

Request modes

Data Loading

```
Request mode:      GET
GET parameters:
POST parameters:
```

In data loading mode connector need to return all records from relate table|collection|query. Basically it is the request for "SELECT *" query.

Data Loading, with sorting

```
Request mode:      GET
GET parameters:    ?dhx_sort[{index}]=mode
POST parameters:
```

Connector need to return all records from relate table|collection|query sorted by provided parameter Query can contain multiple dhx_sort parameters

- {index} - index or name of field for which sorting need to be applied
- {mode} - "asc" or "desc"

Data Loading, with filtering

```
Request mode:      GET
GET parameters:    ?dhx_filter[{index}]=filter
POST parameters:
```

Connector need to return all records from relate table|collection|query sorted by provided parameter Query can contain multiple dhx_filter parameters

- {index} - index or name of field for which filtering need to be applied
- {filter} - filter value

By default filter applied as "LIKE %filter%" If mutiple filters provided - they combined by AND logic

Partial Data Loading

```
Request mode:      GET
GET parameters:    ?posStart={start}&count={count}
POST parameters:
```

Connector need to return {count} of records starting from position {start}

- {count} - count of requested records
- {start} - index of first requested record

This is equal to usage of LIMIT construction in MySQL

Combined data loading

All above modes can be combined

```
?dhx_sort[Name]=asc&dhx_sort[Surname]=asc&dhx_filter[Country]=Ru&dhx_filter[Status]=1&posStart=100&count=10
```

Select records 100-110, where Country contains "RU" and Status contains "1", sorted by Name and Surname

```
?dhx_sort[0]=asc&dhx_sort[1]=asc&dhx_filter[2]=Ru&dhx_filter[3]=1&posStart=100&count=10
```

Same as above, but instead of the names, field indexes are used. Indexes are related to the order of parameters in render_table or equal command on the server side.

Form loading

This mode is used only by Form connector, and not necessary for other types

```
Request mode:      GET
GET parameters:    ?id={id}
```

POST parameters:

Connector need to return single record from related table|collection|query, with record.id equal to {id}

Autocomplete mode

This mode is used only by Combo connector, and not necessary for other types

Request mode: GET
GET parameters: ?mask={filter}
POST parameters:

Connector need to return all records from relate table|collection|query Filtering must be done against the main data field of the connector. (combo filter has only one data field)

{filter} - filter value

By default filter applied as "LIKE %filter"

Data saving modes

Simple insert

```
Request mode:      POST
GET parameters:    ?!nativeeditor_status=inserted
POST parameters:   id={id}&key1=value1&key2=value2...&keyN=valueN
```

Server side code need to insert new record in the database, id provided from client side can (and preferable must) be ignored. Values for different fields provided as key=value pairs in the POST request

Response

```
Content-type:      text/xml
<data>
  <action sid='{id}' tid='{realid}' type='insert'></action>
</data>
```

Where

- {id} - id from incoming requested
- {realid} - id of new record in the database

If operation can't be executed, response must look as

```
<data>
  <action sid='{id}' tid='{id}' type='error'></action>
</data>
```

Simple update

```
Request mode:      POST
GET parameters:    ?!nativeeditor_status=updated
POST parameters:   id={id}&key1=value1&key2=value2...&keyN=valueN
```

Server side code need to update record in the database Id and values for different fields provided as key=value pairs in the POST request

Response

```
Content-type:      text/xml
<data>
  <action sid='{id}' tid='{id}' type='update'></action>
</data>
```

Where

- {id} - id from incoming requested

If operation can't be executed, response must look as

```
<data>
  <action sid='{id}' tid='{id}' type='error'></action>
</data>
```

Simple delete

```
Request mode:      POST
GET parameters:    ?!nativeeditor_status=deleted
POST parameters:   id={id}
```

Server side code need to delete record from a database

Response

```
Content-type:      text/xml
<data>
  <action sid='{id}' tid='{id}' type='delete'></action>
</data>
```

Where

- {id} - id from incoming requested

If operation can't be executed, response must look as

```
<data>
```

```
<action sid='{id}' tid='{id}' type='error'></action>
</data>
```

Grid and Tree specific parameter names

Grid, TreeGrid

- gr_id - id of record (used instead of "id")
- gr_pid - id of parent record (for treegrid only)
- c0 - value of first column
- c1 - value of second column
- ...
- cN - value of Nth column

Tree

- tr_id - id of tree item
- tr_pid - id of parent item
- tr_order - index of item in the parent branch
- tr_text - text value of tree item

Complex saving

Client side can issue multiple saving operation in single request. It can be detected by checking POST["ids"] parameter, if it exists - we are in the complex saving mode.

POST["ids"] contains the comma separated list of records ID

```
var list = POST["ids"].split(",");
for (var i=0; i<list.length; i++){
    var id = list[i];
    var mode = POST[id+"_!nativeeditor_status"];
    var value1 = POST[id+"_key1"];
    ...
    //exec operation, same as for simple update mode
}
```

Response

Response is similar to the simple mode, but will contain multiple "action" tags, one for each id in the "ids" collection

```
<data>
  <action sid='112312312' tid='12' type='insert'></action>
  <action sid='4' tid='4' type='update'></action>
  ...
</data>
```

Simple form operations (dhtmlx Touch)

Loading

```
Request mode:      GET
GET parameters:    ?action=get&id={id}
POST parameters:
```

Return info about single record, which is selected by provided {id}

Saving

insert

```
Request mode:      GET
GET parameters:    ?action=insert
POST parameters:    key1=value1&key2=value2...keyN=valueN
```

Response

```
Content-type:      text/plain
true\n{new_id}
```

Error response

```
Content-type:      text/plain
Error description
```

Update

```
Request mode:      GET
GET parameters:    ?action=update
POST parameters:    id={id},key1=value1&key2=value2...keyN=valueN
```

Response

```
Content-type:      text/plain
true
```

Error response

```
Content-type:      text/plain
Error description
```

Delete

```
Request mode:      GET
GET parameters:    ?action=delete
POST parameters:    id={id}
```

Response

```
Content-type:      text/plain
true
```

Error response

```
Content-type:      text/plain
Error description
```

Data loading response

Grid

```
Content-type: text/xml
<rows>
  //foreach record
  <row id="{id}">
    //foreach field
    <cell>{fieldvalue}</cell>
    //end foreach
  </row>
  //end foreach
</rows>
```

Grid in dyn. load mode

```
Content-type: text/xml
<rows pos="{start}">
...continue the same as for normal Grid...
```

TreeGrid

```
Content-type: text/xml
<rows>
  //foreach record
  <row id="{id}">
    //foreach field
    <cell>{fieldvalue}</cell>
    //end foreach
    //foreach subrow
    <row id="{id}">
      ...continue recursively...
    </row>
    //endforeach
  </row>
  //end foreach
</rows>
```

TreeGrid in dyn. load mode

```
Content-type: text/xml
<rows parent="{parent_id}">
...continue the same as for normal TreeGrid...
```

Tree

```
Content-type: text/xml
<tree>
  //foreach record
  <item id="{id}" text="{text}">
    //foreach subitem
    <item id="{id}" text="{text}">
      ...continue recursively...
    </item>
    //endforeach
  </item>
  //end foreach
</tree>
```

Tree in dyn. load mode

```
Content-type: text/xml
<tree id="{parent_id}">
...continue the same as for normal TreeGrid...
```

Combo

```
Content-type: text/xml
<complete>
  //foreach record
  <option value='{id}'>
    {text}
  </option>
//end foreach
</complete>
```

Dataview, Chart, Datastore, Touch

```
Content-type: text/xml
<data>
  //foreach item
  <item id="{id}">
    //foreach field
    <{fieldname}>
      {fieldvalue}
    </{fieldname}>
    //endforeach
  </item>
//end foreach
</data>
```

Scheduler

```
Content-type: text/xml
<data>
  //foreach item
  <item id="{id}">
    <start_date>{start_date}</start_date>
    <end_date>{end_date}</end_date>
    <text>{text}</text>
    //foreach custom field
    <{fieldname}>
      {fieldvalue}
    </{fieldname}>
    //endforeach
  </item>
//end foreach
</data>
```

Json connector for Touch and datastore

```
Content-type: text/plain
[
  //for each record
  {
    //for each field
    {fieldname}:{fielvalue}",
    //end for each
    id:"{id}"
  }
  //end for each
]
```

Recomendations for server side

Connector types

There are next types of connectors available in php edition

- GridConnector
 - has custom parameter names during saving (check above)
 - can be used with dyn. loading mode (loading only visible ranges of data)
- TreeGridConnector
 - has custom parameter names during saving (check above)
 - can be used with dyn. loading mode (loading only single branch)
- TreeConnector
 - has custom parameter names during saving (check above)
 - can be used with dyn. loading mode (loading only single branch)
- ComboConnector
 - has special autocomplete mode (check above)
- FormConnector
 - has special single record loading mode (check above)
- DataViewConnector
- ChartConnector
- DataConnector
- SchedulerConnector
- JSONDataConnector
 - output data as json, not as xml

If you target DHTMLX3 you need to implement GridConnector, TreeGridConnector, TreeConnector, ComboConnector, FormConnector, DataViewConnector, ChartConnector, SchedulerConnector

If you target DHTMLX Touch you need to implement only DataConnector and JSONDataConnector

Configuration options

Connectors must allow two main rendering modes - render from the table - render from the sql query

During connector initialization user can select - master table|query - id field - data fields - parent_id (for tree and treegrid)

Events

There must be some way to assign custom logic to the key points of data processing. Without such ability connector will be hardly customizable

Data loading

- BeforeRender - for each record, before converting to xml - allows to define custom rules
- BeforeSorting, BeforeFiltering - allows to modify sorting|filtering parameters

Data Saving

- BeforeProcessing - allows to modify incoming data, or block operation
- AfterProcessing - allows to make some post-operation update, or include custom data in response