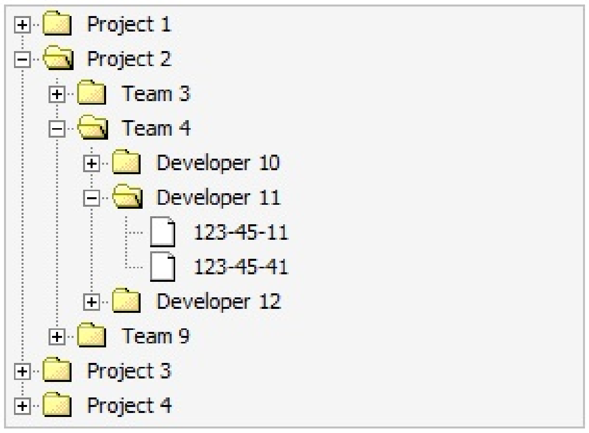
TreeMultitableConnector/TreeGridMultitableCor

Connectors will work only if dynamic loading mode on client-side is enabled.

In many cases, tree or treegrid need to be built from multiple tables (each table represent separate level of tree/treegrid). For this purpose should be used TreeMultitableConnector/TreeGridMultitableConnector (instead of TreeConnector/TreeGridConector).

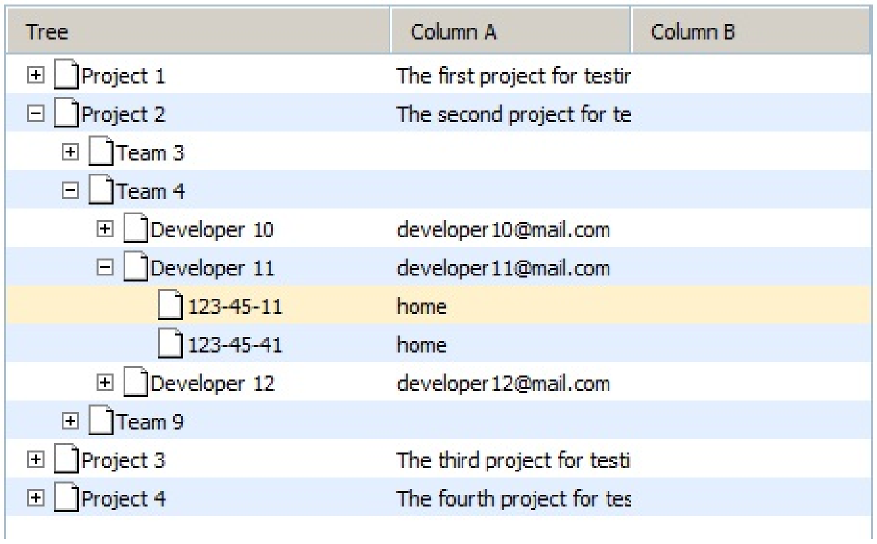
tree

```
require("../../codebase/treemultitable_connector.php");

$tree = new TreeMultitableConnector($res);
$tree->setMaxLevel(3);
$level = $tree->get_level();

switch ($level) {
        case 0:
                $tree->render_table("projects2","project_id
                break;
        case 1:
                $tree->render_sql("SELECT teams2.team_id, t
                break;
        case 2:
                $tree->render_table("developers2", "develop
                break;
        case 3:
                $tree->render_table("phones2", "phone_id",
                break;
}
```



treeGrid

```
require("../../codebase/treegridmultitable_connector.php");

$treegrid = new TreeGridMultitableConnector($res);
$treegrid->setMaxLevel(3);
$level = $treegrid->get_level();
...
```



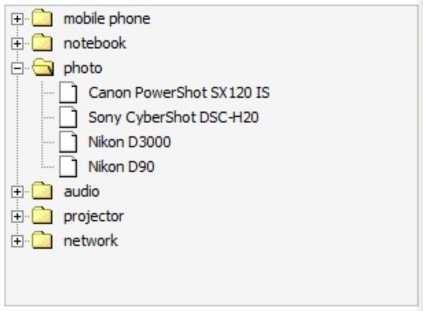
Tree	Column A	Column B
⊞ Project 1	The first project for testir	
⊟ Project 2	The second project for te	
⊞ Team 3		
⊟ Team 4		
⊞ Developer 10	developer10@mail.com	
⊟ Developer 11	developer11@mail.com	
123-45-11	home	
123-45-41	home	
⊞ Developer 12	developer12@mail.com	
⊞ Team 9		
⊞ Project 3	The third project for testi	
⊞ Project 4	The fourth project for tes	

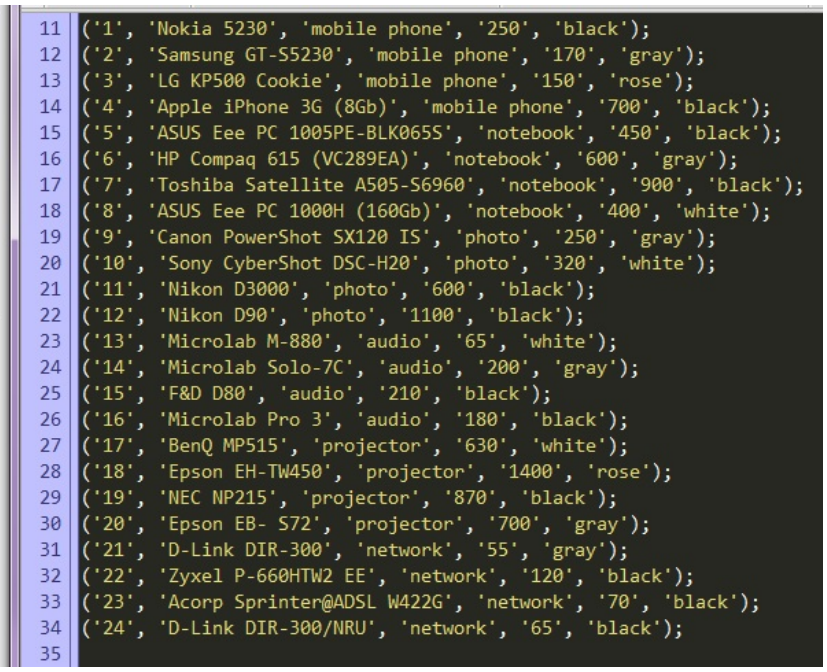
TreeGroupConnector/TreeGridGroupConnector

Default connector for Tree and TreeGrid expects that data in DB is stored with child-parent key relation. But, in some cases, tree or treegrid need to be shown from plain data (by grouping it). Such task can be resolved with help of the TreeGroupConnector or TreeGridGroupConnector.

For example, a table 'products' grouping by category. As result, the top level are 'categories', the second level - 'products', related to the category.

```
//tree
require("./codebase/connector/tree_group_connector.php");

$tree = new TreeGroupConnector($res);
$tree->render_table("products2", "id", "product_name", "",

//treeGrid
<code php>
require_once('../../codebase/treegridgroup_connector.php');

$treegrid = new TreeGridGroupConnector($res,"MySQL");
$treegrid->render_table("products2", "id", "product_name,sca
```




```
('1', 'Nokia 5230', 'mobile phone', '250', 'black');
('2', 'Samsung GT-S5230', 'mobile phone', '170', 'gray');
('3', 'LG KP500 Cookie', 'mobile phone', '150', 'rose');
('4', 'Apple iPhone 3G (8Gb)', 'mobile phone', '700', 'black');
('5', 'ASUS Eee PC 1005PE-BLK065S', 'notebook', '450', 'black');
('6', 'HP Compaq 615 (VC289EA)', 'notebook', '600', 'gray');
('7', 'Toshiba Satellite A505-S6960', 'notebook', '900', 'black');
('8', 'ASUS Eee PC 1000H (160Gb)', 'notebook', '400', 'white');
('9', 'Canon PowerShot SX120 IS', 'photo', '250', 'gray');
('10', 'Sony CyberShot DSC-H20', 'photo', '320', 'white');
('11', 'Nikon D3000', 'photo', '600', 'black');
('12', 'Nikon D90', 'photo', '1100', 'black');
('13', 'Microlab M-880', 'audio', '65', 'white');
('14', 'Microlab Solo-7C', 'audio', '200', 'gray');
('15', 'F&D D80', 'audio', '210', 'black');
('16', 'Microlab Pro 3', 'audio', '180', 'black');
('17', 'BenQ MP515', 'projector', '630', 'white');
('18', 'Epson EH-TW450', 'projector', '1400', 'rose');
('19', 'NEC NP215', 'projector', '870', 'black');
('20', 'Epson EB- S72', 'projector', '700', 'gray');
('21', 'D-Link DIR-300', 'network', '55', 'gray');
('22', 'Zyxel P-660HTW2 EE', 'network', '120', 'black');
('23', 'Acorp Sprinter@ADSL W422G', 'network', '70', 'black');
('24', 'D-Link DIR-300/NRU', 'network', '65', 'black');
```

SCHEDULER

Functionality supported by dhtmlxConnector:

- Select box columns
- Dynamic loading
- Formatting/changing data before loading
- Server-side filtration
- Server-side logging
- Server-side sorting
- Static loading from database tables, File System, Excel files

Functionality supported by dhtmlConnector & dataProcessor:

- Client-side logging
- Client-side validation
- Elementary DB actions with dhtmlxConnector
- Making complex queries
- Making simple queries
- Multi-user sync
- Server-side validation

Select-box columns in scheduler

When scheduler has a select-like collections or unit/timeline view - it's possible to load a list of related options from a server. In such case the data loading need to be defined on client-side as:

```
some_collection=scheduler.serverList("type");
```

On server-side , list of options can be retrieved in 2 ways:

- **hardcoded list**

```
//both values and labels need to be provided
$scheduler->set_options("type",array("1" => "one", "2"=>
$scheduler->render_table("events","event_id","start_date
```

- **list created on base of different table**

```
$options = new OptionsConnector($res);
//there must be two fields selected, one as (value), sec
$options->render_table("types","type_id","type_id(value)
$scheduler->set_options("type",$options);

$scheduler->render_table("events","event_id","start_date
```

DHTMLX Touch Components

Functionality supported by dhtmlxConnector:

- Dynamic loading
- Formatting/changing data before loading
- Server-side filtration
- Server-side logging
- Server-side sorting
- Static loading from database tables, File System, Excel files

Functionality supported by dhtmlConnector & Touch dataProcessor (differs from standard dataProcessor) see in the related chapter of DHTMLX Touch documentation.

Basic loading

In this chapter you'll find base information concerning static loading data from:

- database table
- File System
- PHP array
- Excel file

In order to data is loaded correctly, your 'id' field in the database must be autoincrement.

loading from database table

APPLICABLE TO: Grid, TreeGrid, Tree, Combo, Scheduler, DataView, Chart, Form, DataStore, DHTMLX Touch components

Loading characteristics are defined at stage of component configuration. There are 3 ways to specify the desired data:

- by render_table() method when you work with one table
- by render_sql() method when you work with several tables
- by render_complex_sql() method when you use to call stored procedures in the database + to implement complex queries

work with one table

When all necessary data is placed in one database table you should use the render_table() method:

```
$grid->render_table("grid50","item_id","item_nm,item_cd", "
```

Parameters:

- **database table name**
- **name of identity field** (optional)
- **list of fields** which should be used as values of component item (cells of grid, item label in tree, text of option in combo)
- **list of extra fields** (know more about extra fields here)

If you want to render all fields from DB (except for the key field), you can use a simplified command:

```
$grid->render_table("grid50");
```

That's enough to make connector implement select, insert, update and delete operations.

work with several tables

If your SQL statement contains more than one table, connector won't be able to generate insert/update/delete operations correctly and you will need to do one from the next:

1. Define sql for each operation manually
2. Use server-side events to define your own processing logic
3. Define different configs for select and update operations

The 3rd approach is shown in the code snippet below:

```
if ($grid->is_select_mode())//code for loading data
    $grid->render_sql("Select * from tableA, tableB  where
else //code for other operations - i.e. update/insert/delete
    $grid->render_table("tableA","id","name,price");
```

With such init code grid will be loaded with three columns of data from 2 tables, but during saving only data from the first table will be saved.

complex queries

You are allowed to use any SQL statements to populate a dhtmlx component through dhtmlxConnector. For complex SQL queries we strongly encourage you to use the render_complex_sql() method:

```
$grid->render_complex_sql("SELECT name from tableA WHERE de
```

Parameters:

- ***sql* statement**
- **name of identity field** (optional)
- **list of fields** which should be used as values of component item (cells for grid, item label for tree, text of option for combo)
- **list of extra fields** (know more about extra fields here)
- **parent ID field name** for hierarchical structures (required for tree and treegrid)

In case your SQL query was against a single table, it is quite probable that insert/update/delete operations do not require any additional code. dhtmlxConnector will parse your SQL and generate insert/update/delete statements based on used table and fields names.

calling stored procedures

Starting from version 1.5, you can load data using in SQL statements stored procedures. The library provides a special method for this need - render_complex_sql.

```
$grid->render_complex_sql("exec usp_findusers 'John'", "id"
```

extra data

The last parameter of render_array, render_sql, render_complex_sql, render_table methods allows you to define a list of fields which will be extracted from database table but won't be sent to client-side.
These fields can be used as attributes or flags, mapped to different properties of records (userdata, row styles, images, etc.).

```
$grid->render_table("tableA","id","name,price","extra1,extr
// or
```

```
$grid->render_sql("Select * from tableA, tableB  where  tab
```

extra1 and extra2 fields will be available in all server-side events but won't be sent to client-side, and won't be included in update|insert operations.

tree and treegrid specificity

In case of Tree and TreeGrid , render_array, render_sql, render_complex_sql and render_table accept one more parameter - the relation ID. For default treegrid hierarchy - it's the name of a field , which will be used to link parent and child records.

```
$treeGrid->render_table("tableA","id","name,price","","pare
// or
$treeGrid->render_sql("Select * from tableA, tableB  where
```

aliases

To make usage of extracted data handier you can use aliases for DB field names (makes sense only if you use server-side events):

```
$grid->render_table("tableA","id","name,price(product_price
// or
$grid->render_sql("Select *,tableA.id as aid from tableA, t
```

Back to top

loading from PHP array

Starting from version 1.5, instead of database you can load data from a PHP array which can be filled by any kind of external logic.
To load data from a PHP array, use method render_array. Beware, the method can't be used for storing data but you still can use event handlers, or custom model to intercept data saving command and process them in some custom way.

Parameters:

- **name of PHP array or a PHP array itself**
- **name of identity field** (optional)
- **list of fields** which should be used as values of component item (cells for grid, item label for tree, text of option for combo)
- **list of extra fields** (know more about extra fields here)
- **parent ID field name** for hierarchical structures (required for tree and treegrid)

```
$data = array(
        array("id" => "1", "product" => "Phone AB12",    pr.
        array("id" => "2", "product" => "Tablet device", pr.
);
$conn->render_array($data, "id", "product,price");
```

Back to top

loading from File System

APPLICABLE TO: Grid, TreeGrid, Tree, Combo, Scheduler, DataView, Chart, Form

Starting from version 1.0, dhtmlxConnector allows to use FileSystem as datasource (please note, to start use this functionality you should include *db_filesystem.php* file located in connector's package).

```
require("./codebase/connector/db_filesystem.php");
require("./codebase/connector/grid_connector.php");

$grid = new GridConnector("", "FileSystem");
$grid->render_table("../","safe_name","filename,full_filena
```

In the code snippet above, grid is filled with info about files located in 'd:/www' folder

Parameters of 'render-table' method:

- **folder**, for which data listing is required
- **field's id**. Leave it empty or use safe_name as ID of file
- **list of fields**, possible values are:
  - ***filename*** - name of file
  - ***full_filename*** - full path to file
  - ***size*** - size of file in bytes
  - ***name*** - name part of file name
  - ***extension*** - extension part of file name
  - ***date*** - timestamp of file
  - ***is_folder*** - file|folder flag

limiting files in output

There are 3 ways to limit files in output:

1. by extension type
2. by regexp pattern
3. by meta-type

by extension type:

```
$fileTypes = FileSystemTypes::getInstance();
$fileTypes->addExtention('png');
```

by regexp pattern:

```
$fileTypes = FileSystemTypes::getInstance();
$fileTypes->addPattern('/^.+\..*$/');
```

by meta-type:

The following meta-types can be used:

- **image** - image files
- **document** - doc, xls, txt, rtf
- **web** - php, html, js, css
- **audio** - mp3, wav, ogg
- **video** - avi, mpg, mpeg, mp4
- **only_dir** - folders

```
$fileTypes = FileSystemTypes::getInstance();
$fileTypes->setType('web');
```

Back to top

loading from Excel file

APPLICABLE TO: Grid, TreeGrid, Tree, Combo, Scheduler, DataView, Chart, Form

Starting from version 1.0, dhtmlxConnector allows to use Excel file as datasource.

To start use this functionlity you should:

1. Download phpExcel.

   It isn't included in the connector's package. You can grab it from 2 sources:
   - http://support.dhtmlx.com/x-files/connector/phpExcel.zip
   - http://www.codeplex.com/PHPExcel.
2. Unzip library to the connector's folder.
3. Include related files:
   - '*lib/PHPExcel.php*' (phpExcel package)
   - '*lib/PHPExcel/IOFactory.php*' (phpExcel package)
   - '*db_excel.php*' (standard connector's package)

```
//files from libExcel package
require_once('lib/PHPExcel.php');
require_once('lib/PHPExcel/IOFactory.php');

//connectors
require("../../codebase/db_excel.php");
require("../../codebase/grid_connector.php");

$grid = new GridConnector("../some.xls", "ExcelDBDataWrappe
$grid->render_table("A18:F83", "id", "A,B,C,D,E,F");
```

Parameters:

- constructor ***GridConnector()***
  - **name of excel file**, absolute path or path related to the php file
  - **database type**

- ***render_table()*** method
  - **first parameter** can be
    - range of cells in the spreadsheet, for example A18:F83
      ```
      $grid->render_table("A18:F83", "id", "A,B,C,D,E,F'
      ```
    - number of the top row, from which output need to be started, for example A4
      ```
      $grid->render_table("A1", "id", "A,B,C,D,E,F");
      ```
    - '*', which means include all not empty rows
      ```
      $grid->render_table("*", "id", "A,B,C,D,E,F");
      ```
  - **name of identity field**. You can use 'id' value for auto id generation
  - **list of fields** which will be used as columns in the grid

loading both data and header

Applicable to: Grid

When you need to load both data and header from excel file, you can make it through GridConfiguration:

```
$grid = new GridConnector($excel_file, "Excel");

$config=new GridConfiguration();
//array of cells, with labels for grid's header
$config->setHeader($grid->sql->excel_data(array("A3","B3","
$grid->set_config($config);
```

```
$grid->render_table("A18", "id", "A,B,F");
```

Back to top

Dynamic loading

Dynamical Loading mode allows to load data not all at once, but partially, by client-side request. It decreases initial loading time and loading of server.

To work correctly, on client-side should be enabled related mode:

- **grid** - *smart rendering* and *paging* modes
- **treegrid** - *dynamic branch loading* mode
- **tree** - *dynamic branch loading* mode
- **combo** - *partial autocomplete* (you don't need it for normal autocomplete)
- **dataview** - *dynamic scrolling* or *dynamic paging*

To activate the mode you should use the method dynamic_loading():

```
$grid->dynamic_loading([$rowsNum]);
```

In 'dynamic loading' mode you can't use GROUP BY within SQL query

Parameters:

- *none* for a tree,treegrid.
- *number of rows* which should be initially loaded (the value should be more than a number of rows visible in the grid, or at least any positive number) for a grid.
- *maximum number of options* which server will send to a combo in the 'autocomplete mode' for a single data request.

tree/treegrid specific

Normally, connector makes all operations automatically, and doesn't need customization.
But, in case of dynamic loading in tree/treeGrid, database can contain field indicated leaf or branch the current item is. In beforeRender event handler you are allowed to mark an item as a leaf or branch (it decreases a number of SQL queries and increases perfomance).

```
function custom_define($item){
                if ($item->get_value("is_a_branch"))
                        $item->set_kids(true);
                else
                        $item->set_kids(false);
}
$tree->event->attach("beforeRender","custom_define");
```

The same approach can be used for non-dynamical mode of tree/treeGrid as well. It's not obligatory but lets you increase data generation performance.

Filtering

There are 3 ways to implement server-side filtration:

- by specifying additional parameters in URL
- by using in-header filter types while component configuration
- by means of beforeRender event

Beware, server-side filtration of dhtmlxTreeGrid doesn't maintain open states.

URL manipulation

APPLICABLE TO: Grid, TreeGrid, Tree, Combo, Scheduler, DataView, Chart, Form, DataStore, DHTMLX Touch components

You can control which data is loaded into component by specifying additional parameters in URL.
Check article Extending functionality for full url parameters description.

- ***filtration by one field***

```
grid.load("some.php?connector=true&dhx_filter[1]=mask");
```

With such url - data will be taken with additional rule.

```
WHERE field_for_column_1 LIKE %mask%
```

- ***filtration by multiple fields***

```
grid.load("some.php?connector=true&dhx_filter[1]=mask&dh
```

For components other than dhtmlxGrid you should provide exact field name instead of column index.

Beware that such filtering is not-secure and if you need to hide some data - be sure to define such limitation on server side, not as part of URL.

in-header filter types

APPLICABLE TO: Grid, TreeGrid, Combo

To enable server-side filtering you can use one of the following in-header filter types while configuring dhtmlxGrid/dhtmlxTreeGrid on client-side:

- ***#connector_text_filter*** - text filter. Retrieves values which contain mask defined through text field
- ***#connector_select_filter*** - select filter. Retrieves values which contain mask defined through dropdown list of possible values

```
mygrid.setHeader("Column A, Column B");
mygrid.attachHeader("#connector_text_filter,#connector_sele
```

text filter

Text filter usage doesn't require any additional configuration code. Grid/TreeGrid will automatically send data about new entered text and filter server-side data using %mask% pattern.
If you need to change filtering pattern or implement more advanced logic - beforeFilter event should be used.

- ***default filtration logic***

```
function custom_filter($filter_by){
          //WHERE some_field LIKE 'value'
          if (!sizeof($filter_by->rules))
               $filter_by->add("some_field","value","LIK
}
$conn->event->attach("beforeFilter","custom_filter");
```

- ***redefined filtration logic***

```
function custom_filter($filter_by){
      if (!sizeof($filter_by->rules))
               $filter_by->add("some_field","value","LIK
          //change WHERE some_field LIKE '%value%' to th
          $index = $filter_by->index("some_field");
          if ($index!==false)  //there is client side in
               $filter_by->rules[$index]["operation"]=">
}
$conn->event->attach("beforeFilter","custom_filter");
```

Through *rules[$index]* you can refer to:

- the name of a field (*rules[$index][“name”]=“age”*)
- the value of a field (*rules[$index][“value”]=“30”*)
- the type of an operation (*rules[$index][“operation”]=”>”*)

select filter

By default, grid/treegrid will use DISTINCT select against related field, and fetch all possible options.
If you need to define custom list of options you can use one of 2 ways:

- ***hardcoded list***

```
$grid->set_options("item_nm",array("1" => "1", "2"=>"two
$grid->render_table("grid50","item_id","item_nm,item_cd"
```

- ***list created on base of different table***

```
$filter1 = new OptionsConnector($res);
$filter1->render_table("countries","country_id","country
$grid->set_options("item_nm",$filter1);
```

```
$grid->render_table("grid50","item_id","item_nm,item_cd"
```

You can use both *render_table* and *render_sql* methods for OptionsConnector object, the same as for any normal connector.

Beware that name of field, used in select filter need to have alias **(value)**.

custom filters (using beforeRender event)

APPLICABLE TO: Grid, Combo

By using beforeRender event it's possible to define filtration rules as PHP code (doesn't work for dyn. modes).

```
function custom_filter($data){
     if ($data->get_value("some")<0)
          $data->skip(); //not include in output
}
$conn->event->attach("beforeRender","custom_filter");
```

Formatting/changing data before loading

base formatting methods

APPLICABLE TO: Grid, TreeGrid, Tree, Combo, Scheduler, DataView, Chart, Form, DataStore, DHTMLX Touch components

When you need to update values which were returned from database table or set some specific formatting before sending data to client-side, you should use the beforeRender event handler.

common usage

```
$res=mysql_connect($mysql_server,$mysql_user,$mysql_pass);
mysql_select_db($mysql_db);
require("../../codebase/grid_connector.php");

function color_rows($row){
        if ($row->get_index()%2)
        $row->set_row_color("red");
}

$grid = new GridConnector($res);
$grid->event->attach("beforeRender","color_rows");
$grid->render_table("grid50000","item_id","item_nm,item_cd"
```

- ***color_rows*** function sets colors for rows subject to their indices
- during data generation, for each record outputed for client-side beforeRender event will be executed, i.e. ***color_rows*** function will be called for each record
- ***$row*** is an instance of GridDataItem object related to the current record

date formatting

APPLICABLE TO: Grid, TreeGrid, Tree, Combo, Scheduler, DataView, Chart, Form, DateStore, DHTMLX Touch Components

```
function formatting($row){
                  //render field as details link
                  $data = $row->get_value("some_field");
                  $row->set_value("some_field","<a href='de

                  //formatting date field
                  $data = $row->get_value("other_field");
                  $row->set_value("other_field",date("m-d-Y
}

$grid = new GridConnector($res);
$grid->event->attach("beforeRender","formatting");
```

- ***get_value*** and ***set_value*** methods allow you to get or set value of any field related to the record (it doesn't affect actual values in DB)
- If alias was used during data configuration - you need to use it instead of real db field name as the first parameter of get|set command.

using extra fields

APPLICABLE TO: Grid, TreeGrid, Tree, Combo, Scheduler, DataView, Chart, Form, DataStore, DHTMLX Touch components

More complex formating rules can be defined by using extra fields while configuration.

```
function formatting($row){
           //set row color
           $row->set_row_color($row->get_value("color"));
           //save in userdata
           $row->set_userdata("some_data",$row->get_value("
}

$grid = new GridConnector($res);
$grid->event->attach("beforeRender","formatting");
$grid->render_table("some_table","id","name,price","color,c
```

- field color isn't outputed to client-side but used to define property of row.
- during update|insert operation only *'name'* and *'price'* columns may be changed, *'color'* will stay untouched.
- *'count'* field will be sent to client-side as userdata of the row and it will be possible to access it on client-side through related data.

tree/treegrid specificity

APPLICABLE TO: Tree, TreeGrid

treeGrid provides TreeGridDataItem and tree provides TreeDataItem as input parameter of beforeRender event handler. Both of them support base operations and few specific ones.

```
function custom_format($item){
                if ($item->get_value("complete")>75)
                        $item->set_check_state(1);

                if ($item->get_value("duration")>10)
                        $item->set_image("true.gif");
                else
                        $item->set_image("false.gif");
}
$tree->event->attach("beforeRender","custom_format");
```

- ***set_image*** method allows to set image of tree element (for treegrid it accepts the only parameter, while for tree it can be 3 different images for 3 states of tree's item)
- ***set_check*** method exists only in TreeDataItem object and allows to set the state of related checkbox (tree need to have checkboxes enabled in js. configuration code as well)
- beforeRender event can be used in dynamic Tree and TreeGrid to define which elements of hierarchy are branches and which are leafs (see details here).

While deleting items that have children you can face the problem that the parent item is deleted but children are not. In this case you should use the beforeDelete event:

```
function beforeDeleteFunc( $data ) {
   // custom logic
};
$conn->event->attach("beforeDelete","beforeDeleteFunc");
```

Sorting

There are 2 ways to implement server-side sorting:

- by specifying additional parameters in URL
- by using sorting type 'connector' while component configuration

URL manipulation

APPLICABLE TO: Grid, TreeGrid, Tree, Combo, Scheduler, DataView, Chart, Form, DataStore, DHTMXL Touch components

You can control how data will be sorted inside column by specifying additional parameters in URL.
Check article Extending functionality for full url parameters description.

```
//ORDER by field_2 ASC
grid.load("some.php?connector=true&dhx_sort[2]=asc");
//ORDER by field_2 ASC, field_3 DESC
grid.load("some.php?connector=true&dhx_sort[2]=asc&dhx_sort
```

sorting type 'connector'

APPLICABLE TO: Grid, TreeGrid

To sort grid/treegrid content with connectors you need to use '*connector*' as sorting type while grid initialization.

```
grid.setColSorting("connector,str,na");
```

In the code snippet above, the first column will be sorted on server-side with connectors, the second as string on client-side, the third column won't be sortable.

By assigning to sorting type 'connector' you just 'say' that sorting will be implemented on server-side.
To define the way, 'behaviour' of sorting you should use beforeSort event. Event doesn't allow to write custom sorting logic, but you can affect SORT BY clause of generated SQL request.

default sorting by one field

```
function custom_sort($sorted_by){
          //SORT BY some_field ASC
          if (!sizeof($sorted_by->rules))
               $sorted_by->add("some_field","ASC");
}
$conn->event->attach("beforeSort","custom_sort");
```

default sorting by multiple fields

```
function custom_sort($sorted_by){
          //SORT BY some_field ASC, some_other ASC
```

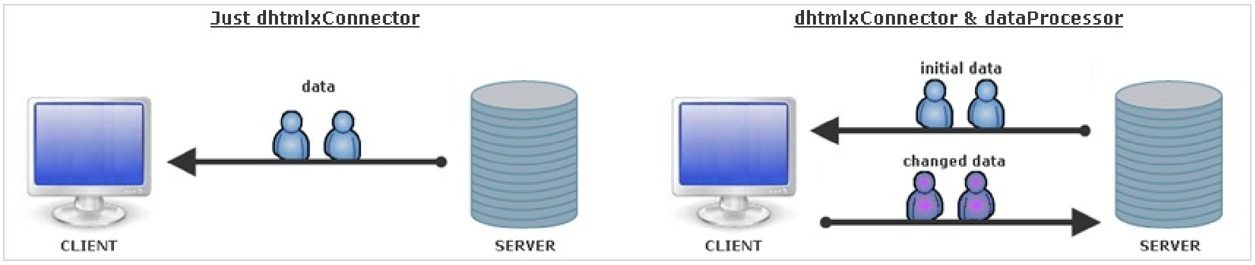
```
            if (!sizeof($sorted_by->rules)){
                  $sorted_by->add("some_field","ASC");
                  $sorted_by->add("some_other","ASC");
            }
}

$conn->event->attach("beforeSort","custom_sort");
```

custom sorting rule

```
function custom_sort($sorted_by){
            // SORT BY LENGTH(some_field)
            $sorted_by->rules[0]["name"]="LENGTH(some_field)"
}

$conn->event->attach("beforeSort","custom_sort");
```

Client-side requirement - dataProcessor

Connector itself lets just pass data from server- to client-side.
When you need to pass data back (e.g. you've updated record and want to save updates in database) you should use dataProcessor additionally.



Shortly, data exchange with dataProcessor can be characterized as follows:

1. After you make some changes client sends query with several parameters to server. The parameter we are most interested in is **!nativeeditor_status** (the parameter name can be changed. Go here for details). It defines the status of operation and can have one of the following values:
   - **updated**
   - **inserted**
   - **deleted**
2. Server tries to implement the required query. If the operation was performed successfully, server returns the same status it got. Otherwise - status “**error**”.
3. While exchanging, data is undergone default processing both on server- and client-side, i.e. data will be updated/inserted/deleted automatically (just initialize dataProcessor and dhtmlxConnector), no additional code needs to be added. You can affect on this default processing by means of events (see details below).

Files to include on client-side

If you use the '***dhtmlxSuite***' package (*dhtmlx.js, dhtmlx.css* code files) - you needn't to add any additional files.
But if you use the components standalone you should add one additional file - **dataprocessor.js**.

Beware, **dataprocessor.js** should be included BEFORE **connector.js**.

Initialization and linking to connector

To initialize dataProcessor you should write 2 commands:

```
var dp = new dataProcessor(url)
dp.init(mygrid)
```

To link dataProcessor with connector you should specify connector file as a parameter of the constructor:

```
dp = new dataProcessor("myconnector.php");
dp.init(mygrid);
```

Client-side logging

Dataprocessor has its own client-side logger, which can be enabled by including one additional js file - *dhtmlxdataprocessor_debug.js*

Validation

client-side validation

Dataprocessor allows to validate data before sending to server-side (see details here).

server-side validation

Details of server-side validation also see in the related chapter of this documentation.

Changing default processing

As it was mentioned before, to affect on default data processing either on server- or client-side you should use events (they can be either dataProcessor or dhtmlxConnector events).

changing default data processing on server-side

There are the following ways to affect on server-side processing:

1. To use handler functions of **dhtmlxConnector** events:
   - beforeUpdate Event
   - beforeInsert Event
   - beforeDelete Event
   - beforeProcessing Event
2. To create data model (OOP style; available from version 1.5)

```
class EventModel{
        function get($request){
                return array_of_data();
        }
        function update($action){
                //call $action->success(); or $action
        }
        function insert($action){
                //call $action->success(); or $action
        }
        function delete($action){
                //call $action->success(); or $action
        }
}

$connector = new GridConnector($this->db);
$connector->configure("events", "event_id", "start_da
$connector->useModel(new EventModel());
$connector->render();
```

Any of methods in the model can be skipped, in such case the connector will try to generate its own version of logic.

Methods **update()**, **insert()**, **delete()** take the DataAction object as a parameter (the same as the beforeProcessing event do). Method **get()** takes the request object and must return array of hash (the same as the render_array method do).

You can also combine 2 variants and use data model and events at the same time. The handlers will be invoked in the following order:

- 'before' events (beforeProcessing,…);
- data model;
- 'after' events (afterUpdate,…).

changing default data processing on client-side

Changing default data processing on client-side can be done in one of the following ways:

1. On server-side through handler functions of **dhtmlxConnector** events:
    - afterUpdate Event
    - afterInsert Event
    - afterDelete Event
    - afterProcessing Event
2. On client-side through handler functions of **dataProcessor** events:
    - onBeforeUpdate
    - onAfterUpdate

3. On client-side through **dataProcessor**'s method defineAction(). The method allows to define handler function of the specified status.

```
dp.defineAction("update",function(sid,response){
...
return true;// return false to cancel default data pr
})
```

- meanwhile, you can change status of server-side response through **dhtmlxConnector**'s method set_status() and assign the appropriate processing through defineAction().
  Status can be changed in 2 ways:
  I. by setting other predefined status ('updated', 'inserted', 'deleted'). In this case you just change default processing, write some additions to it.
  II. by setting custom status on server-side. In this case you cancel default processing at all and should define all actions on client-side by yourself.

```
$data->set_status("my_status");
```

For more details see chapter 'Custom status'

Renaming the query status parameter

Starting from version 1.5 you get the possibility to rename the query status parameter named by default as **!nativeeditor_status**.

When can you use it?
The most obvious example - the case when you want to use dhtmlxConnector+dataProcessor with the CodeIgniter framework.
Here you face the following problem - dataProcessor protocol is incompatible with the framework cause the framework blocks the default name of the parameter. The problem can be fixed by renaming the parameter on both server and client sides:

Client side:

```
var dp = new dataProcessor(./data);
dp.action_param = "dhx_editor_status";
```

Server side:

```
DataProcessor::$action_param ="dhx_editor_status";
```

Making Queries

simple queries

Applicable: Grid, TreeGrid, Tree, Combo, Scheduler, DataView, Chart, Form, DataStore, DHTMLX Touch Components

By default, connector generates all INSERT/UPDATE/DELETE queries automatically, based on configuration.
For more details of this topic, see the 'Base Concepts' chapter.

In case of dnd, connector will process an action as a sequence of 'insert' and 'delete' operations.

complex queries

When you need to define your own logic you should use one of two ways:

- defining custom SQL code for operation
- using server-side events to customize operations

custom queries for an action

Applicable to: Grid, TreeGrid, Tree, Combo, Scheduler, DataView, Chart, Form, DataStore, DHTXML Touch Components

You can define your own SQL code for specific action (INSERT, UPDATE or DELETE) as follows:

```
$grid->sql->attach("Update","Update tableA set name='{name}
//...
$grid->render_complex_sql(" .. ","id","price,name");
```

Parameters:

- action name. Possible values are: 'Update', 'Insert', 'Delete'
- SQL statement. It can use fields(or their aliases) which were mentioned in *render_sql* or *render_table* method while loading data.

using server-side events

Applicable to: Grid, TreeGrid, Tree, Combo, Scheduler, DataView, Chart, Form, DataStore, DHTMLX Touch Components

To customize operations you can use the following server-side events:

- beforeUpdate
- beforeInsert
- beforeDelete
- beforeProcessing

```
//data preprocessing before update
function my_update($data){
       $price = $data->get_value("price");
       $price = intval($price);
       $data->set_value("price","10");
}
$conn->event->attach("beforeUpdate","my_update")
```

```
//including additional field to request
function my_update($data){
       $data->add_field("userId",1); //will be included in 
}
$conn->event->attach("beforeUpdate","my_update")
```

```
//fully custom code
function my_update($data){
       global $conn;
       $price=$data->get_value("price");
       $id=$data->get_value("id");
       $conn->sql->query("UPDATE some_table SET price='{$pr
       $data->success(); //success() marks operation as fin
}
$conn->event->attach("beforeUpdate","my_update")
```

transactions

APPLICABLE TO: Grid

Grid allows to use transactions , for data INSERT/UPDATE/DELETE operations (be sure that used DB engine has support for transactions). They can be enabled in 2 modes:

1. **global**

```
$conn->sql->set_transaction_mode("global");
```

    - Component uses single transaction for all records inside single request.
    - If any operation fails - all record operations will be rolled back, all updated rows will be returned with error status
    - If custom validation was assigned and block any record - all records inside request will be blocked

2. **record**

```
$conn->sql->set_transaction_mode("record");
```

    - Component uses separate transactions for each record in request
    - If any operation fails, it will not affect other operations

INSERT operation in Oracle DB

Applicable to: Grid, TreeGrid, Tree, Combo, Scheduler, DataView, Chart, Form, DataStore, DHTMLX Touch Components

Oracle doesn't have auto ID generation functionality, so you need to provide some custom ID for each insert operations.

There are 2 ways to achive this:

1. **custom ID generation** - id generated by PHP code

```
function pre_insert($data){
        $new_id = time()%10000; //any other ID genera
        $data->add_field("EMPLOYEE_ID",$new_id);
}
$grid->event->attach("beforeInsert","pre_insert");
$grid->render_table("EMPLOYEES","EMPLOYEE_ID","FIRST_
```

2. **using sequence** for ID generation

```
$grid->sql->sequence("EMPLOYEES_INC.nextVal"); //sequ
$grid->render_table("EMPLOYEES","EMPLOYEE_ID","FIRST_
```

Multi-user sync

APPLICABLE TO: Grid, TreeGrid, Tree, Combo, Scheduler, DataView, Chart, Form

Starting from version 1.0, dhtmlxConnector can work in multi-user mode that allows users to see changes of each other in real time.

To enable mode you need to write additions both to client- and server-sides:

client-side

1. init component with dataprocessor (if it hasn't initialized yet)

```
dp = new dataProcessor("myconnector.php");
dp.init(mygrid);
```

2. call **setAutoUpdate()** method where input parameter is delay between update calls

```
dp.setAutoUpdate(2000);
```

server-side

1. call **enable_live_update()** method

```
        require("../../codebase/grid_connector.php");
        $grid = new GridConnector($res);
        $grid->enable_live_update('actions_table');
```

actions_table - table in database, which will be used for sync. data storage.
It needs to have the next structure:

```
CREATE TABLE `actions_table` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `dataId` int(11) NOT NULL,
  `type` varchar(255) NOT NULL,
  `user` varchar(255) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM;
```

Security

dhtmlxConnector is open to external access of another programs and any external user is allowed to change data in database.
Thus, adding some kind of session-based authentication is strongly recommended.

Basic security

Built-in security manager allows you to limit connector for certain operations.

```
$gridConn->access->deny("read"); //blocks Select action
$gridConn->access->deny("insert"); //blocks Insert action
$gridConn->access->deny("update"); //blocks Update action
$gridConn->access->deny("delete"); //blocks Delete action
```

By default, connector allows all operations.

Protection from cross-site scripting (XSS)

Starting from version 1.5, dhtmlxConnector allows you to protect an app from XSS attacks.

To avoid XSS attacks, the library checks all data inputted by users and according to the set security level doesn't allow html or javascript code to be inserted.

3 security levels are available:

- **DHX_SECURITY_SAFETEXT** (default) - all html data is removed from the input field;
- **DHX_SECURITY_SAFEHTML** - allows html data, but removes possible script tags and handlers;
- **DHX_SECURITY_TRUSTED** - input fields are not filtered at all (similar to previous version of connectors).

To set the necessary security level, use the next code:

```
ConnectorSecurity::$xss = DHX_SECURITY_SAFETEXT;
//ConnectorSecurity::$xss = DHX_SECURITY_SAFEHTML;
//ConnectorSecurity::$xss = DHX_SECURITY_TRUSTED;
```

If you want to enable the same behavior as the previous versions of connectors have, set the **DHX_SECURITY_TRUSTED** mode.

Preventing CSRF and XSRF attacks

The functionality requires 3.5 or later version of the DHTMLX Suite package

The protection is available starting from version 1.5 and can be activated by the following code line:

```
ConnectorSecurity::$security_key = true;
```

After calling such a command, connectors start to include additional security keys to all data loading operations and process data updating calls only if they contain the same keys. As a result of this processing, it's impossible to trigger a data updating operation from some third-party site, even if an attacker has access to a valid user session.

The technique is based on php sessions and assumes that any php session will be preserved between separate script calls (default php behavior).

Please be sure that you understand what CSRF attack is, cause the stated technique won't prevent access to the connector from external urls, it will only prevent execution actions through some one elses session.

Validation

Before saving on server you can validate data and assign handler function for any gotten response.

Shortly, validation contains 2 key points:

1. value checking
2. processing in case of validation error

common

To implement server-side validation of incoming data you can use the following events:

- beforeUpdate
- beforeInsert
- beforeDelete
- beforeProcessing

beforeProcessing event occurs for all types of operations while other events occurs only for related operations, i.e. you can set different validation rules for different operations or for all of them at once.

Event will receive DataAction object as the parameter. This object can be used to retrieve related data and allow|deny operation (please note, it contains only data which was received from client-side, not the all data related to the record).

value checking

To check value of a field you should use the method get_value()

```
function validate($data){
              if ($data->get_value("some_field")=="")
                   ...
}
$conn->event->attach("beforeProcessing","validate");
```

processing in case of validation error

In case of error you can go one of two ways:

1. Use predefined methods for error processing on client-side, i.e. set:

   - **$data→invalid()** (invalid record will be highlighted in bold font)
   - **$data→error()** (invalid record will be highlighted in red bold font)
     The difference between methods in question consists just in way of highlighting.

   ```
   function validate($data){
                   if ($data->get_value("some")=="")
                         $data->invalid();
   }
   $conn->event->attach("beforeProcessing","validate'
   ```

2. Assign your own processing on client-side through dataProcessor's method defineAction() (see details here)

   ```
   dp.defineAction("invalid",function(sid,response){
   var message = response.getAttribute("message");
   alert(message);
   return true;// return false to cancel default data pr
   })
   ```

custom status

You can change returned status and set you own by means of the method set_status(). In this case any default processing (doesn't matter defineAction() will return true or false) will be cancelled and you will specify data processing on client-side wholly.

server-side:

```
function validate($data){
             if ($data->get_value("some")=="")
                  $data->set_status("my_status")
}
$conn->event->attach("beforeProcessing","validate");
```

client-side:

```
dp.defineAction("my_status",function(sid,response){
...
})
```

adding custom information to response

You can send some custom information to client-side through the following methods:

- set_response_text
- set_response_xml
- set_response_attribute

Saving data changes made in form

You can use the connectors for simple form saving. The connectors will recognize and automatically execute CRUD operations for the following patterns:

- Get data for some record in DB, by id

  ```
  GET: connector.php?action=get&id={some}
  ```

  Returns: data in 'json'(if you use JSONDataConnector) or 'xml'(if you use DataConnector) format.

- Delete data for some record in DB, by id

  ```
  GET: connector.php?action=delete
  POST: id={some}
  ```

  Returns: “true” as string, when operation is executed correctly.

- Update data for some record in DB, by id

  ```
  GET: connector.php?action=delete
  POST: id={some}&property_name={value}
  ```

  Returns: “true” as string, when operation is executed correctly.

- Insert data in DB

  ```
  GET: connector.php?action=insert
  POST: property_name={value}
  ```

Returns: “true\n{new id}” as string, when operation is executed correctly.

Basically, for DHTMLX Touch components to save data you can use the following code:

```
dhx.ajax().post("connector.php?action=insert", $$('form').g
```

Data export

APPLICABLE TO: grid

External *grid-to-pdf* and *grid-to-excel* services allow to convert grid's data to PDF or Excel format directly on server-side: from any table without actually rendering data in the grid. Shortly, you should perform the following steps:

- prepare data.
- activate conversion service.
- start exporting.

data preparation

- data to export is defined by render_table() or render_sql() method.
- to define grid structure (header, footer etc.) on server-side, see chapter ''.

activation conversion service

- **pdf**

```
$convert = new ConvertService("http://dhtmlx.com/docs/pr
```

- **excel**

```
$convert = new ConvertService("http://dhtmlx.com/docs/pr
```

start exporting

- **pdf**

  ```
  $convert->pdf("some.pdf",false);
  ```

- **excel**

  ```
  $convert->excel("some.xls",false);
  ```

Parameters:

- (optional) name of result file. The default value - data.pdf (data.xls)
- (optional) the second parameter specifies how file will be exported:
  - *true* - as inline content (if browser has related plugin - document will be opened automatically)
  - *false* - as individual file.
    The default value - *false*.

Service automatically start to export data defined through GridConnector.

samples

- **pdf**

  ```
  require("../../../codebase/grid_connector.php");
  require("../../../codebase/convert.php");

  //url to data conversion service
  $convert = new ConvertService("http://dhtmlx.com/docs/pr
  $convert->pdf(); //equal to calling pdf("data.pdf", fals

  $grid = new GridConnector($res);
  $grid->set_config(new GridConfiguration());     //mandat
  $grid->render_table("grid50");          //table name and
  ```

- **Excel**

```
require_once("../../config.php");
$res=mysql_connect($mysql_server,$mysql_user,$mysql_pass
mysql_select_db($mysql_db);

require("../../../codebase/grid_connector.php");
require("../../../codebase/convert.php");

//url to data conversion service
$convert = new ConvertService("http://dhtmlx.com/docs/pr
$convert->excel(); //equal to calling excel("data.xls",

$grid = new GridConnector($res);
$grid->set_config(new GridConfiguration()); //mandatory
$grid->render_table("grid50");        //table name and
```

useful tips

- In case of dynamic Smart Rendering you can't use export from client-side and need to define header's data on server-side. See details here.
    - in order to the **render_sql** method works correctly, you should use full header initialization (not empty GridConfiguration() call). Otherwise, the component will try to fetch grid's structure directly from DB that is unallowable for the **render_sql** method.

Elementary DB operations with Connector

dhtmlxConnector allows to execute some actions against DB.

SQL queries

DBWrapper object can be accessed as:

```
$connector->sql
```

Then, it can be used in queries in the following way:

```
$connector->sql->query("DELETE FROM some_table WHERE ID=1")
//or
$res = $connector->sql->query("SELECT * FROM some_table WHE
$data =  $connector->sql->get_next($res);
//or
$connector->sql->query("INSERT INTO some_table(type) VALUES
$id = $connector->sql->get_new_id();
```

INSERT query

```
$id = $connector->insert(array(
                "type" => "simple",
                 ...
));
```

Parameters:

- hash of values

UPDATE query

```
$connector->update(array(
                "type_id" => '1'
                "type" => 'simple'
));
```

Parameters:

- hash of values. For successful result it must contain *identity field*.

DELETE query

```
$connector->delete($id);
```

Parameters:

- ID of the record that should be deleted

Creating a new connector

You can create an extra connector object on the fly and use it for DB operations.

```
$temp = new Connector($db_connection);
$temp->configure("some_table");

$temp->insert(array(
        "some1" => "value 1",
        "some2" => "value 2"
));
$temp->delete("2");
```

Errors handling and logging

While developing we strongly recommend to use logs for errors detection.

server-side logging

DHTMLX permits logging to be controlled in two ways:

- Basic
- Detailed

basic

If any error occurs during data processing, client-side data processor object will receive “error” action containing short info about the problem (full info will be written to the log).

```
$conn->enable_log("path");
```

- *path* - absolute or relative path to text file where log will be written.

detailed

When critical error occurs all log records for current session (full error info) will be sent to client and shown in browser (useful for debugging, not recommended for production):

```
$conn->enable_log("path",true);
```

- *path* - absolute or relative path to text file where log will be written.
- The second parameter enables detailed mode.

adding custom records to the log

During development you may need to write some custom data to the log

(can be useful for custom server-side events).
To add custom text or data to log file:

- Activate logging (basic or detailed).
- Call *log()* function with your data inside.

```
$conn->enable_log("temp.log");
LogMaster::log("any text here");
```

client-side logging

Dataprocessor has its own client-side logger, which can be enabled by including one additional js file - *dhtmlxdataprocessor_debug.js*

Event handling

common use

DHTMLX Connector uses events to manage Insert/Update/Delete operations.

The code below shows how to attach event:

```
$conn->event->attach(event_name,handlerFunc);
```

Parameters:

- *event_name* - name of the event.
- *handlerFunc* - handler function.

Handler functions get one input parameter and this parameter depends on attaching event.
In the following table you'll see a full list of events and the appropriate input parameters of handler functions.

Event	Input parameter of handler function
beforeSort	SortInterface Object
beforeFilter	FilterInterface Object
beforeRender	DataItem Object
beforeProcessing afterProcessing beforeInsert beforeUpdate beforeDelete afterInsert afterUpdate	DataAction Object

afterDelete	
beforeOutput	none

data behavior models

Starting from version 1.5, the library 'extends' OOP support and allows creating a separate class for events processing. You get the possibility to work with one class that will control all processing instead of working with separate handler functions.

```
class SomeBehavior{
        public function beforeRender($data){
                //some logic
        }
        ... any other handlers ...
}
$conn->event->attach(new SomeBehavior());
```

The handlers of the model class take the same parameters as the ordinary events do.

Extending functionality

protocol details

data fetching

POST

Command parameters:

- none

GET

Command parameters:

- ***connector=true*** - flag of connector based request. The flag is set automatically, once you include the *connector.js* file on the page.
- ***dhx_colls=field1,field2…fieldN*** - optional, can contain a list of fields for which collections will be requested. dhtmlxGrid use such parameters to request data for combo columns and select filters (such requests are executed just once , for initial data loading)

```
some.php?connector=true&dhx_colls=2,3
```

- ***dhx_sort[field]=directon*** - instruct server-side connector to sort dataset by defined field

```
some.php?connector=true&dhx_sort[2]=asc
```

- ***dhx_filter[field]=mask*** - instruct server-side connector to filter dataset by defined field

```
//filter by %test%
```

```
some.php?connector=true&dhx_filter[2]=test
```

grid specific

Requsting part of data (Dynamic Smart Rendering or Dynamic Paging)

GET

- ***posStart*** - position from which data is requested
- ***count*** - count of requested rows

combo specific

Requsting part of data

GET

- ***pos*** - position from which data is requested

Filter by label field

GET

- ***mask*** - filtering mask for label field

tree/treeGrid specific

Requesting branch of tree

GET

- ***id*** - parent id for requested branch

data updating (using dataProcessor)

GET

- ***editing=true*** - mark of dataprocessor-based call

POST

- ***ids*** - list of updated records
- ***[id]_[property]*** - for each field inside updated record related field in POST is generated
- ***[id]_!nativeeditor_status*** - action type

Back to top

porting connectors to another platforms

when should you use it?

The existing version of connectors supports a limited set of server platforms. In case database/framework/scripting language you'd like to use is not supported, you can port existing solution on your platform.

when shouldn't you use it?

Connectors are just wrappers around existing grid functionality, that's why if you need to use this solution once, you would better use grid API directly, instead of creating your own server connector.

implementation levels

Connector supports many operations implementing data processing. There is an opportunity to provide supporting basic operations and ignore higher-level ones, in case they are not used in your project.

basic support

Data assignment is basic connector operation which proceeds in the following way: connector connects to database, selects data and outputs this in *stdout* using XML format of the current component.

Key points:

1. XML-data input must occur only after sending the appropriate http-header
2. XML must start with XML declaration containing the appropriate data coding

3. None other content must be sent to *stdout* (neither before nor after sending data by connector)

```
header("Content-type:text/xml");
print("<?xml version='1.0' encoding='utf-8' ?>");
print(xml_formatted_data);
```

- incoming parameters - there are no any;
- restriction for output data - there is no restriction.

XML Format used by connector

Basic implementation allows to use resulting script as input parameter for load()/loadXML() methods.

Technically, there is an opportunity to use JSON or any other supported by component format, but you should bear in mind that complex scenarios are XML-oriented and you would better use XML

filtration and sorting

At this stage, the number of supported operations is considerably extended: you are allowed to use #connector_text_filter, sorting type 'connector' and filter/sort data through URL manipulations.

Key points:

- An obvious advantage of this functionality can be estimated only for operating with grid, for other components such functionality doesn't seem to be necessary (it can be used only to filter/sort data through URL manipulations).
- The level 'uses' code from the previous one. Beware, that now, before using XML, data is sorted/filtered according to incoming data.

GET

Command parameters:

- ***dhx_filter*** - hash of filtering rules
  - filtration pattern - any entry(like %x%)
  - in case you have a few parameters, you should link them by AND logic
  - parameter is available only for fields with active filter
  - empty filter value means that filter wasn't set and it must be ignored

  ```
  // where field1 like %some% AND field2 like %other%
  dhx_filter[1]=some&dhx_filter[2]=other
  ```

- ***dhx_sort*** - hash of sorting rules
  - possible values for *asc* and *dsc* parameters
  - in case you have a few parameters, you should link them by AND logic
  - order of filtering implementation - not defined, at the moment no one component is able to create multi-field sorting

  ```
  // order by field1 ASC, field2 DESC
  dhx_sort[1]=asc&dhx_filter[2]=dsc
  ```

  Hashes of rules in question use name of fields (where filtration is enabled) or columns' indices (in case of grid).

Restriction for output data: order and structure are defined through filtering/sorting parameters.

combo specific:
dhtmlxCombo has an additional filtering GET parameter:

- ***mask*** - sets filtration in text label columns using “like x%” rule.

```
some.php?mask=abc
```

dynamic loading

After data output implementation goes dynamical loading. Dynamical loading has some particular features while working with hierarchical (tree/treegrid) and simple (grid/combo) components.

For hierarchical components, loading of data branch occurs at once.

```
some.php?id=123
```

Command parameters:

- ***id*** - 'parent id' which defines the appropriate branch (beware, all filtration/sorting rules set while the previous stage are applied to component).

For other components, data will be outputted according to incoming parameters.

```
//grid
some.php?posStart=20&count=50
//combo
some.php?pos=50
```

Command parameters:

grid:

- ***posStart*** - index of initial string
- ***count*** - number of strings to output

combo:

- ***pos*** - index of initial string

- number of strings is defined by server settings

While working with grid, initial request (grid doesn't know yet how many strings are expected) doesn't contain any additional parameters ('posStart' and 'count' are not defined)

Back to top

XML format used by connector

Grid

```
    <rows>
            <row id="some1"><cell>value 1,1</cell><cell>valu
            <row id="some2"><cell>value 2,1</cell><cell>valu
    </rows>
```

- **row** - the tag represents row in the grid
- **cell** - the tag represents cell in the grid

row@id - the mandatory attribute, must be unique for each row; row and cell tags may contain additional attributes which will be processed in default way.

In *dynamic loading* mode

```
    <rows total_count="50000" pos="1230">
            <row id="some1230"><cell>value 1,1</cell><cell>va
            <row id="some1231"><cell>value 2,1</cell><cell>va
    </rows>
```

- **rows@total_count** - count of rows in grid, needs to be sent only for the first data request (when posStart parameter is not defined)
- **rows@pos** - equal to posStart parameters, defines where grid data will be added

TreeGrid

```
<rows>
        <row id="some1">
           <cell image="some.gif">value 1,1</cell><cell>
           <row id="some11"><cell>child value 1</cell><c
        </row>
        <row id="some2"><cell>value 2,1</cell><cell>valu
</rows>
```

- **row** - the tag represents row in the grid, one row tag can contain other row tags inside (parent - child relation in the hierarchy)
- **cell** - the tag represents cell in the grid
- **row@id** - the mandatory attribute, must be unique for each row
- **cell@image** - can define image for tree cell in grid (optional)

Dyn. loading

```
<rows parent="id">
        <row id="some1" xmlkids="1">
           <cell image="some.gif">value 1,1</cell><cell>
        </row>
        <row id="some2"><cell>value 2,1</cell><cell>valu
</rows>
```

- **row@xmlkids** - defines that row can contain child nodes
- **rows@parent** - the name of parent ID (equal to incoming “id” parameter) , ID of virtual root - 0

Tree

```
<tree>
     <item id="some" text="name">
              <item id="child1" text="name 1"/>
              <item id="child2" text="name 2"/>
     </item>
</tree>
```

- **item tag** - item in a tree, can nest other item tags to represent parent-child relation
- **item@id** - ID, unique for each item
- **item@text** - label

In case of dyn. loading

```
<tree id="id">
     <item id="some" text="name" childs="1">
     </item>
</tree>
```

- **item@childs** - defines that row can contain child nodes
- **tree@id** - the name of parent ID (equal to incoming “id” parameter) , ID of virtual root - 0

Combo

```
    <data>
        <option value="some">Some label</option>
        <option value="other">Other label</option>
    </data>
```

- **option** - represents option in combo

In case of dyn. loading:

```
    <data add="true">
        <option value="some">Some label</option>
        <option value="other">Other label</option>
    </data>
```

If “pos” parameter was provided, response must contain data@add attribute , to mark that it is not a new set of values, but addition to existing one.

Using dhtmlxConnector with CodeIgniter, YII, CakePHP frameworks

Starting from version 1.5 dhtmlxConnector can be used with popular PHP frameworks. You can built web applications with your favorite frameworks and still use dhtmlxConnector.

In this article we will give you the brief tips referring to such use-case of dhtmlxConnector. The detail information you can get in the related tutorials:

1. Using dhtmlxConnector with the CodeIgniter framework;
2. Using dhtmlxConnector with the YII framework;
3. Using dhtmlxConnector with the CakePHP framework.

As an example, let's take creating an app that presents grid loaded data from db. The app is built with one of the frameworks and use dhtmlxConnector to load data in.

Tips common for all frameworks

Model: Doesn't have any specificity and created as usual.

View:

- View contains a standard JavaScript code of the component and defines the client side logic;
- To load data use standard 'loading' methods of the components but as the **URL** parameter specify the relative path to an action that answers for loading logic. If you use dataProcessor, specify in its constructor this relative path to the action as well;

```
mygrid = new dhtmlXGridObject('grid_here');
...
mygrid.init();
mygrid.loadXML("./data"); //refers to the 'data' action

var dp = new dataProcessor("./data"); //refers to the 'data
dp.init(mygrid);
```

Controller:

- Controller defines the server side logic and contains at least 2 actions: the one loads the view, the other loads the data;
- To handle an event, add a function of the corresponding event name (e.g. to handle beforeRender event, add *public function beforeRender($action){…}*) and define the handling logic as usual. The functions will take the same parameters as the ordinary events do;
- SQL query to database is configured with the help of **configure()** method that takes as parameters:
  - **$sql** - any sql code that will be used as the base for data selection (for the **Yii** framework the hard coded value - **'-'**);
  - **$id** - the name of the id field;
  - **$text** - a comma separated list of rendered data fields;

        - **$extra** - (*optional*) a comma separated list of extra fields;
        - **relation_id** - (*optional*) used for building hierarchy in case of Tree and TreeGrid.
    - GridConnector (or any other connector) takes as parameters:
        - the DB connection variable that refers to the model used in the app)
        - the hard coded value specified the framework (“phpCI”, “PHPYii” or “PHPCake”).

```
//sample code for the YII framework
<?php
require_once(dirname(__FILE__)."/../../../dhtmlx/connector/
require_once(dirname(__FILE__)."/../../../dhtmlx/connector/

class EventController extends Controller
{
        public function actionIndex(){ $this->render('index
        public function actionGrid() { $this->render('grid'

        public function actionGrid_data()
        {
                $grid = new GridConnector(Events::model(),
                $grid->configure("-", "event_id", "start_da
                $grid->render();
        }

        public function beforeProcessing($action){
                //validation before saving
                if ($action->get_value("event_name") == "")
                        $action->invalid();// if data isn't
                        $action->set_response_attribute("de
                }
        }
}
```

Framework-specific tips

CodeIgniter

View

- While working with CodeIgniter you should rename the query status parameter **!nativeeditor_status** cause the default name of the parameter is blocked by the framework. Renaming must be repeated on the server side.

```
var dp = new dataProcessor("./data"); //refers to the 'data
dp.action_param = "dhx_editor_status";
dp.init(mygrid);
```

Controller

- Required connector file - **db_phpci.php** (and of course, you should also include connector files related to components used in the app, e.g. if you use dhtmlxGrid - you also include **grid_connector.php**);
- The query status parameter **!nativeeditor_status** must be renamed on the server side as well.

```
require_once("./dhtmlx/connector/grid_connector.php");
require_once("./dhtmlx/connector/db_phpci.php");
DataProcessor::$action_param ="dhx_editor_status";

class Grid extends CI_Controller {
        public function index()
        {
                $this->load->view('grid'); //grid's view
        }
        public function data()
        {
                $this->load->database();

                $connector = new GridConnector($this->db, "
                $connector->configure("events", "event_id",
                $connector->render();
        }
}
```

YII

Controller

- Required connector file - **db_phpyii.php**.

```
<?php
    require_once(dirname(__FILE__)."/../../../dhtmlx/connec
    require_once(dirname(__FILE__)."/../../../dhtmlx/connec

       class EventController extends Controller {
              public function actionIndex() { $this->render
              public function actionGrid()  { $this->render
              public function actionGrid_data()
              {
                   $grid = new GridConnector(Events::model(
                   $grid->configure("-", "event_id", "start_
                   $grid->render();
              }
       }
?>
```

CakePHP

Controller

- Required connector file - **db_phpyii.php**.

```
<?php
require_once("../Vendor/connector/grid_connector.php");
require_once("../Vendor/connector/db_phpcake.php");

        class EventController extends AppController {
                public function grid(){}
                public function index(){}
                public function grid_data()
                {
                    $this->autoRender = false;

                    $connector = new GridConnector($this->Eve
                    $connector->configure("events", "event_id
                    $connector->render();
                }
        }
?>
```

How can I:

General:

- … assign aliases to DB fields?
- … attach event?
- … check value of an item?
- … create custom database error message?
- … customize content of a cell?
- … deny access to a certain operation?
- … filter data on server-side?
- … handle errors and log them?
- … set value of an item?
- … sort data on server-side?
- … validate data on client-side?
- … validate data on server-side?

Loading data:

- … enable dynamic loading?
- … format/change data before loading?
- … load data from database table?
- … load data from Excel file?
- … load data from File System?
- … send additional information to client-side?

Updating data:

- … alter the default styles set for responses?
- … avoid updating certain field(s) on server-side?

Components-specific:

- ***Combo:***
  - … populate combo with dat
- ***Form:***
  - … populate 'select' item wit db?
  - … populate 'combo' item w db?
  - … load data from db?
  - … save changes made in a
- ***Grid:***
  - … define grid configuration side?
  - … export grid's data from s Excel file?
  - … export grid's data from s file?
  - … load data from a table th contain identity field?
  - … populate select/combo c data?
  - … set a custom style for a cell?
- ***Tree and TreeGrid:***
  - … load data from multiple t
  - … load data from table that contain relation field?
  - … mark an item as a leaf o
- ***DataStore:***
  - … use connector functional dhtmlXDataStore objects?

- … change default data processing while updating?
- … set custom status of operation?
- … terminate any further processing?
- … change values before saving?
- … implement transactions?
- … link dataProcessor with connector?
- … realize multi-user sync?
- … save data changes made in form?
- … update data on server-side?

General how-tos

How can I assign aliases to DB columns?

Using render_sql() or render_table() you can assign aliases to required tables or columns. To do this, in sql statement you should use word 'as' (as in any usual sql statement), in other parameters - parentheses.

```
$grid->render_table("tableA","id","name,price(product_price
//or
$grid->render_sql("Select *,tableA.id as aid from tableA, t
```

How can I attach event?

To attach event you should use **event→attach()**. For more details of this topic, see 'Event handling' guide.

```
$conn->event->attach(event_name,handlerFunc);
```

How can I check value of an item?

Using get_value() method you can check value of any item.

```
$grid->get_value($name);
```

How can I create custom database error

message?

To add a custom error message you can use the event onDBError and write the desired message in the appropriate handler.

```
function doOnDBError($action, $exception){
       $action->set_response_text("Some details about error
}
$conn->event->attach("onDBError",doOnDBError);
```

How can I deny access to a certain operation?

By default, connector allows all operations. To deny some operation use ***deny(name_of_operation)*** method that can get one of the following:*read/update/insert/delete*. For more details see 'Security' guide.

```
$conn->access->deny("update");
```

How can I customize content of a cell?

You can use beforeRender event to define how content of a cell must be formatted.

```
function custom_data($row){
      $data = $row_get_value("some_column");
      $row->set_value("some_column", "$data <input type='te
}

$grid->event->attach("beforeRender","custom_data");
```

In the sample above, the grid will have custom content - html input 'text'.

How can I filter data on server-side?

You have 3 ways to filter data on server backend:

- by specifying additional parameters in URL (on client-side)

```
grid.load("some.php?connector=true&dhx_filter[1]=mask");
```

- by using in-header filter types while component configuration (on client-side)

```
mygrid.setHeader("Column A, Column B");
mygrid.attachHeader("#connector_text_filter,#connector_s
```

- by means of beforeRender event (on server-side)

```
function custom_filter($data){
   ...
}
$conn->event->attach("beforeRender","custom_filter");
```

See the guide 'Filtration' for more information.

How can I handle errors and log them?

You can enable logging using enable_log() method. For more details see 'Error handling and logging' guide.

```
$conn->enable_log("path to log file");// to show short info
//or
$conn->enable_log("path to log file", true);// to show full
```

How can I set value to an item?

Using set_value() method, you can set value to any item of component.

```
$dataItem->set_value($name,$value)
```

How can I sort data on server-side?

You have 2 ways to sort data on server backend:

- by specifying additional parameters in URL (on client-side)

  ```
  grid.load("some.php?connector=true&dhx_sort[2]=asc");
  ```

- by using sorting type 'connector' while component configuration (on client-side)

  ```
  grid.setColSorting("connector,str,na);
  ```

See the guide 'Sorting' for more information.

How can I validate data on client-side

dataProcessor lets you validate data on client-side. Use *setVerificator(index,method)* method to define the appropriate columns and validators. See details in the related chapter of dataProcessor' documentation.

```
dp.setVerificator(column_index,verification_func)
```

How can I validate data on server-side

To perform server-side validation you should use one of the dhtmlxConnector events stated below and specify the needed validation rules in the appropriate events' handlers functions:

- beforeUpdate
- beforeInsert
- beforeDelete
- beforeProcessing

```
function validate($data){
...
}
$conn->event->attach("beforeProcessing","validate");
```

For more details of server-side validation, see guide 'Validation'.

This topic does not exist yet

You've followed a link to a topic that doesn't exist yet. If permissions allow, you may create it by using the `Create this page` button.

Create this page

This topic does not exist yet

You've followed a link to a topic that doesn't exist yet. If permissions allow, you may create it by using the `Create this page` button.

Create this page

Data management how-tos (loading data)

How can I enable dynamic loading?

To enable dynamic loading you should:

- on client-side enable the related mode (e.g. smart rendering or paging for grid)

  ```
  grid.enableSmartRendering(mode,buffer);
  ```

- on server-side call method dynamic_loading()

  ```
  $grid->dynamic_loading([$rowsNum]);
  ```

See the guide 'Dynamic loading' for more information.

How can I format/change data before loading?

To set some specific formatting or change data before sending to client-side, you should use the beforeRender event handler. For more details of this topic, see 'Formatting/Changing Data before Loading'

```
function color_rows($row){
        if ($row->get_index()%2)
        $row->set_row_color("red");
}

$grid->event->attach("beforeRender","color_rows");
```

How can I load data from database table?

To load data from database table you should use one of two methods:

- render_table() (for loading from single table)

```
$grid->render_table("tableA","item_id","column1,column2"
```

- render_sql() (for loading from several tables)

```
$grid->render_sql("Select * from tableA, tableB  where
```

How can I load data from Excel file?

To load data from Excel file you should download phpExcel library from http://support.dhtmlx.com/x-files/connector/phpExcel.zip and include additional files:

- 'lib/PHPExcel.php' (phpExcel package)
- 'lib/PHPExcel/IOFactory.php' (phpExcel package)
- 'db_excel.php' (standard connector's package)

Then, call render_table()) method where as parameters you should specify cell range and Excel columns. Set the second parameter to 'id' for auto id generation.

```
//files from phpExcel package
require_once('lib/PHPExcel.php');
require_once('lib/PHPExcel/IOFactory.php');
//connectors
require("../../codebase/db_excel.php");
require("../../codebase/grid_connector.php");

$grid = new GridConnector("../some.xls", "ExcelDBDataWrappe
$grid->render_table("A18:F83", "id", "A,B,C,D,E,F");
```

For more details, see the guide chapter 'loading from Excel file' in 'Base concepts' guide.

How can I load data from File System?

To load data from File System you should include one additional file *db_filesystem.php* and call render_table() method where as parameters you should specify **folder** (that requires data listing), **field's id** (leave it empty or use *safe_name* as ID of file) and **lists of fields**.

```
require("./codebase/connector/db_filesystem.php");
require("./codebase/connector/grid_connector.php");

$grid = new GridConnector("", "FileSystem");
$grid->render_table("../","safe_name","filename,full_filena
```

For more details, see the guide chapter 'loading from File System' in 'Base concepts' guide.

How can I send additional information to client-side?

To send additional information to client-side that won't be outputed but you'll have access to it, use the fourth(optional) parameter of render_table() method. There you should specify columns that contain desired additional information.

```
$grid->render_table("some_table","id","name,price","color,c
```

For more inoformation, see the chapter 'Using extra fields' in 'Formatting/Changing Data before Loading' guide.

Data management how-tos (updating data)

How can I alter the default styles set for responses?

You can alter the default styles for each state, like:

```
dp.styles.error = ""; //avoids a special style for the 'err
//you should write it on the client-side (in your HTML file
//dp -  dataProcessor object
```

How can I avoid updating certain field(s)?

To remove some field from updating, you can use the following technique:

```
//write it before '$grid->render_table'
function filter_set($action){
   $action->remove_field("id"); //the named field won't be
}
$grid->event->attach("beforeProcessing", filter_set);
```

How can I change default data processing while updating?

To affect on default data processing either on server- or client-side you should use events either of dataProcessor or dhtmlxConnector. For more details, see the chapter 'changing default precessing' in 'Client-side requirement - dataProcessor'.

How can I set custom status of operation?

Using set_status() method you can set status (custom or another predefined) of operation. Using defineAction() method of dataProcessor you can assign the appropriate processing for this status.

```
//server-side
$data->set_status("my_status");
```

```
//client-side
dp.defineAction("my_status",function(sid,response){
...
return true;// return false to cancel default data processi
})
```

How can I terminate any further processing?

Calling success() method you terminate any further action processing, i.e. data updating will be stopped and considered as finished.

```
function my_update($data){
   ...
   $data->success(); // marks operation as finished
}
$conn->event->attach("beforeUpdate","my_update")
```

How can I change values before saving?

To customize values before saving you should use server-side events stated below:

- beforeUpdate
- beforeInsert
- beforeDelete
- beforeProcessing

```
function my_update($data){
   ...
}
$conn->event->attach("beforeUpdate","my_update")
```

For more information, see 'Using server-side events' in 'Making queries' guide.

How can I implement transactions?

Connector allows to use transactions for INSERT/UPDATE/DELETE operations (be sure that used DB engine has support for transactions). To activate transaction mode - use ***set_transaction_mode()*** method. For more details see the chapter 'Transactions' in 'Making queries' guide.

```
$conn->sql->set_transaction_mode("global"); //for all recor
//or
$conn->sql->set_transaction_mode("record");// for each reco
```

How can I link dataProcessor with connector?

To link dataProcessor with connector you should specify connector file as a parameter of dataProcessor constructor:

```
dp = new dataProcessor("myconnector.php");
dp.init("mygrid");
```

How can I realize multi-users sync?

To activate multi-user mode that will allow users to see changes of each other in real-time, you must add code both on client- and server-side. On server-side, to basic initialization connector code you should add:

```
$conn->enable_live_update('actions_table');  //actions_tabl
```

On client-side, you should initialize dataprocessor and call ***setAutoUpdate()***:

```
dp.setAutoUpdate(2000);//input parameter is delay between u
```

See Multi-user sync for more information.

How can I update data on server-side?

To update data on server-side, on client-side you should initialize dataProcessor and link dhtmlxConnector to it. Default updating will be done automatically. For more details, see the guide 'Client-side requirement - dataProcessor'.

How can I save data changes made in form (simple way)?

dhtmlxConnector automatically exec CRUD operations for next patterns:

```
//to get data of a DB record
   GET: connector.php?action=get&id={some}
//to delete data of a DB record
   GET: connector.php?action=delete
//to update data of a DB record
   GET: connector.php?action=delete
   POST: id={some}&property_name={value}
//to insert a new record to DB
   GET: connector.php?action=insert
   POST: property_name={value}
```

For more details, see the guide 'Saving data changes made in form'.

Component-specific how-tos (combo)

How can I populate combo with data from db?

To define options of combo you should use **ComboConnector** on the server-side and specify the connector file in the loadXML(url) method on client-side:

client-side:

```
var combo=new dhtmlXCombo("combo_zone2","alfa2",200);
combo.loadXML("connector.php");
```

server-side:

```
<?php
require_once("../codebase/connector/combo_connector.php");
$res=mysql_connect("localhost","root","");
mysql_select_db("mydb");

$data = new ComboConnector($res, "MySQL");
$data->render_table("categories","id","valueColumn, labelCo
?>
```

- Names of the fields can have aliases (**value** or **label**) to identify the appropriate attribute.

  ```
  $data->render_sql("SELECT *, CONCAT(FirstName, LastName)
  ```

  Note, in the filtering mode a combo filters data by the “label” field.

Component-specific how-tos (form)

How can I populate 'select' item with data from db?

To define options of the **select** item you should use **SelectOptionsConnector** on the server-side and specify the ***connector*** parameter for the appropriate item on client-side:

client-side:

```
var formData = [{type: "select", label: "Categories", conne
```

server-side:

```
<?php
require_once("../codebase/connector/options_connector.php")
$res=mysql_connect("localhost","root","");
mysql_select_db("myDB");

$data = new SelectOptionsConnector($res, "MySQL");
$data->render_table("categories","id","valueColumn, labelCo

?>
```

For more information of this topic, see 'SelectOptionsConnector' article.

How can I populate 'combo' item with data from db?

To define options of the **'combo'** item you should use **ComboConnector** on the server-side and specify the ***connector*** parameter for the appropriate item on client-side:

client-side:

```
var formData = [{type: "combo", name: "myCombo", label: "Se
```

server-side:

```
<?php
require_once("../codebase/connector/combo_connector.php");
$res=mysql_connect("localhost","root","");
mysql_select_db("myDB");

$data = new ComboConnector($res, "MySQL");
$data->render_table("categories","id","valueColumn, labelCo
?>
```

- Names of the fields can have aliases (**value** or **label**) to identify the appropriate attribute.

  ```
  $data->render_sql("SELECT *, CONCAT(FirstName, LastName)
  ```

  Note, in the filtering mode a combo filters data by the “label” field.

How can I load data from db?

To load data to a form you should use **FormConnector** on server-side and the method load (id) on client_side:

client-side

```
myForm.load('formdata.php?id=1');
```

where as the parameter must be specified a connector file with the id of loading record.
Values of record's columns will be used as values of form's controls.

server-side

```
<?php
require_once("../codebase/connector/form_connector.php");//
$res=mysql_connect("localhost","root","");//connects to a s
mysql_select_db("tasks");// connects to the DB. 'tasks' is
$conn = new FormConnector($res,"MySQL");// connector initia
$conn->render_table("customers","id","name, address, email"
?>
```

For more information of this topic, see 'dhtmlxForm:easy dealing with server-side' tutorial.

How can I save changes made in a form to db?

To save form changes to DB, you should use the method save() on client-side.
You can call this method, for example, on clicks of some button.

```
myForm.attachEvent("onButtonClick", function(id){
        if (id=='saveButton'){
                myForm.save();
        }
}
```

For more information of this topic, see 'dhtmlxForm:easy dealing with server-side' tutorial.

Component-specific how-tos (grid)

How can I define grid structure on server-side?

To define grid structure on server-side you can use one of two modes:

- automatic
- manual

In both modes you should use set_config() method where as input parameter GridConfiguration object must be specified.

In automatic mode, grid will use names of table's fields as labels for the columns.

```
$config = new GridConfiguration();
$grid->set_config($config);
$grid->render_table("grid50000","item_id","item_nm,item_cd"
```

In manual mode, structure is defined by php command. Names of commands mimic names of js commands with similar functionality.

```
$config = new GridConfiguration();
$config->setHeader(array("column 1","column 2"));
$config->setColTypes(array("ro","ed"));
$grid->set_config($config);
$grid->render_table("grid50000","item_id","item_nm,item_cd"
```

For more information of the topics covered here, see 'defining grid structure on server-side' guide.

How can I export grid's data from server to Excel file?

To export data to Excel file you should:

1. Include one additional file

```
require("../../../codebase/convert.php");
```

2. Activate conversion service

```
$convert = new ConvertService("http://dhtmlx.com/docs
```

3. Start exporting

```
$convert->excel("some.xls",false);
```

   Both method's parameters are optional. The first parameter is a name of output file. The second parameter specifies how file will be exported: *true* - as inline content, *false* - as individual file.

After you call the method excel(), service automatically will start to export data defined through GridConnector.

For more information of this topic, see 'Data export' guide.

How can I export grid's data from server to pdf

file?

To export data to pdf file you should:

1. Include one additional file

   ```
   require("../../../codebase/convert.php");
   ```

2. Activate conversion service

   ```
   $convert = new ConvertService("http://dhtmlx.com/docs
   ```

3. Start exporting

   ```
   $convert->pdf("some.pdf",false);
   ```

   Both method's parameters are optional. The first parameter is a name of output file. The second parameter specifies how file will be exported: *true* - as inline content, *false* - as individual file.

After you call the method pdf(), service automatically will start to export data defined through GridConnector.

For more information of this topic, see 'Data export' guide.

How can I populate select/combo columns with data?

To define options of select/combo columns you have 2 ways:

1. to load data from the same table the grid is populated with data from

```
$grid->set_options("item_nm",array("1" => "one", "2"=
$grid->render_table("grid50","item_id","item_nm,item_
```

2. to load data from another table

```
$options = new OptionsConnector($res);
$options->render_table("countries","country_id","coun
$grid->set_options("item_nm",$options);

$grid->render_table("grid50","item_id","item_nm,item_
```

For more information of this topic, see 'Select/combobox columns in grid' article.

How can I load data from a table that doesn't contain identity field?

Using KeyGridConnector instead of GridConnector, you can load data from a table without identity field. In this case, any data field will serve as identity.

```
$grid = new KeyGridConnector($res);
$grid->render_table("mytable","name","name,address,phone");
```

For more details, see 'KeyGridConnector' guide.

How can I set a custom style for a row or a cell?

dhtmlxConnector contains a bit of methods that allow to set the appearance of a grid.
These methods can be divided into 2 groups:

for a cell customization:

- set_cell_attribute
- set_cell_class
- set_cell_style

for a row customization:

- set_row_attribute
- set_row_color
- set_row_style

```
function color_rows($row){
        if ($row->get_index()%2)
        $row->set_row_color("red");
}

$grid = new GridConnector($res);
$grid->event->attach("beforeRender","color_rows");
$grid->render_table("records","item_id","item_nm,item_cd");
```

Tips:

- to overwrite background you should use '**background**' not '**background-color**' attribute.
- skin's css can overwrite a number of css attributes. To avoid it - use the **!important** flag.

```
$row->set_row_attribute("class","backrgroundclass");
```

```
.backrgroundclass{
   background:red !important;
}
```

Component-specific how-tos (tree/treegrid)

How can I load data from multiple tables?

Using TreeMultitableConnector/TreeGridMultitableConnector instead of TreeConnector/TreeGridConector you can load data from multiple tables.

```
require("../../codebase/treemultitable_connector.php");

$tree = new TreeMultitableConnector($res);
$tree->setMaxLevel(3);
$level = $tree->get_level();
```

For more details, see 'TreeMultitableConnector/TreeGridMultitableConnector' guide.

How can I load data from table that doesn't contain relation field?

Using TreeGroupConnector/TreeGridGroupConnector instead of TreeConnector/TreeGridConector you can load data from table that doesn't contain relation field.

```
require("./codebase/connector/tree_group_connector.php");

$tree = new TreeGroupConnector($res);
$tree->render_table("products2", "id", "product_name", "",
```

For more details, see 'TreeGroupConnector/TreeGridGroupConnector' guide.

How can I mark an item as a leaf or branch of tree?

In beforeRender event handler you are allowed to mark an item as a leaf or branch. For this purpose, you should use set_kids() method. As input parameter *true* or *false* must be specified. *true* marks an item as a branch, *false* - as a leaf.

```
$item->set_kids(false);
```

For more details see 'tree/treeGrid specific' chapter in 'Dynamic loading' guide.

Component-specific how-tos (DataStore)

How can I use connector functionality for dhtmlXDataStore objects?

There are 2 connector types you can use for dhtmlXDataStore objects:

- JSONDataConnector
- DataConnector
  For more details, see the related chapter of this documentation.

DHTMLX Touch how-tos

How can I use connector functionality for DHTMLX Touch components?

There are 2 connector types you can use for touch components:

- JSONDataConnector
- DataConnector
  For more details, see the related chapter of this documentation.

To implement CRUD operation you should use connectors mentioned above together with Touch dataProcessor. For more details, see DHTMLX Touch documentation.

How can I save data changes made in form (simple way)?

Basically, to save data changes you can use code like:

```
dhx.ajax().post("connector.php?action=insert", $$('form').g
```

For more details, see the guide 'Saving data changes made in form'.

Reference

API methods

All the methods can be divided into main 2 categories:

- **Connector methods** - methods of dhtmlxConnector instance. Available throughout the program.
  - ***Connector Object*** - relates to dhtmlxConnector instance.
- **Event methods** - methods available just inside events' handler functions (different events provide different methods. The appropriate ones depend on input object).
  - ***DataAction Object*** - relates to action queries.
    Input object of: beforeInsert, beforeUpdate, beforeDelete, beforeProcessing, afterProcessing, afterDelete, afterUpdate, afterInsert events.
  - ***DataItem Object*** - relates to used dhtmlx component.
    Input object of: beforeRender event.
  - ***SortInterface Object*** - relates to collection of sorting rules.
    Input object of: beforeSort event.
  - ***FilterInterface Object*** - relates to collection of filtration rules.
    Input object of: beforeFilter event.

Order of execution

- **Connector Object** - methods of the main object
  - render_table
  - render_sql
  - set_encoding
  - dynamic_loading
  - enable_log
  - is_select_mode
- **Events objects** - objects that are passed as input parameters to events. Mentioned here methods can be used only inside handlers' functions of the appropriate events.
  - DataAction Object
  - DataItem Object
  - SortInterface Object
  - FilterInterface Object

Connector object

add_section

(*available from version 1.5 and applicable just to DataConnector/JSONDataConnector*)
adds the first child tag(section) to data

```
$conn->add_section($name, $value);
```

Parameters:

- *$name* - the section name
- *$value* -the section value

See also:

- JSONDataConnector/DataConnector

dynamic_loading

enables the dynamical loading mode for connector.

```
$conn->dynamic_loading([$rowsNum]);
```

Parameters:

- *tree, treegrid* - no parameters
- *grid* - number of rows which should be initially loaded (the value should be more than number of rows visible in grid, or at least any positive number)
- *combo* - maximum number of options which server will send for a single data request in autocomplete mode

See also:

- Dynamic loading

enable_log

enables logging for connector.

```
$conn->enable_log("path to log file");
```

Parameters:

- *path* - absolute or relative path to text file where log will be written.

See also:

- Errors handling and logging

is_select_mode

returns current active mode.

```
$conn->is_select_mode();
```

Parameters:

- none.

Returns:

- *true*, if select mode is active.
- *false*, if update mode is active.

render_array

configures connector to take data from a PHP array which can be filled by any kind of external logic.

```
$conn->render_array($data,$id,$text,$extra,$relation_id);
```

Parameters:

- *$data* - name of PHP array.
- *$id* - name of id field.
- *$text* - comma separated list of data fields.
- *$extra* - comma separated list of extra fields, optional.
- *$relation_id* - used for building hierarchy in case of Tree and TreeGrid.

See Also:

- render_sql
- render_complex_sql
- Basic concepts

render_sql

configures connector to take data based on provided SQL text.

```
$conn->render_sql($sql,$id,$text,$extra,$relation_id);
```

Parameters:

- *$sql* - any sql code, which will be used as a base for data selection.
- *$id* - name of id field.
- *$text* - comma separated list of data fields.
- *$extra* - comma separated list of extra fields, optional.
- *$relation_id* - used for building hierarchy in case of Tree and TreeGrid.

See Also:

- render_table
- Basic concepts

render_complex_sql

configures connector to take data based on provided SQL text.
Works similar to render_sql, but have 2 differences:

1. Uses provided sql exactly as it is;
2. Can be used only for data loading (for data saving you can use separate connector, event handlers, or custom model).

The method can be used to call stored procedures in the database.

```
$conn->render_complex_sql($sql,$id,$text,$extra,$relation_i
```

Parameters:

- *$sql* - any sql code, which will be used as a base for data selection or the name of a stored procedure.
- *$id* - name of id field.
- *$text* - comma separated list of data fields.
- *$extra* - comma separated list of extra fields, optional.
- *$relation_id* - used for building hierarchy in case of Tree and TreeGrid.

See Also:

- render_sql
- render_table
- Basic concepts

render_table

configures connector to take data from single table.

```
$conn->render_table($table,$id,$text,$extra,$relation);
```

Parameters:

- *$table* - name of table.
- *$id* - name of id field.
- *$text* - comma separated list of data fields.
- *$extra* - comma separated list of extra fields, optional.
- *$relation_id* - used for building hierarchy in case of Tree and TreeGrid.

Description:

- If you want to render all fields from DB (except of identity field), you can use simplified command:

  ```
  $conn->render_table($table);
  ```

See Also:

- render_sql
- render_complex_sql
- Basic concepts

set_encoding

allows you to set encoding that will be applied to generated XML (default encoding is UTF-8).

```
$conn->set_encoding("iso-8859-1");
```

Parameters:

- encoding name.

set_limit

limits the number of data items that will be loaded to a component.

```
$conn->set_limit([$rowsNum]);
```

Parameters:

- number of data items which should be loaded

DataAction Object

add_field()

allows you to register field for usage while INSERT/UPDATE/DELETE operation.

```
$action->add_field($name,$value);
```

Parameters:

- *$name* - name or alias of the field
- *$value* - value of the field in question

See also:

- remove_field
- Making queries

error()

marks provided data as invalid and cancels default UPDATE processing (invalid record will be highlighted in red bold font).

```
$action->error()
```

Parameters:

- none

See Also:

- invalid
- Validation

get_data()

returns hash of data values that will take part in INSERT/UPDATE/DELETE operations.

```
$action->get_data();
```

Parameters:

- none.

get_id()

returns the id of the currently selected record.

```
$action->get_id();
```

Parameters:

- none.

get_new_id()

returns the ID received after INSERT operation (valid for afterProcessing event only).

```
$action->get_new_id()
```

Parameters:

- none.

get_status()

returns action type of the item.

```
$action->get_status()
```

Returns:

- *update*, *insert*, *delete*, *error* or *custom status* set through set_status() method

Parameters:

- none.

See Also:

- set_status

get_value()

returns value of a field.

```
$action->get_value($name);
```

Parameters:

- *$name* - name of column or alias.

See Also:

- set_value
- Validation

invalid()

marks provided data as invalid and cancels default UPDATE processing (invalid record will be highlighted in bold font).

```
$action->invalid()
```

Parameters:

- none.

See Also:

- error
- Validation

remove_field()

allows to remove one of default fields from INSERT/UPDATE/DELETE operation.

```
$action->remove_field($name,$value);
```

Parameters:

- *$name* - name or alias of the field.
- *$value* - value of the field in question.

See Also:

- add_field

set_response_attribute()

allows you to define custom attribute which will be added to the related action tag in XML response.

```
$action->set_response_attribute($name,$value);
```

Parameters:

- *$name* - name of the attribute.
- *$value* - value of the attribute in question.

See Also:

- set_response_text
- set_response_xml

set_response_text()

allows you to set string that will be included in response

```
$action->set_response_text($text);
```

Parameters:

- *$text* - any text value that will be returned on client-side with related action.

See Also:

- set_response_xml
- set_response_attribute

set_response_xml()

allows you to set xml-string that will be included in response (if you need to provide just a text - use set_response_text() method).

```
$action->set_response_xml($text);
```

Parameters:

- *$text* - any text value that will be returned on client-side with related action.

See Also:

- set_response_text
- set_response_attribute

set_status()

sets the status of operation.

```
$action->set_status($value)
```

Parameters:

- *$value* - status of operation.
  It can be either some predefined value (updated, inserted, deleted, error) or any custom status. Beware, when you set custom status you cancel default data processing.

See Also:

- get_status
- Validation

set_value()

sets value of a field.

```
$action->set_value($name,$value)
```

Parameters:

- *$name* - name or alias of the field
- *$value* - value of the field in question

See Also:

- get_value

success()

confirms that operation has been executed correctly and cancels any default data processing.

```
$action->success()
```

Parameters:

- in case INSERT operation: a new ID of element (which will be updated on client-side)

See also:

- Making queries

DataItem Object

Different components have different sets of methods. Follow the link of component you use.

- dhtmlxChart
- dhtmlxCombo
- dhtmlxDataView
- dhtmlxForm
- dhtmlxGrid
- dhtmlxTree
- dhtmlxTreeGrid
- dhtmlxScheduler

dhtmlxChart

get_id

returns the id of an item.

```
$dataItem->get_id();
```

Parameters:

- none.

See also:

- set_id

get_index

returns the index of the currently selected item (for example, in grid, it will be the index of a row)

```
$dataItem->get_index();
```

Parameters:

- none.

get_value

returns the value of a field.

```
$dataItem->get_value($name);
```

Parameters:

- *$name* - name or alias of the field.

See Also:

- set_value

set_id

sets the id of an item.

```
$dataItem->set_id($value);
```

Parameters:

- *$value* - id of the item.

See also:

- get_id

skip

ignores the current item while rendering output. Allows you to define custom filtration rules.

```
$dataItem->skip()
```

Parameters:

- none.

set_value

sets the value of a field.

```
$dataItem->set_value($name,$value)
```

Parameters:

- *$name* - name or alias of the field
- *$value* - value of the item in question

See Also:

- get_value

set_userdata

(available from version 1.5) sets user data for the item.

```
$dataItem->set_userdata($name,$value)
```

Parameters:

- *$name* - the user data name
- *$value* - user data string

See also:

- Formatting/changing data before loading

dhtmlxCombo

get_id

returns the id of an item.

```
$dataItem->get_id();
```

See also:

- set_id

get_index

returns the index of the currently selected item (for example, in grid, it will be the index of a row)

```
$dataItem->get_index();
```

Parameters:

- none.

get_value

returns the value of a field.

```
$dataItem->get_value($name);
```

Parameters:

- *$name* - name or alias of the field.

See Also:

- set_value

select

marks the specified combo option as selected.

```
$dataItem->select();
```

Parameters:

- none.

set_id

sets the id of an item.

```
$dataItem->set_id($value);
```

Parameters:

- *$value* - id of the item.

See also:

- get_id

skip

ignores the current item while rendering output. Allows you to define custom filtration rules.

```
$dataItem->skip()
```

Parameters:

- none

set_value

sets the value of a field.

```
$dataItem->set_value($name,$value)
```

Parameters:

- *$name* - name or alias of the field
- *$value* - value of the field in question

See Also:

- get_value

set_userdata

(available from version 1.5) sets user data for the item.

```
$dataItem->set_userdata($name,$value)
```

Parameters:

- *$name* - the user data name
- *$value* - user data string

See also:

- Formatting/changing data before loading

dhtmlxDataView

get_id

returns the id of an item.

```
$dataItem->get_id();
```

See also:

- set_id

get_index

returns the index of the currently selected item (for example, in grid, it will be the index of a row)

```
$dataItem->get_index();
```

Parameters:

- none.

get_value

returns the value of a field.

```
$dataItem->get_value($name);
```

Parameters:

- *$name* - name or alias of the field.

See Also:

- set_value

set_id

sets the id of an item.

```
$dataItem->set_id($value);
```

Parameters:

- *$value* - id of the item.

See also:

- get_id

skip

ignores the current item while rendering output. Allows you to define custom filtration rules.

```
$dataItem->skip()
```

Parameters:

- none

set_value

sets the value of a field.

```
$dataItem->set_value($name,$value)
```

Parameters:

- *$name* - name or alias of the field
- *$value* - value of the field in question

See Also:

- get_value

set_userdata

(available from version 1.5) sets user data for the item.

```
$dataItem->set_userdata($name,$value)
```

Parameters:

- *$name* - the user data name
- *$value* - user data string

See also:

- Formatting/changing data before loading

dhtmlxForm

get_id

returns the id of an item.

```
$dataItem->get_id();
```

See also:

- set_id

get_index

returns the index of the currently selected item (for example, in grid, it will be the index of a row)

```
$dataItem->get_index();
```

Parameters:

- none.

get_value

returns the value of a field.

```
$dataItem->get_value($name);
```

Parameters:

- *$name* - name or alias of the field.

See Also:

- set_value

set_id

sets the id of an item.

```
$dataItem->set_id($value);
```

Parameters:

- *$value* - id of the item.

See also:

- get_id

skip

ignores the current item while rendering output. Allows you to define custom filtration rules.

```
$dataItem->skip()
```

Parameters:

- none

set_value

sets the value of a field.

```
$dataItem->set_value($name,$value)
```

Parameters:

- *$name* - name or alias of the field
- *$value* - value of the field in question

See Also:

- get_value

set_userdata

(available from version 1.5) sets user data for the item.

```
$dataItem->set_userdata($name,$value)
```

Parameters:

- *$name* - the user data name
- *$value* - user data string

See also:

- Formatting/changing data before loading

get_id

returns the id of an item.

```
$dataItem->get_id();
```

See also:

- set_id

get_index

returns the index of the currently selected item (for example, in grid, it will be the index of a row)

```
$dataItem->get_index();
```

Parameters:

- none.

get_value

returns the value of a field.

```
$dataItem->get_value($name);
```

Parameters:

- *$name* - name or alias of the field.

See Also:

- set_value

set_cell_attribute

sets an attribute for a cell of item.

```
$dataItem->set_cell_attribute($name,$attr,$value)
```

Parameters:

- *$name* - name of alias of the cell
- *$attr* - name of attribute
- *$value* - value of attribute

See Also:

- set_cell_style
- set_cell_class

set_cell_class

sets a css class for a cell of item.

```
$dataItem->set_cell_class($name,$value);
```

Parameters:

- *$name* - name or alias of the cell
- *$value* - css class for the cell

See Also:

- set_cell_style
- set_cell_attribute

set_cell_style

sets the style for a cell of item.

```
$dataItem->set_cell_style($style);
```

Parameters:

- *$style* - HTML style attributes.

See Also:

- set_cell_class
- set_cell_attribute

set_id

sets the id of an item.

```
$dataItem->set_id($value);
```

Parameters:

- *$value* - id of the item.

See also:

- get_id

set_options

assigns options collection to a column.

```
$dataItem->set_options($name,$options);
```

Parameters:

- *$name* - the name of the column
- *$options* - an array of options or a connector object

Sample:

```
$grid->set_options("column_id",array("1" => "one", "2"=>
//or
$filter1 = new OptionsConnector($res);
$filter1->render_table("countries","country_id","country
$grid->set_options("column_id",$filter1);
```

skip

ignores the current item while rendering output. Allows to define custom filtration rules.

```
$dataItem->skip()
```

Parameters:

- none

set_row_attribute

sets attribute for any row in the component.

```
$dataItem->set_row_attribute($attr,$value);
```

Parameters:

- *$attr* - name of attribute
- *$value* - value of attribute

See Also:

- set_row_style
- set_row_attribute

set_row_color

sets color for any row in the component.

```
$dataItem->set_row_color($color)
```

Parameters:

- *$color* - color of a row

See Also:

- set_row_style
- set_row_attribute

set_row_style

sets style for any row of the component.

```
$dataItem->set_row_style($style);
```

Parameters:

- *$style* - HTML style attributes

See Also:

- set_row_color
- set_row_attribute

set_value

sets value of a field.

```
$dataItem->set_value($name,$value)
```

Parameters:

- *$name* - name or alias of the field
- *$value* - value of the field in question

See Also:

- get_value

set_userdata

(available from version 1.5) sets user data for the item.

```
$dataItem->set_userdata($name,$value)
```

Parameters:

- *$name* - the user data name
- *$value* - user data string

See also:

- Formatting/changing data before loading

get_check_state

returns the state of the selected item.

```
$dataItem->get_check_state();
```

Returns:

- state of checkbox related to the item:
    - '0' - unchecked
    - '1' - checked.
    - '-1' - third (custom) state.

See also:

- set_check_state

get_id

returns the id of an item.

```
$dataItem->get_id();
```

See also:

- set_id

get_index

returns the index of the currently selected item (for example, in grid, it will be the index of a row)

```
$dataItem->get_index();
```

Parameters:

- none.

get_parent_id

returns the parent id of the selected item.

```
$dataItem->get_parent_id();
```

Parameters:

- none

get_value

returns the value of a field.

```
$dataItem->get_value($name);
```

Parameters:

- *$name* - name or alias of the field.

See Also:

- set_value

set_attribute

allows you to set some attribute for a item.

```
$dataItem->set_attribute($name, $value);
```

Parameters:

- *$name* - the name of an attribute.
- *$value* - the value that an attribute is set to.

set_check_state

allows you to set the state of the checkbox related to an item.

```
$dataItem->set_check_state($value);
```

Parameters:

- *$value* - state of the item.
  Valid values are:
  - '0' - unchecked.
  - '1' - checked.
  - '-1' - third custom state.

See also:

- get_check_state

set_id

sets the id of an item.

```
$dataItem->set_id($value);
```

Parameters:

- *$value* - id of the item.

See also:

- get_id

set_image

allows you to set image for any item in tree.

```
$dataItem->set_image($img);
```

Parameters:

- *$img* - relative path to image which will be set for a item.

Description:

- Can be used with variable number of parameters.

  ```
  $dataItem->set_image($img1,$img2,$img3)
  ```

  - *$img1* - image of folder in opened state
  - *$img2* - image of folder in closed state
  - *$img3* - image of leaf item

set_kids

allows you to mark an item as a leaf or branch of tree.

```
$dataItem->set_kids($is_enabled);
```

Parameters:

- *$is_enabled* - (*boolean*) specifies whether an item must be marked as a branch.
  - *true* - item will be marked as a branch.
  - *false* - item will be marked as a leaf.

See also:

- Dynamic loading

skip

ignores the current item while rendering output. Allows you to define custom filtration rules.

```
$dataItem->skip()
```

Parameters:

- none

set_value

sets the value of a field.

```
$dataItem->set_value($name,$value)
```

Parameters:

- *$name* - name or alias of the field
- *$value* - value of the field in question

See Also:

- get_value

set_userdata

(available from version 1.5)
sets user data for the item.

```
$dataItem->set_userdata($name,$value)
```

Parameters:

- *$name* - the user data name
- *$value* - user data string

See also:

- Formatting/changing data before loading

dhtmlxTreeGrid

get_id

returns the id of an item.

```
$dataItem->get_id();
```

See also:

- set_id

get_index

returns the index of the currently selected item (for example, in grid, it will be the index of a row)

```
$dataItem->get_index();
```

Parameters:

- none.

get_parent_id

returns the parent id of the selected item.

```
$dataItem->get_parent_id();
```

Parameters:

- none

get_value

returns the value of a field.

```
$dataItem->get_value($name);
```

Parameters:

- *$name* - name or alias of the field.

See Also:

- set_value

set_cell_attribute

sets an attribute for a cell of item.

```
$dataItem->set_cell_attribute($name,$attr,$value)
```

Parameters:

- *$name* - name of alias of the cell
- *$attr* - name of attribute
- *$value* - value of attribute

See Also:

- set_cell_style
- set_cell_class

set_cell_class

sets the css class for a cell of item.

```
$dataItem->set_cell_class($name,$value);
```

Parameters:

- *$name* - name or alias of the cell
- *$value* - css class for the cell

See Also:

- set_cell_style
- set_cell_attribute

set_cell_style

sets the style for a cell of item.

```
$dataItem->set_cell_style($style);
```

Parameters:

- *$style* - HTML style attributes.

See Also:

- set_cell_class
- set_cell_attribute

set_id

sets the id of the item.

```
$dataItem->set_id($value);
```

Parameters:

- *$value* - id of the item.

See also:

- get_id

set_image

sets image for all check states of the node in tree.

```
$dataItem->set_image($img);
```

Parameters:

- *$img* - relative path to image which will be set for the item of tree (relative to the folder with tree's icons).

skip

ignores the current item while rendering output. Allows you to define custom filtration rules.

```
$dataItem->skip()
```

Parameters:

- none

set_row_attribute

sets an attribute for any row in the component.

```
$dataItem->set_row_attribute($attr,$value);
```

Parameters:

- *$attr* - name of attribute
- *$value* - value of attribute

See Also:

- set_row_style
- set_row_attribute

set_row_color

sets color for any row in the component.

```
$dataItem->set_row_color($color)
```

Parameters:

- *$color* - color of a row

See Also:

- set_row_style
- set_row_attribute

set_row_style

sets the style for any row of the component.

```
$dataItem->set_row_style($style);
```

Parameters:

- *$style* - HTML style attributes

See Also:

- set_row_color
- set_row_attribute

set_value

sets the value of a field.

```
$dataItem->set_value($name,$value)
```

Parameters:

- *$name* - name or alias of the field
- *$value* - value of the field in question

See Also:

- get_value

set_userdata

(available from version 1.5) sets user data for the item.

```
$dataItem->set_userdata($name,$value)
```

Parameters:

- *$name* - the user data name
- *$value* - user data string

See also:

- Formatting/changing data before loading

get_id

returns the id of an item.

```
$dataItem->get_id();
```

See also:

- set_id

get_index

returns the index of the currently selected item (for example, in grid, it will be the index of a row).

```
$dataItem->get_index();
```

Parameters:

- none.

get_value

returns the value of a field.

```
$dataItem->get_value($name);
```

Parameters:

- *$name* - name or alias of the field.

See Also:

- set_value

set_id

sets the id of an item.

```
$dataItem->set_id($value);
```

Parameters:

- *$value* - id of the item.

See also:

- get_id

skip

ignores the current item while rendering output. Allows you to define custom filtration rules.

```
$dataItem->skip()
```

Parameters:

- none

set_options

assigns options collection to a column.

```
$dataItem->set_options($name,$options);
```

Parameters:

- *$name* - the name of the column
- *$options* - an array of options or a connector object

Sample:

```
$scheduler->set_options("type",array("1" => "one", "2"=>
//or
$options = new OptionsConnector($res);
$options->render_table("types","type_id","type_id(value)
$scheduler->set_options("type",$options);
```

set_value

sets the value of a field.

```
$dataItem->set_value($name,$value)
```

Parameters:

- *$name* - name or alias of the field
- *$value* - value of the field in question

See Also:

- get_value

set_userdata

(available from version 1.5) sets user data for the item.

```
$dataItem->set_userdata($name,$value)
```

Parameters:

- *$name* - the user data name
- *$value* - user data string

See also:

- Formatting/changing data before loading

SortInterface Object

add

adds a new sorting rule to collection

```
$coll->add("name","type");
```

Parameters:

- *name* - name of the field.
- *type* - sorting type: ASC or DESC

See also:

- Sorting

clear

clears collection (deletes all specified sorting rules).

```
$coll->clear();
```

Parameters:

- none.

index

checks if there are some rules in collection for the specified field and returns the related hash.

```
$index = $coll->index("name");
```

Parameters:

- *name* - name of the field

Returns:

- *true* - if collection contains rule(s) for the specified field.
- *false* - if collection doesn't contain any rules for the field.

rules

array of all specified sorting rules.

```
$name = $coll->rules[0]["name"];
$coll->rules[0]["direction"]="DESC";
```

Properties:

- *name* - name of related field
- *direction* - direction of sorting

See also:

- Sorting

FilterInterface Object

add

adds a new filtration rule to collection.

```
$coll->add("name","value");  // $name LIKE $value
//or
$coll->add("name","value","comp_oper"); //name < value
```

Parameters:

- *name* - name of the field.
- *value* - comporison value (filtration criteria).
- *comp_oper* - comparison operator: '>' or '<'.

See also:

- Filtering

clear

clears collection (deletes all specified filtration rules).

```
$coll->clear();
```

Parameters:

- none.

index

checks if there are some rules in collection for the specified field and returns related hash.

```
$index = $coll->index("name");
```

Parameters:

- *name* - name of the field

Returns:

- *true* - if collection contains rule(s) for the specified field.
- *false* - if collection doesn't contain any rules for the field.

See also:

- Filtering

rules

array of all specified sorting rules.

```
$name = $coll->rules[0]["name"]
$coll->rules[0]["value"]="cde";
```

Properties:

- *name* - name of related field.
- *value* - comporison value (filtration criteria).
- *operation* - comporison operator. If empty - default LIKE instruction is used.

See also:

- Filtering

Events

events list

- beforeSort Event
- beforeFilter Event
- beforeRender Event
- beforeOutput Event

- beforeUpdate Event
- beforeInsert Event
- beforeDelete Event

- afterUpdate Event
- afterInsert Event
- afterDelete Event

- beforeProcessing Event
- afterProcessing Event

- onDBError Event

order of execution

data loading

- - incoming request parsed
- beforeSort
- beforeFilter
- - sql select executed
- - for each selected record
    - beforeRender
    - - generate xml for record
- - output data to client

data updating

- - incoming request parsed
- - global transaction started (if enabled)
- beforeProcessing
- for each record
    - - record level transaction if started (if enabled)
    - beforeInsert|beforeDelete|beforeUpdate
    - - record level transaction commited (if enabled)
- afterProcessing
- - global transaction commited (if enabled)

beforeSort

The event occurs before data loading and allows to customize sorting of dataset

```
$conn->event->attach("beforeSort",handlerFunc);
```

Parameters *handlerFunc*:

- $sort_by - SortInterface object

Availability:

- Available for Grid, Treegrid

Sample:

```
function custom_sort($sorted_by){
   $sorted_by->rules[0]["name"]="LENGTH(some_field)";// SOR
}
$conn->attach->event("beforeSort","custom_sort");
```

beforeFilter

The event occurs before data loading and allows to customize filtering of dataset.

```
$conn->event->attach("beforeFilter",handlerFunc);
```

Parameters *handlerFunc*:

- $filter_by - FilterInterface object

Availability:

- Available for Grid, Combo

Sample:

```
function custom_filter($filter_by){
       if (!sizeof($filter_by->rules))
             $filter_by->add("some_field","value","LIKE");
//change WHERE some_field LIKE '%value%' to the WHERE some_
       $index = $filter_by->index("some_field");
       if ($index!==false)  //there is client side input fo
             $filter_by->rules[$index]["operation"]=">";
}
$conn->event->attach("beforeFilter","custom_filter");
```

beforeRender

The event occurs event occurs after data has been selected from the database but before its outputting to client.

```
$conn->event->attach("beforeRender",handlerFunc);
```

Parameters *handlerFunc*:

- $dataItem - DataItem Object

Availability:

- Available for Grid, TreeGrid, Tree, Combo, Scheduler, DataView, Chart, Form

Sample:

```
function color_rows($row){
        if ($row->get_index()%2)
        $row->set_row_color("red");
}

$conn->event->attach("beforeRender","color_rows");
```

beforeOutput

The event occurs event occurs after data has been selected from the database and ready to be sent to client side. Event can be used to mix some custom data in XML output. The most common use-case - header structure for the grid.

```
$conn->event->attach("beforeOutput",handlerFunc);
```

Parameters *handlerFunc*:

- Doesn't get any parameters

Availability:

- Available for Grid, TreeGrid, Tree, Combo, Scheduler, DataView, Chart, Form

Sample:

```
function grid_header(){
        echo '<head>
        <column width="50" type="dyn" align="right" color="
        <column width="150" type="ed" align="left" color="#
        </head>';
}
$conn->event->attach("beforeOutput","grid_header");

//In case of dyn. loading mode, one more check need to be a
function grid_header(){
        if (!isset($_GET["posStart"]))
                echo '<head>
                <column width="50" type="dyn" align="right"
                <column width="150" type="ed" align="left"
                </head>';
}
```

```
$conn->event->attach("beforeOutput","grid_header");
$conn->dynamic_loading(100);
```

beforeUpdate

The event occurs before updating values in database and can cancel default update statement or validate incoming values.

```
$conn->event->attach("beforeUpdate",handlerFunc);
```

Parameters *handlerFunc*:

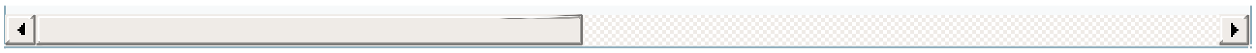
- $action - DataAction Object

Availability:

- Available for Grid, TreeGrid, Tree, Scheduler, DataView, Form

Sample:

```
//creates and runs own update statement using values came i
function myUpdate($action){
        mysql_query("UPDATE Countries SET item_nm='{$action
        $action->success();
}
//or_________________________________________________________
//checks if value of name is empty, then cancel update. Oth
function myUpdate($action){
        if($action->get_value("name")=="")
        $action->invalid();
}
//or_________________________________________________________
//sets new value for name and proceeds with default update.
function myUpdate($action){
        $new_value = rand(0,100);
        $action->set_value("name",$new_value);
}

$conn->event->attach("beforeUpdate","myUpdate");
```



beforeInsert

The event occurs before inserting values into database and can cancel default insert statement or validate incoming values.

```
$conn->event->attach("beforeInsert",handlerFunc);
```

Parameters *handlerFunc*:

- $action - DataAction Object

Availability:

- Available for Grid, TreeGrid, Tree, Scheduler, DataView, Form

Sample:

- See related beforeUpdate sample.

beforeDelete

The event occurs before deleting record in database and can cancel default delete statement or validate incoming values.

```
$conn->event->attach("beforeDelete",handlerFunc);
```

Parameters *handlerFunc*:

- $action - DataAction Object

Availability:

- Available for Grid, TreeGrid, Tree, Scheduler, DataView, Form

Sample:

- See related beforeUpdate sample.

afterUpdate

The event occurs after updating values in database and can can be used to pass additional values to client-side.

```
$conn->event->attach("afterUpdate",handlerFunc);
```

Parameters *handlerFunc*:

- $action - DataAction Object

Availability:

- Available for Grid, TreeGrid, Tree, Scheduler, DataView, Form

Sample:

- See related afterProcessing sample.

afterInsert

The event occurs after inserting values into database and can can be used to pass additional values to client-side.

```
$conn->event->attach("afterInsert",handlerFunc);
```

Parameters *handlerFunc*:

- $action - DataAction Object

Availability:

- Available for Grid, TreeGrid, Tree, Scheduler, DataView, Form

Sample:

- See related afterProcessing sample.

afterDelete

The event occurs after deleting record in database and can be used to pass additional values to client-side.

```
$conn->event->attach("afterDelete",handlerFunc);
```

Parameters *handlerFunc*:

- $action - DataAction Object

Availability:

- Available for Grid, TreeGrid, Tree, Scheduler, DataView, Form

Sample: See related afterProcessing sample.

beforeProcessing

The event occurs before beforeInsert, beforeUpdate, beforeDelete events occur and relates to all operations: Insert, Update and Delete. It can be cancelled in the same way as the aforementioned events.

```
$conn->event->attach("beforeProcessing",handlerFunc);
```

Parameters *handlerFunc*:

- $action - DataAction Object

Availability:

- Available for Grid, TreeGrid, Tree, Scheduler, DataView, Form

Sample: See related beforeUpdate sample.

afterProcessing

The event occurs after beforeInsert, beforeUpdate, beforeDelete events occur and relates to all operations: Insert, Update and Delete. It can be used to pass additional values to client side.

```
$conn->event->attach("afterProcessing",handlerFunc);
```

Parameters *handlerFunc*:

- $action - DataAction Object

Availability:

- Available for Grid, TreeGrid, Tree, Scheduler, DataView, Form

Sample:

```
 //adds new xml as a child of default <action> tag which is
function doAfterProcessing($action){
       $action->set_response_xml("<guid>some value</guid>")
}
$conn->event->attach("afterProcessing",doAfterProcessing);
```

onDBError

The event occurs when some DB level error occurs, during data saving operations. It is relates to all operations: Insert, Update and Delete. It can be used to pass error info to the client side.

```
$conn->event->attach("onDBError",handlerFunc);
```

Parameters *handlerFunc*:

- $action - DataAction Object
- $exception - SQL related exception object

Availability:

- Available for Grid, TreeGrid, Tree, Scheduler, DataView, Form

Sample:

```
 //adds new xml as a child of default <action> tag which is
function doOnDBError($action, $exception){
       $action->set_response_xml("<errorinfo>".((String)$exc
}
$conn->event->attach("onDBError",doOnDBError);
```

Samples

- Client-side initialization of components
- Server-side initialization of components
- Defining grid structure on server-side
- dhtmlxConnector for grid (including filtration, sorting, formatting)
- dhtmlxConnector for form

Samples of client-side initialization

grid

```
myGrid = new dhtmlXGridObject("myContainer");
//... grid configuration commands
myGrid.load("myconnector.php");
```

static mode

```
myGrid = new dhtmlXGridObject("myContainer");
myGrid.setColTypes("tree,...");
//...grid configuration commands
myGrid.loadXML("myconnector.php");
```

dynamic mode

```
myGrid = new dhtmlXGridObject('myContainer');
myGrid.setColTypes("tree,...");
//...
myGrid.kidsXmlFile="connector.php";
myGrid.loadXML("connector.php");
```

static mode

```
tree=new dhtmlXTreeObject("treeboxbox_tree","100%","100%",0
//...
tree.loadXML("connector.php");
```

dynamic mode

```
tree=new dhtmlXTreeObject("treeboxbox_tree","100%","100%",0
//...
tree.setXMLAutoLoading("connector.php");
tree.loadXML("connector.php");
```

static mode

```
var combo=new dhtmlXCombo("combo_zone2","alfa2",200);
combo.loadXML("connector.php");
```

auto-complete mode

```
var combo=new dhtmlXCombo("combo_zone2","alfa2",200);
combo.enableFilteringMode(true,"connector.php",true);
combo.loadXML("connector.php");
```

scheduler

```
scheduler.init(some_obj)
scheduler.load("connector.php");
```

form

```
var form = new dhtmlXForm(someId);
form.load("connector.php");
```

dataView

```
var view = new dhtmlXDataView({...});
view.load("connector.php");
```

chart

```
var mychart = new dhtmlXChart({...});
mychart.load("connector.php");
```

dataStore

```
var myDataStore = new dhtmlXDataStore();
myDataStore.load("connector.php");
```

Samples of server-side initialization

db types

MySQL (php_mysql)

```
require("connector/grid_connector.php");

$res=mysql_connect("localhost","root","");
mysql_select_db("myDatabase");

$gridConn = new GridConnector($res,"MySQL");
```

MySQLi (php_mysqli)

```
require("connector/grid_connector.php");
require("connector/db_mysqli.php");

$mysqli = new mysqli($server, $user, $pass, $mysql_db);
$gridConn = new GridConnector($mysqli,"MySQLi");
```

SQLite (php_sqlite)

```
require("connector/grid_connector.php");
require("connector/db_sqlite.php");
if (!$db = sqlite_open('db', 0777, $sqliteerror)) {
        die($sqliteerror);
}

$gridConn = new GridConnector($db,"SQLite");
```

PostgreSQL (php_pg)

```
require("connector/grid_connector.php");
require("connector/db_postgre.php");
```

```
$res = pg_connect("host=localhost port=5432 dbname=mary");

$gridConn = new GridConnector($res,"Postgre");
```

Oracle (php_oci)

```
require("connector/grid_connector.php");
require("connector/db_oracle.php");

$res = oci_connect("scott", "tiger", $db);

$gridConn = new GridConnector($res,"Oracle");
```

MSSQL (php_mssql)

```
require("connector/grid_connector.php");
require("connector/db_mssql.php");

$res=mssql_connect('.\SQLEXPRESS',"sa","1",false);
mssql_select_db("sampleDB");

$gridConn = new GridConnector($res,"MsSQL");
```

MSSQL (php_sqlsrv)

```
require("connector/grid_connector.php");
require("connector/db_sqlsrv.php");

$conn = sqlsrv_connect( $serverName, $connectionInfo);


$gridConn = new GridConnector($conn,"SQLSrv");
```

PDO (php_pdo)

```
require("connector/grid_connector.php");
require("connector/db_pdo.php");

$res = new PDO('mysql:host=localhost;dbname=test', $user, $

$gridConn = new GridConnector($res,"PDO");
```

File System

```
require("connector/grid_connector.php");
require("connector/db_pdo.php");

$gridConn = new GridConnector("/var/www/test","FileSystem")
$grid->render_table("./", "id", "A,B,C,D,E,F");
```

Excel file

```
require("connector/grid_connector.php");
require("connector/db_excel.php");

$res = new PDO('mysql:host=localhost;dbname=test', $user, $

$grid = new GridConnector("some.xls", "ExcelDBDataWrapper")
$grid->render_table("A18:F83", "id", "A,B,C,D,E,F");
```

Component types

For other components syntax of constructor exactly the same, except, the “Grid” in object name replaced with component's name

Grid

```
require("connector/grid_connector.php");
//...
$gridConn = new GridConnector($res,"MySQL");
```

TreeGrid

```
require("connector/treegrid_connector.php");
//...
$treeGridConn = new TreeGridConnector($res,"MySQL");
```

Tree

```
require("connector/tree_connector.php");
//...
$treeConn = new TreeConnector($res,"MySQL");
```

Combo

```
require("connector/combo_connector.php");
//...
$comboConn = new ComboConnector($res,"MySQL");
```

Scheduler

```
require("connector/scheduler_connector.php");
```

```
//...
$schedulerConn = new SchedulerConnector($res,"MySQL");
```

Form

```
require("connector/form_connector.php");
//...
$formConn = new FormConnector($res,"MySQL");
```

DataView

```
require("connector/dataview_connector.php");
//...
$dataConn = new DataViewConnector($res,"MySQL");
```

Chart

```
require("connector/chart_connector.php");
//...
$chartConn = new ChartConnector($res,"MySQL");
```

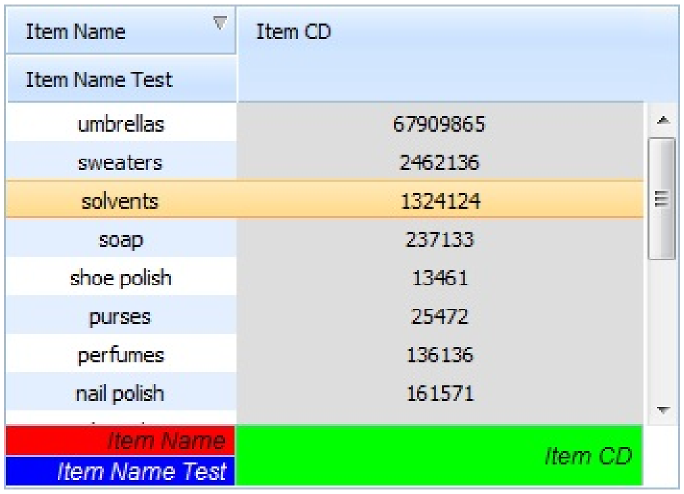
DataStore

```
require_once("../../connector/data_connector.php");
require_once("../../connector/db_sqlite.php");

if (!$db = sqlite_open('db', 0777, $sqliteerror)) {
        die($sqliteerror);
}
$storeConn = new JSONDataConnector($db,"SQLite");
```

DHTMLX Touch Components

```
require_once("../../connector/data_connector.php");
require_once("../../connector/db_sqlite.php");

if (!$db = sqlite_open('db', 0777, $sqliteerror)) {
        die($sqliteerror);
}
$touchConn = new JSONDataConnector($db,"SQLite");
```

Defining grid structure on server-side



Item Name / Item Name Test	Item CD
umbrellas	67909865
sweaters	2462136
solvents	1324124
soap	237133
shoe polish	13461
purses	25472
perfumes	136136
nail polish	161571
Item Name / Item Name Test	Item CD

client-side

index.html

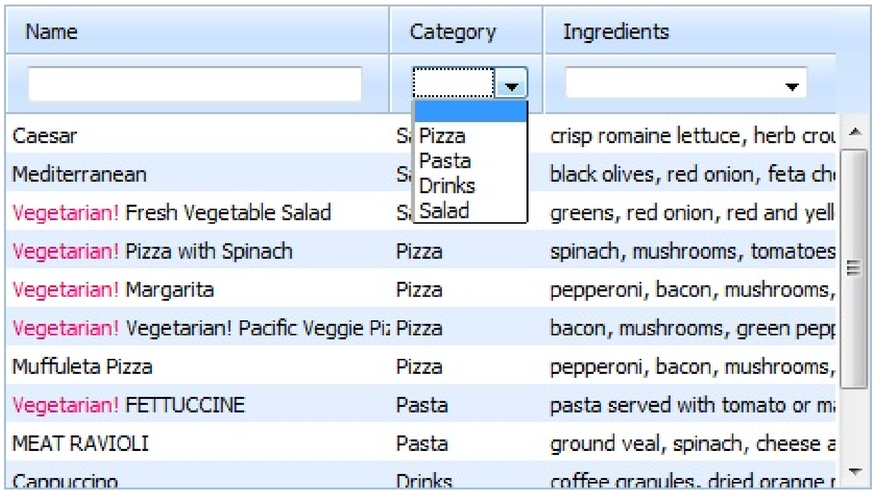
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <script src="codebase/dhtmlx.js" type="text/javascript"
    <script src="codebase/connector/connector.js"></script>
    <link rel="STYLESHEET" type="text/css" href="codebase/d
    <div id="gridbox" style="width:350px;height:250px";bord

    <script type="text/javascript">

     function doOnLoad(){
         var mygrid = new dhtmlXGridObject('gridbox');
         mygrid.setImagePath("codebase/imgs/");
         mygrid.setSkin("dhx_skyblue");
         mygrid.load("griddata.php");
     }
    </script>
</head>
<body onload="doOnLoad()">
</body>
</html>
```

server-side

griddata.php

```
<?php
require_once("../codebase/connector/grid_config.php");
require_once("../codebase/connector/grid_connector.php");

$res=mysql_connect("localhost","root","");
mysql_select_db("tasks");
$conn = new GridConnector($res);
        $config = new GridConfiguration();

        $config->setHeader("Item Name,Item CD");
        $config->attachHeader("Item Name Test,#rspan");
        $config->attachFooter("Item Name,Item CD", Array("ba
        $config->attachFooter("Item Name Test,#rspan", "bac
        $config->setColIds("col1,col2");
        $config->setInitWidths('120,*');
        $config->setColSorting("connector,connector");
        $config->setColColor("null,#dddddd");
        $config->setColHidden("false,false");
        $config->setColTypes("ro,ed");
        $config->setColAlign('center,center');

$conn->set_config($config);
$conn->render_table("grid50000","id","item_nm,item_cd");

?>
```

dhtmlxConnector for grid



Name	Category	Ingredients
Caesar	S	crisp romaine lettuce, herb crou
Mediterranean	S	black olives, red onion, feta ch
Vegetarian! Fresh Vegetable Salad	S	greens, red onion, red and yell
Vegetarian! Pizza with Spinach	Pizza	spinach, mushrooms, tomatoes
Vegetarian! Margarita	Pizza	pepperoni, bacon, mushrooms,
Vegetarian! Vegetarian! Pacific Veggie Pi	Pizza	bacon, mushrooms, green pepp
Muffuleta Pizza	Pizza	pepperoni, bacon, mushrooms,
Vegetarian! FETTUCCINE	Pasta	pasta served with tomato or m
MEAT RAVIOLI	Pasta	ground veal, spinach, cheese a
Cappuccino	Drinks	coffee granules, dried orange

client-side

index.html

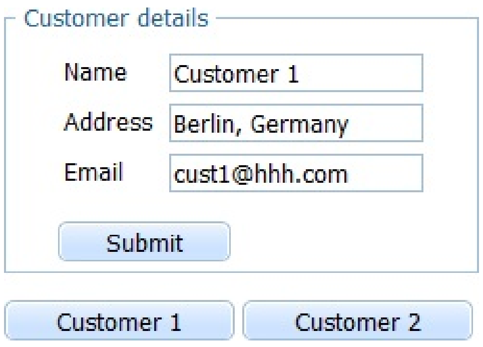
```
<!DOCTYPE html>
<head>
    <script src="codebase/dhtmlx.js" type="text/javascript"
    <script src="codebase/connector/connector.js"></script>
    <script src="codebase/connector/dataprocessor.js"></scr
    <link rel="STYLESHEET" type="text/css" href="codebase/d
    <div id="gridbox" style="width:450px;height:250px";bord
<script type="text/javascript">



var mygrid,myDP;
function doOnLoad(){
    mygrid = new dhtmlXGridObject('gridbox');
        mygrid.setImagePath("codebase/imgs/");
        mygrid.setHeader("Name,Category,Ingredients");
        mygrid.attachHeader("#connector_text_filter,#connec
        mygrid.setInitWidths("200,80,*");
        mygrid.setColTypes("ro,ed,ed");
        mygrid.setColSorting("str,connector,str");
        mygrid.init();
        mygrid.load("xml/griddata.php");

        myDP = new dataProcessor("xml/griddata.php");
        myDP.init(mygrid);

}
</script>
</head>
<body onload="doOnLoad()" >
</body>
</html>
```

server-side

griddata.php

```
<?php
require_once("../codebase/connector/grid_connector.php");

$res=mysql_connect("localhost","root","");
mysql_select_db("tasks");
$conn = new GridConnector($res);
$conn->enable_log("temp.log");

function custom_sort($sorted_by){
          if (!sizeof($sorted_by->rules))
               $sorted_by->add("group","DESC");
}
function formatting($row){
                  //render field as details link
                                 if ($row->get_value("vege
                                      $row->set_value("nam
        }
function validate($data){
             if ($data->get_value("group")=="")
                  $data->error();
}
$conn->event->attach("beforeProcessing","validate");
$conn->event->attach("beforeSort","custom_sort");
$conn->event->attach("beforeRender","formatting");

$conn->render_table("meals","id","name,group, ingredients",

?>
```

dhtmlxConnector for form



index.html

```
<!DOCTYPE HTML>
<html>
<head>
        <link rel="STYLESHEET" type="text/css" href="../code
        <script  src="../codebase/dhtmlx.js"></script>
        <script type="text/javascript" src="../codebase/conn
        <div id="box" style="width:250px; height:160px; bac
</head>

<body>
<script type="text/javascript">
//---form configuration. Defining controls
        formData = [
                                {type: "block", list:[
                                {type: "fieldset",  name: "
                                        {type: "input", nam
                                        {type: "input", nam
                                        {type: "input", nam
                                        {type: "button", na
                ]}]},
                                {type: "block", list:[
                                {type: "button", name:"butt
                                {type:"newcolumn"},
                                {type: "button", name:"butt
        ];
```

```
//---form initialization
        var myForm = new dhtmlXForm("box",formData);//objec
//---loading data from db.
        myForm.load('formdata.php?id=1');// as the paramete
//---dataProcessor initialization
        var mydp = new dataProcessor ("formdata.php");// ob
        mydp.init(myForm);// initializes dataProcessor. As

        myForm.attachEvent("onButtonClick", function(id){
                        if (id=='button1'){//
                                myForm.load('formdata.php?i
                        }
                        else if (id=='button2'){
                                myForm.load('formdata.php?i
                        }
                        else if (id=='save'){
                                myForm.save();// saves data
                        }
    });
</script>
</body>
</html>
```

formdata.php

```
<?php
  require_once("../codebase/connector/form_connector.php");
  $res=mysql_connect("localhost","root","");
  mysql_select_db("tasks");
  $conn = new FormConnector($res,"MySQL");
  $conn->render_table("customers","id","name, address, emai
?>
```