FGS Factory Framework Guide

FGS Factory Framework (the Framework) is a part of FGS Tools software package. This document presents main features, architecture and interaction between different parts of FGS Factory Framework.

Component architecture of the Framework

The Framework is based on three visual components **Form**, **Grid**, **Search** and one non-visual component **Dataset**. The **Form** enables a user to add, edit, delete and view table rows. The **Grid** enables the user to view, navigate, and edit individual table rows in a tabular manner or in a spreadsheet-like format. The **Search** enables the user to enter row filters which are used by the Grid. The **Dataset** creates hidden filters for table rows.

Components **Form**, **Grid**, **Search** and **Dataset** which create user interface to one common database table are united in a component **Unit**. The **Unit** component serves to make interaction among its components and, therefore, serves as a controller.

Components interact among themselves in the following ways:

- **Dataset** sets hidden row filters for **Grid**
- **Search** sets explicit (entered by the user) row filters for **Grid**
- **Grid** transfers necessary data to **Form** for editing, deleting or viewing table rows

Such interaction of components imposes the following constraints on composition of one **Unit**:

- There maybe be only one component of each type
- There must be at least one component **Form** or **Grid**
- If there is no a **Grid** in a **Unit**, then components **Search** and **Dataset** are useless
- Components **Search** and **Dataset** are optional components

Components have a set of attributes, defining its state. These attributes are stored / restored in/from session variables.

The Framework presents three types of user interface to database tables:

- interface to one database table (controller **Crud**)
- interface to two database tables associated by «master-detail» relationship (controller **MasterDetail**)
- Interface to a number of database tables associated or not by «master-detail» relationship. One of these tables must be master table and the rest must have or have not relationship master- detail with the master table (controller **UnitSet**)

Each interface has got its own controller. A name of a controller is used as a name of the corresponding interface. So interface to one database table is named as **Crud** interface and interfaces presented by the **MasterDetail** and **UnitSet** controllers are called **MasterDetail** and **UnitSet** interfaces accordingly.

The Crud controller consists only of one **Unit**.

The MasterDetail controller consists of two Units (the master **Unit** and the detail **Unit)** and performs interaction between components of different Units.

The UnitSet controller consists of two or more Units (one master Unit and the rest are detail or auxiliary Unit) and performs interaction among components of different Units.

Interactions among components are performed by requests and events. A request arises when the user clicks on a hyperlink (GET request) or presses on a HTML form button (GET or POST request). Events are generated by components in response to requests but this is not mandatory. Registration of possible requests and dispatching of incoming requests are handled by a request handler (**RequestHandler** component). Registration of possible events and dispatching of generated events are handled by an event handler (**EventHandler** component).

Each type of controllers is a descendent of an abstract class **Controller**.

Data exchange among components, saving and restoring of components state are handled by the **Registry** component.

Accesses to data are handled by t**he AccessHandler** component.

Connecting to a database, querying and data manipulation are performed by a special database driver.

The Timer component is used for profiling execution time of scripts.

The Form component is called also as input form and consists of input elements and buttons.

The Grid component consists of output or input elements of **the Column** type and elements to choose action on an individual row or a rows' set. **The Grid** component may also be used for editing rows.

The Search component is called also as search form and consists of components for entering search conditions or predicates by the user.

The Dataset component consists of predicates, which are not entered by the user but are known earlier or can be known only at run time.

Input and output elements can use dynamic lists having database table's rows as options. Loading such lists is handled by a list loader (**the ListLoader** component). Using of **ListLoader** makes it possible to load a list only once and use it by different components.

Components files have extension **inc**. Paths to components files are defined in the "classes.php" file in the $cp array of elements. This array of elements uses Function __autoload.

Some helper components realize Singleton design pattern. These components are created at the very start of operation of a control script by including a bootstrap script. Access to instances of these components is performed through global variables:

$timer is the only instance of **the Timer** component
$db is the only instance of the current database driver
$registry is the only instance of **the Registry** component
$requestHandler is the only instance of **the RequestHandler** component
$eventHandler is the only instance of **the EventHandler** component
$listLoader is the only instance of **the ListLoader** component
$accessHandler is the only instance of **the AccessHandler** component

Concept of specifications

The Framework uses concept of specification: appearance and behavior of a component are ruled by its specification. The Framework creates instances of components from its specifications. To change a component's appearance or behavior, we have to change its specification.

Component's specification is understood as a set of attributes, defining appearance and behavior. These attributes can be conventionally divided into three groups:

- HTML attributes
- HTML events
- System attributes

HTML attributes and HTML events are defined by HTML standard for the corresponding HTML element if such exists. System attributes are the attributes, which determine the behavior of the components operation.

For example, a specification for **the Form** component consists of the attributes:

- HTML attributes: method, enctype, action, name and others related to HTML element «form»
- HTML events: onreset, onsubmit related to HTML element «form»
- System attributes: system ID, array of input elements, buttons, name of database table etc

Specifications are stored in special database tables. These tables are used only by the Configurator and called as "system tables". A "system" means using a database table only by the Configurator. A Developer uses **the Configurator** to create and to change specifications. To speed up operation, the data from these tables is exported to specification's files. The Framework uses these files to create instances of components. Each interface language (English, Russian etc) has its own set of specifications, which are stored in a separate directory. Besides, the specifications of different types of components are stored in separate directories.

To define the path to components specifications, the Framework uses the following constants:

component	Contant of a path
controller	CONTROLLER_PATH

Input form	FORM_PATH
Grid	GRID_PATH
Search form	SEARCH_PATH
Dataset	DATASET_PATH
List	LIST_PATH
database table	TABLE_PATH
menu	MENU_PATH
Input element	ELEMENT_PATH
Output column	COLUMN_PATH

A path to a component's specification file is defined as follows:

Path constant of a component/ Interface language

Structures of specifications are developed to minimize coding. Creation of components' specifications is described in the «The Confgurator Manager guide» document.

Every component has a set of properties stipulating its state. These state properties and its initial values properties are defined by its specification. A specification and state of a component determine a set of requests and events that it can process. For example if an input form (**Form)** is in mode «insert» of a new row, then it cannot update or remove a row. In addition, if an input form's specification has only «update» mode, then all requests to insert or to delete a row are ignored.

As every controller provides an interface to a database access, so therefore there should be some access control mechanizm. For this purpose, a controller's specification has so called Access Control List (the "acl" attribute) or a list to control an access. The list may include roles and users who can have database access given by this controller.

It's necessary to note that some system attributes are not entered explicitly by the developer but are created only during specification's exporting based on the data already eneterd. **The Configurator** makes some checks of entered data and displays warning of discovered errors but there can't be guarantee of other errors. So the developer should follow rules of specifications' input and update specifications according to **the Configurator's** warnings.

Requests processing algorithm

To successfully use the Framework, it's necessary to know how the Framework processes an HTTP request from a client (browser), what operations and in what sequence the Framework does.

The FGS Tools package includes two examples of control scripts: configurator.php and application.php. The script configurator.php is the control script for the specifications' Configurator and the application.php script is the control script for applications developing based on the package.

At first we need to define concept of a request's «window».

To imitate different sessions in one window of a browser and to perform concurrent running of instances of the same controller, the Framework uses concept of a request's «window». Each instance of the controller is running in its own «window».

At the beginning, a control script (for example configurator.php) tries to set an incoming request's «window» or the current request window. If the fgsw parameter is passed in the request then the window with id equal to the fgsw parameter is set as the current request window. But if there is not the fgsw parameter in the request then the window with id equal to «main» is set as the current request window.

On the second step, the control script tries to set the current controller, which should be run in the current request window. If there is the fgsc parameter in the request, then the controller with id equal to the fgsc parameter is set as the current controller. But if there is no the fgsc parameter in the request and if a controller was already run in the current request window, then the previous controller is set as the current controller. If there is no the fgsc parameter in the request and if there was no any running controller in the current request window then the control script just displays the menu of **Configurator** or a an application.

On the third step, the control script creates the current controller.

We need to consider the last part of controller's constructor (**Crud**, **MasterDetail** or **UnitSet**):

```
if($registry->newController())
{
	$this->initState();
}
else
{
	$this->restoreState();
}
```

This code performs the following actions:

If the current controller was not run earlier in the current request window, then the constructor initiates the controller's state. If the current controller was run earlier in the current request window then the constructor restores the controller's state.

If there is the fgsinit parameter in the request, then the constructor also initiates the controller's state.

Remember that a controller contains Unit components, which themselves can consist of components **Form**, **Grid**, **Search** and **Dataset**. The last listed components also are containers. Therefore, saving and restoring a controller's state means also saving and restoring state of all its child's components.

After creation of the current controller the control script invokes the method «run» of the controller.

Let's consider code of the method «run» of **the Controller** component.

Checking of access control to the controller is performed on Step 4. This is performed by **the AccessHandler** component using Access Control List. The Access Control List is an attribute of a controller's specification and has roles and users that have permission to run the controller.

Registration of possible request and events is performed on Step 5.

Usually a GET request is created via clicking on a hyperlink and a POST request via pressing on a submit button. The Framework processes GET or POST requests initiated or triggered only by **a Form**, **Grid**, **Search** components and **Controllers**.

A Form, **Grid or Search** component register possible requests and events taking into account only its own specification. A **Unit** component registers possible request and events triggered or processed by itself or its components. A **MasterDetail** or **UnitSet** component registers possible requests and events that triggered or processed by components of different Units.

Registration of possible requests and events is performed by the «subscribe» method.

The Framework identifies unambiguously a GET or POST request triggered by a component by four parameters:

Id of a request window (the fgsw parameter)
Id of a controller (the fgsc parameter)
Id of a request's triggers (the fgst parameter)
Id of a Unit which contains a request initiator or trigger (the fgsu parameter)

All components have to add these parameters fgsc, fgst and fgsu to their requests. The fgsw parameter is optional and it has to be added if it's necessary to run a request in a window, which is different from of the «main» window. Components make this either by adding hidden fields to HTML forms or adding these parameters to hyperlinks.

Dispatching of a GET or a POST request is performed on step 6. There maybe generated events during processing the request. For example, a **Form** component generates the event «done» after successfully executed inserting, updating or removing a row.

It's necessary to note that there can be only one incoming/processing request and a number of generated events.

Dispatching of generated events is performed on step 7. During processing of events, components can also generate events.

Loading data for rendering is performed on step 8. Usually it's loading rows to display in a Grid component and/or loading a row for a Form component.

Saving of the current controller's state and all its components is performed on step 9

HTML-code generating or page rendering is performed on the last step 10. The Framework has standard classes FgsFormView, FgsGridView, FgsSearchView to render Form, Grid and Search components accordingly. To render input fields and buttons of Form and Search components, the Framework uses the FormKit component, while to render input/output elements of a Grid, it uses GridKit component.

So processing of a request includes 10 steps:

1. Setting the request current window
2. Setting the current controller
3. Creation of the current controller, initialization or restoring of the controller's state
4. Access control checking
5. Registration of possible requests and events
6. Dispatching the request to process
7. Dispatching generated events to process
8. Data loading for page rendering
9. Saving the controller's state
10. Page rendering

Interaction among components of a Unit

Let's consider interaction among **Grid, Form and Unit** components during a row editing.

A row editing is performed in two steps. Usually a component Form is in mode «insert». Therefore the Form component has to be set in mode «update»

At first consider transition of **the Form** component to «update» mode.

Each row in the Grid component has the icon to update this row.

The Grid component registers as a possible request the request to update a row. This request is triggered by the Grid component itself.

The Form component registers as a possible event the event to update a row. This event is generated by the Grid component.

After the user clicks on the icon to update a row **the RequestHandler** component discovers the request to update and sends it to process for the Grid component.

The Grid component validates the incoming request and if everything is OK generates the event to update the chosen row.

The EventHandler component discovers this event and sends it to process for the Form component (step 7).

Using the event's arguments, the Form component changes its mode to «update», loads the chosen row and initiate values of its input elements.

After that the control script performs the rest steps of request processing. So the first step of a row editing is completed.

Let's consider the stage of real of row editing.

As **the Form** is now in the «update» mode, it registers as a possible request the request to save the updated row. The request is triggered by the Form component itself.

The Unit component registers as a possible event the event of successful execution of the update operation that generated by the Form component.

After the user edited the chosen row and pressed on the «Save» button **the RequestHandler** component discovers the request to save updated row and sends it to process for the Form component (step 6).

After validation of input data **the Form** component forms data for SQL query and sends this data to the current database driver to perform row updating. If row updating is successfully executed then the Form component generates the event «done» about successful execution of updating. After that the Form component changes its mode to the «insert» mode and initiates values of its input elements.

The EventHandler component discovers this event and sends it to process for the Unit component (step 7).

The Unit component processes the received event by setting visibility attributes of its components.

After that the control script performs the rest steps of request processing.

Let's consider interaction between **Grid** and **Search** components during setting of a new row filter for the Grid component.

The Search component is always ready to input a new row filter. Therefore setting of the new row filter is performed at once.

The Unit component registers as a possible event for the Grid component the event of setting a new row filter which generated by the Search component.

The Search component registers as a possible request the request to set a new row filter. The request is triggered by the Search component itself.

After the user typed data of a new row filter and pressed on the «Set Filter» button the **RequestHandler** component discovers the request to set the new row filter and sends this request to process for the Search component (step 6).

After successful validation of entered data the Search component generates the event «search» about setting the new row filter.

The Grid component processes the received event by setting value 0 for its «offset» attribute.
.
After that the control script performs the rest steps of request processing.

Interaction between components of different Units

Let's consider interaction of components of different Units for a MasterDetail controller which contains master Unit and detail Unit. Remember that this controller presents user interface to two database tables with master- detail relationship. These tables are called the master and detail tables.

At the very beginning of operation of the controller of the type MasterDetail, the user has access only to the master table (master mode of the controller) interface.

Each row in the Grid of the master Unit (master Grid) has got the icon to activate user interface to detail table (detail mode of the controller).

The master Grid component registers as a possible request the request to activate user interface to detail table. This request is triggered by the master Grid itself.

In master mode the controller registers for itself and for Form and Grid components from the detail Unit as a possible event the event «detail», which is generated by the master Grid component.

After the user clicked on the icon to activate user interface to detail table on the step 6, the **RequestHandler** component discovers the request to activate and sends it to process for the master Grid.

The master Grid validates the incoming request and if everything is OK it generates the event «detail» to activate interface to the detail table for the chosen row. The value of the primary key of the chosen row is sent as a parameter of the event.

The **EventHandler** component discovers this event and sends it to process for the controller, the detail Form and the detail Grid (step 7).

The controller changes its mode to the «detail» mode. In this mode the controller displays only the master Grid with the chosen row and components of the detail Unit.

Using the event's arguments, the detail Form component changes its mode to the «insert» mode and initiate values of its input elements. Also the detail Form stores the received primary key and uses it as foreign key value for the detail table.

The detail Grid also processes the received event. The detail Grid stores the received primary key and uses it as foreign key value for the detail table.

After that the control script performs the rest steps of request processing.

In the «detail» mode possible requests and events are registered only for components of the detail Unit and the request to return to the «master» mode for the controller.

Database drivers

The Framework was developed to use with different database management systems (DBMS).

The Framework interacts with a database through special drivers which takes into account peculiarities of the database.

The FGS Tools package needs two drivers to use a database:

- A driver for data manipulation
- A driver for loading tables' structure into system tables of the Configurator

A driver for data manipulation has to be coded with Singleton design pattern and have the following methods:

method	Comments
getInstance	To get the only instance of a driver
__construct	Constructor must be of private type
table_join	Method of making table join
connect	Method to create database connection
value	Method to select only one value from a table
page	Method to select rows with the limit clause
set	Method to select rows without the limit clause
row	Method to select only one row from a table by primary key
record	Method to select one row by a SQL query
execute	Method to execute a SQL query
insert	Method to insert a new row by a array of fields and theirs values
update	Method to update a row by a array of fields and theirs values and its primary key
remove	Method to remove a row by its primary key
code	Method to code values before storing them in database
decode	Method to decode values retrieved from database
IsTime	Method to determine that a field type is of «time» type
IsText	Method to determine that a field type is of «text» type
IsNumber	Method to determine that a field type is of «number» type
IsBlob	Method to determine that a field type is of «binary» type
IsAggregateFunction	Method to determine that a SQL function is a aggregate function
localdatetime	Method to format a date and time in local format
localdate	Method to format a date in local format

The class constructor also has to create an array of comparison conditions codes and theirs SQL equivalents of a database.

Components Form and Grid create all necessary SQL queries and the developer does not need to manually code SQL queries for standard operations such as inserting, updating, removing and searching rows.

Handling of lists

Lists can be constant or variable (dynamic). Options of constant lists are stored in the system table fgs_list. A variable list is created from a table rows. List's specifications are stored in the "fgs_list" and options of constant lists are stored in the "fgs_option" system tables.

Examples of a constant list:

- List consisting of the two options: Yes and No

- List consisting of the days of a week
- List consisting of month names

A variable list is based on the rows of a database table. The primary key of the table is used as option value and a text field as option description. To restrict a list's options, you can use a dataset.

List are used to input or output data.

To input data, you can use components of selecting a single option (InputSingleSelect) or multiple options (InputMultipleSelect) of a list.

Lists are used to display a table's field with the ColumnLookup component. For example you can display a company name instead of its code. If a field can have only two values, 0 and 1, you can display Yes or No instead of them.

Usually if a field is entered by selection of one or multiple list's options then the same list is used to display the field.

List's handling depends on a list type – constant or dynamic. Let's consider difference between a constant list and a dynamic list operation on the following example.

Let's suppose that an input form has got an input element of select type. If the list's type is constant, then the **Configurator** creates the element's specification with the list's options. If a list's type is variable. then the **Configurator** creates the element's specification with the list's specification only. The list's options are loaded by the **ListLoader** component at run time when the input element sends the list's attributes to **ListLoader** to load the list. If the list was loaded earlier then the ListLoader returns the list's options. If the list was not loaded then the ListLoader loads the list and returns the list's options.

A list can be used by different components, so such list handling minimizes number of SQL queries to load dynamic lists.

Conditions and statements

Input components have built-in simple validation of input data.

To make more complex validation, the Framework uses conditions and statements. One condition can have a number of statements.

A statement is a logical expression which can be presented as:

<Operand 1> comparison operator < Operand 2>

So to validate input data means to check statements of a condition.

A condition can be used to restrict actions for table rows or displaying certain fields. For example, we can allow updating and removing table rows only for the users with certain roles.

The statement is the evolution of the predicate idea with regard to validation of input data. Statements, like predicates, can also be united in logical groups. Difference between these two notions lies in that predicates' checking is preformed by DBMS and statements' checking is preformed by the components ConditionTester or Validator of the Framework. The ConditionTester is used to check standard statements and Validator for custom ones.

By skillfully combining and using appropriate configuration of statements you can do very complex validation of input data.

Ajax support

The Framework has built-in support of autocomplete input fields, chained select's, inline editing of rows. Also the jQuery Datepicker can be use to input dates. You can use these features without any coding on JavaScript, SQL or PHP.

Let's consider how the Framework does this.

The special jQuery component sends all necessary data in JSON format for JavaScript.

The following files are for Ajax support:

file	comments
fgs.js	Library of JavaScript functions
autocomplete.php	Handler of Ajax request for autocomplete input fields
cell_edit.php	Handler of Ajax request for inline editing of rows
list.php	Handler of Ajax request for chained selects
toggleDebug.php	Handler of Ajax request for On/Off of debug mode
toggleFGS.php	Handler of Ajax request for show/hide the Form, Grid or Search components
toggleMenu.php	Handler of Ajax request for show/hide of a menu

The Framework uses the jQuery Framework.

The jQuery component adds next JavaScript code for all pages:

$(document).ready(initControls)

The initControls function can be found in the fgs.js file. It executes this sequence of functions:

- initDataPicker to tie the jQuery Datepicker to input fields for dates
- initAutocomplete to tie the jQuery autocomplete to autocomplete input fields
- inlineEdit to tie all necessary JavaScript events to rows' fields for inline editing

The handler of Ajax requests for autocomplete input fields (autocomplete.php) receives system id of an input element as an argument. The handler creates the object using the specification of the input element. Using this object's attributes the ListLoader loads the required list. The handler transforms the list to JSON format and sends it to the client.

The handler of Ajax requests for chained selects (list.php) receives system id of the next select element as an argument. The handler creates the object using the specification of the next select element. The handler gets rendering of the object and sends it to the client.

Handler of Ajax requests for inline editing of rows (cell_edit.php) receives system id of an input column and updated value as arguments. The handler creates the object using the specification of the column. If validation of the updated value is successful then it is stored in database.

Using standard components extensions

During development of a real application you will get problems that cannot be resolved by using standard components of the Framework.

A component's specification can include attributes which indicate using custom components. For example, in a Unit's specification you can set a custom controller, Form, Grid, Search, Dataset and renderers. For a list you can set custom list loader. You can also use custom validators, conditions, predicates etc.

If you need to use a custom renderer it is necessary to configure the «renderer» attribute. The value of this attribute means a custom renderer with a static method «render».

Extension of standard components has to meet some requirements. You can find information about possibility of adding new features and necessary requirements in a component description n this Guide.

The Evaluator component

The Evaluator plays important role and provides other components with access to data which will become known only at run time. These data can be the user's id, role or data entered by the user.

The Evaluator is used to calculate expressions at run time. It has got only the «get» static method which receives an expression to evaluate and an optional parameter as arguments. If you need to call the method with multiple parameters then you should pass them as an array.

An expression that you passed in the method has to be in certain format. The Evaluator treats some symbols as calculation instructions.

If the «#» symbol is the first symbol of an expression then the **eval** function is used to evaluate the expression.

If the «@» symbol is the first symbol of an expression then the expression without this first symbol is considered as a function if such function exists or as name of a class with the «get» static method.

If the «&» symbol is the first symbol of the estimated expression then three following symbols of the expression that starting from 1 are considered as the source array id and the rest of the part of the expression without the beginning and ending whitespaces is considered as name of a variable.

If the first symbol of an expression is not «#», «@» and «&» then the method returns the expression itself.

A source array id may indicate the following array sources:

source array id	source array	variable name is treated as:
ses	$_SESSION	Index of an element of the array
glb	$GLOBALS	Index of an element of the array
srv	$_SERVER	Index of an element of the array
env	$_ENV	Index of an element of the array
req	$_REQUEST	Index of an element of the array
coo	$_COOKIE	Index of an element of the array
rgv	The $globals array of the Registry	Index of an element of the array
arg	Passed array of parameters	Index of an element of the array
usr	Data array of the logged user	Index of an element of the array

The Registry component

The Registry plays important role for the Framework. It provides:

- information exchange among components and in fact is the storage of applications data with global scope
- saving and restoring components' state
- imitation of different session in one browser
- interface to global data of an application
- interface to data of the logged user
- storage of debug information if debug information is set to be stored by the Registry
- forming of mandatory parameters for unambiguously identification of a GET or POST request (fgsw, fgsc, fgsu,fgst)

The Registry stores components' state parameters in the element with index FGS_SESSION_KEY of the $_SESSION array. FGS_SESSION_KEY must not be equal to other session variables.

The Registry uses one more constant FGS_CONTROLLER_KEY and system ids of the created Units must not be equal of this constant. FGS_CONTROLLER_KEY should not be also equal to «controller», «form», «grid», «search», «element», «operator», «connector», «argument», «argumentMin» or «argumentMax».

Let's consider what parameters are and where the Registry stores them.

The current controller id of the current request window:

$_SESSION[FGS_SESSION_KEY][window id][FGS_CONTROLLER_KEY] ['controller']

Token of successful ending of a control script

$_SESSION[FGS_SESSION_KEY][window id][FGS_CONTROLLER_KEY] ['end']

A controller’s parameter:

$_SESSION[FGS_SESSION_KEY][window id] ['controller'] [parameter name]

A Unit’s state parameter:

$_SESSION[FGS_SESSION_KEY][window id] [Unit id] ['controller'] [parameter name]

A Form’s state parameter:

$_SESSION[FGS_SESSION_KEY][window id] [Unit id] [‘form’] [parameter name]

A Form’s input element value:

$_SESSION[FGS_SESSION_KEY][window id] [Unit id] [‘element] [attribute rowid]

A Grid’s state parameter:

$_SESSION[FGS_SESSION_KEY][window id] [Unit id] [‘grid’] [parameter name]

A Search’s state parameter:

$_SESSION[FGS_SESSION_KEY][window id] [Unit id] [‘search’] [parameter name]

A Search’s connector value:

$_SESSION[FGS_SESSION_KEY][window id] [Unit id] [‘connector’] [attribute rowid]

A Search’s operators values:

$_SESSION[FGS_SESSION_KEY][window id] [Unit id] [‘operator’] [attribute rowid]

A Search’s arguments values:

$_SESSION[FGS_SESSION_KEY][window id] [Unit id] [‘argument’] [attribute rowid]

A Search’s argumentMin of the «range» predicate value:

$_SESSION[FGS_SESSION_KEY][window id] [Unit id] [‘argumentMin’] [attribute rowid]

A Search’s argumentMax of the «range» predicate value:

$_SESSION[FGS_SESSION_KEY][window id] [Unit id] [‘argumentMax’] [attribute rowid]

The Registry is coded by using Singleton design pattern.

Properties of the Registry:

property	comments
window	Window id of the current request
store	Multifunctional array for date exchange between components
globals	Array for storing global variables of an application
vars	Array for storing debug variables' values
requests	Array for storing the RequestHandler and the EventHandler methods' calls for debugging
sql	Array for storing SQL queries' result for debugging
class	Array for storing classes initialization for debugging
exportErrors	Array for storing specifications' errors during exporting of components' specifications

Methods of the Registry

method	Comments
getInstance	Getting of the only instance of the class
__construct	Class constructor must be of private type
setGlobal($var,$value)	Method to store value ($value) of the $var variable in the «globals» array of the component
getGlobal($var)	Method to return value of the $var variable from the «globals» array of the component
getController()	Method to set the current controller id
newController()	Method to determine if the controller runs the first time
getWindow()	Method to set the current request window id
set($unit, $component,$variable,$value)	Method to store value ($value) of the $var variable related to the $component component of the $unit Unit
get($unit, $component,$variable)	Method to return value of the $var variable related to the $component component of the $unit Unit
saveState($obj,$component,$unit=null)	Method to save state of the $obj object with $component type from the $unit Unit
restoreState(&$obj,$component,$unit=null)	Method to restore state of the $obj object with $component type from the $unit Unit
fgsHidden($unit, $component)	Method to get HTML code for additional hidden fields which are necessary to add to forms of the $component component from the $unit Unit
fgsRef($unit $component)	Method to get additional parameters which are necessary to add to hyperlink of the $component component from the $unit Unit
fgsToggle($unit,$sid $component)	Method to show/hide the $component component from the $unit Unit
checkLogin()	Method of checking user's login
setUser($a)	Method of storing of the logged user's data in session variables
getUser()	Method of getting of the logged user's data from session variables

getUserRole()	Method of getting of the logged user's role
getUserId()	Method of getting of the logged user's id

The Abstract Component

Components Form, Grid, Search and Dataset are descendants of **the Component** abstract class.

Properties of **the Component**

property	comments
$unit	Unit id
$sid	Component id
$type	Type of component
$session	Array of component's properties to save/restore
$initial	associative array of initial values of component's properties
$errors	Array of component's errors at run time
$table	Component's base table
$display	Property to show/hide component

Method of **the Component**

method	comments
__construct($unit,$sid,$hide)	The Component's constructor for the component with id equal $sid from the $unit Unit. $hide is the array of the show/hide parameters
saveState()	Method to save component's state
restoreState()	Method to restore the component's state
hide()	Method to show/hide the component

The Form component

The Form component is an analogue of the HTML element «form» and enables the user to input data in database tables (input form). With an input form you can input data into only one table. The table is the base table of the input form.

The Form is a container of components of two kinds:

- Input elements (input components)
- Buttons

The Form and its components have properties «current input mode» or «mode» and «possible input modes» or modes. For example a standard input form usually can have the following possible modes:

- Mode of inserting of a new row (insert)
- Mode of updating of a row (update)
- Mode of removing a row (remove)
- Mode of viewing or reading a row (Read)

- Mode of seeking of a row (seek)

Mode «seek» is usually used for a site's sign in form when the user types its own login and password.

An input form also can have so called modes for consecutive processing of row set:

- Mode of consecutive updating of a row set (MultiUpdate)
- Mode of consecutive removing of a row set (MultiRemove)
- Mode of consecutive viewing of a row set(MultiRead)

Modes for consecutive processing of a row set arises when the user choose rows and an action for the chosen rows in the Grid component of the common Unit. The user can consecutively do editing or removing of chosen rows.

It's necessary note that updating modes can be more than one. It can be useful for the user's table. We can do updating the user's all data excluding its login and password in the «update» mode and do updating only its login and password in the «update_partial» special mode.

Possible input modes permits to have different set of input elements and buttons for inserting, updating, removing or viewing a row and use for it only one specification of the **Form**.

Also the concept and property of starting mode are introduced for input forms. The starting mode is set at a form's initialization. The concept is introduced for input forms used only for one kind of data processing (updating, removing, search etc).

The «session» property of an input form depends on its current mode.

The **Form** component executes cascade removing i.e. with removing of a master table row it removes also the correspondent rows of all detail tables.

After changing a row's primary key the **Form** component updates all correspondent foreign keys of dependent tables (cascade updating).

The **Form** can have triggers i.e. actions executed before or after actions:

Trigger before insert
Trigger after insert
Trigger before update
Trigger after update
Trigger before remove
Trigger after remove
Trigger before seek
Trigger after seek

The standard **Form** component has got only mock functions instead of real triggers. So if you need to use triggers you have to develop an extension of the Form component and make appropriate configuration of the Unit that will use this custom form component.

Sometimes we need to join tables in SQL query for loading a row. For this purpose, there is the pseudo component ElementTableJoiner in the Framework. The Configurator only uses the ElementTableJoiner to set the «joins» property of the Form. The Form uses the «joins» property to make tables' join for the SQL query. The tables' join type is always «left join». An example of using tables' join is described in the «Configurator guide» document.

A Form component is a part of a Unit component. The Form can interact only with the Unit itself and the Grid of this Unit. The Form changes its input mode in accordance with an event generated by the Grid of the common Unit. The event includes the array of primary keys of rows to process. After completion of an action the Form generates the «done» event and the Unit handles it.

In interfaces MasterDetail and UnitSet the master Grid can generate the «detail» event and the detail Form and detail Grid are the handlers of it. The detail Form can add new rows only if it configured with the special pseudo component InputForeignKey. The InputForeignKey is exported as the InputSystem component with the «foreign_key» property.

In accordance with accepted request processing algorithm the Form rendering must include adding of hidden fields fgsw, fgsc, fgsu and fgst. To get HTML code of these fields, you have to call the method fgsHidden of the Registry with the Unit id and value «form» as arguments.

Buttons of the Form

A button of an input form is an InputButton component. The main attributes of the InputButton are the following ones:

Field	Label	Comments
element_action	Action	Action to be performed by pressing on a button
element_event	Event	Event generated by the Form after completion of an action
element_confirm	Confirm?	Sign to confirm pressing on a button
element_name	Attribute name	Attribute «name» of a button
element_value	Value	Attribute «value» of a button

The Form registers actions of its InputButton components as possible requests.

Input elements of the Form

The following components are used for input elements of the Form

component	Comments
InputAutocomplete	for autocomplete input fields
InputDate	For input of dates
InputDatetime	For input of date&&time
InputFile	For files uploading
InputHidden	for hidden fields
InputMultipleSelect	For input with multiple select
InputSecret	For input of passwords
InputSingleSelect	For input with single select

InputSystem	For input of system fields
InputText	For input field of the «text» type
InputTextarea	For input with textarea
InputTime	For input of times
SelectBooleanCheckbox	For input of Boolean true/false

All components are descendant of the InputElement abstract class.

Input elements' values are stored in session variables.

Some input components have internal storage format for its values. After getting its value from database such component decodes it to its internal format. The Framework makes coding from internal to storing format of these components' values. These coding and decoding are performed by the current database driver.

The InputMultipleAutocomplete is used to input several values delimited with a token and the InputAutocomplete to input only one value with a autocomplete input field.

The InputDate, InputDatetime and InputTime components have an associative array of date and/or time parts (year, month, day, hour, minute and second) as internal format.

Sometimes an input component's value has to be placed within global scope. For this purpose, the input component has got the «register» property. Such components store its values in the «globals» array of the Registry with the index equal to value of the «register» property.

Under the InputMultipleSelect there are hidden three components: SelectManyCheckbox, SelectManyListbox and SelectManyMenu. From processing point of view they are the same and differ only by HTML code i.e. rendering. Choice of one of these components and its configuring depends only on input form's requirements. However, the Form creates all these components as the InputMultipleSelect component with the appropriate renderer. The InputMultipleSelect component has got an array of selected options as internal storage format of its value.
Also under the InputSingleSelect components are hidden three components: SelectOneListbox, SelectOneMenu and SelectOneRadio. Choice of one of these components and its configuring depends only from input form requirements. However the Form creates all these components as the InputSingleSelect component with the appropriate renderer.

The Framework uses MD5 hash of the InputSecret component value instead of its value to save in database.

The InputFile component serves to upload files to server. Main mandatory attributes of this component are:

- Path for loaded files
- Maximum of file size for loaded files
- Possible extensions for loaded files

The InputFile component performs files loading as follows:

- A file to be loaded is stored in an appropriate directory
- A file's name is stored in a table field without any change
- If an input field is mandatory and if a file to be loaded was not executed or failed to be then in the «insert» mode an error is fixed but in the «update» mode is not.

Because files' loading can be accompanied by other actions, the InputFile component has got the optional attribute «extension» for a custom component. Besides, you can use triggers of the Form component to perform required actions.

The InputSystem component is used to input data in special (system) fields of tables.

Examples of such fields:

- id of the current user who updated the current row
- date and time of adding the current row
- IP-address of the current computer from which the site was accessed

Once more application of the InputSystem component is input of foreign keys for detail tables. Note that it's necessary only for the detail Form.

Input elements can be united in groups by setting the «fieldset» property.

Input components have got built-in simple validation of input values:

-if non empty value was entered for required fields
-checking correct format of entered values for fields with date and time types
-if selected options are valid options for input components as InputMultipleSelect or InputSingleSelect

There are validators to perform more complex validation in the Form. Usually a standard validator's specification includes attributes:

- Field to be checked
- Condition for input value of the field
- Condition's parameter
- Condition parameter's type

Condition parameter's type can have one of these values, i.e. «scalar value», «input element» or «array». Condition's parameter with the «scalar value» type can be known at configuration time or calculated only at run time.

Using conditions with two or more parameters requires using the «array» type as parameter's type. For example, the condition «input value must be within a range» requires two parameters: minimal and maximum limit of the range.

Using the «input element» type as parameter's type makes it possible to compare values of two input fields. For example, you can check if values of the fields «password» and «confirm password» are equal.

Checking standard conditions is performed by the ConditionTester component.

If a validator uses a nonstandard condition then it's necessary to set two additional attributes:

- class of the nonstandard condition
- type of the class method to check: static or non static

The class method to check must have name «test».

Checking nonstandard conditions is performed by the Validator component.

An input field can have any number of validators. The order of the validators' checking is set at its configuration.

The Framework has set of standard conditions

Condition ID	comments
AlphabeticDigit	Value must include only letters and numbers
Digits	Value must include only numbers
EmailAddress	Value must be a valid Email address
Equal	Value must be equal with some value
Float	Value must be number of float type
Greater	Value must be greater than certain value
GreaterOrEqual	Value must be greater or equal to certain value
InArray	Value must be an array element
Integer	Value must be a number of integer type
Less	Value must be less than some value
LessOrEqual	Value must be less or equal to certain value
NotInArray	Value must not be an array element
OutOfRange	value must be out of a range
Range	value must be within a range
Regex	Value must include a pattern
StringLength	Length of the value must be within a range
URL	Value must be a valid URL

To make validation of input data, the Form component calls the "validate" validation method of its elements in a loop with the formValue associative array as an argument. The formValue is the array of the Form input components' values plus elements for attributes:

- The Form system id with «xxx_sid» as index of the element
- The Form mode «xxx_mode» as index of the element

These indexes are chosen to avoid coinciding with input fields' names.

An input component performs loop of checking its validators in turn. If a validator is nonstandard then the «test» method of the Validator is called and, otherwise, the «test» method of the ConditionTester if a validator is standard. The «test» method of the Validator receives the

validator's specification and the formValue array. The «test» method of the ConditionTester receives the validator's array of statements specification and the formValue array.

There are input filters to filter input data in the Framework.

Main attributes of a filter's specification are:

- Field to be filtered
- Converter of a field value
- Parameters of filtering
- Type of parameter of filtering

As for validators, you can chose one of these values «scalar value», «input element» or «array» as a type of parameter of filtering.

It's necessary to note that a converter is also a name of a filter's class. It means that any converter corresponds to its own class.

The Framework includes set of standard input filters

Filter	Comments
FilterAlphabeticDigit	Filter removes all symbols excluding letters and numbers
FilterDigits	Filter removes all symbols excluding numbers

If an input filter has to be used after checking of a condition then the filter's specification has to have attributes as such for validators:

- field to be checked
- Condition to be met by input value of the field
- Condition's parameters
- Condition parameter's type

The Framework has the FgsFormView component to render input forms and the FormKit component to render input components.

To use a custom renderer, it's necessary to set name of the custom renderer for input form renderer in Unit's specification.

To use custom rendering of an input element, it's necessary to set the "renderer" attribute in the element's specification. The value of the "renderer" attribute must be a name of the class that performs HTML coding of the element with its static method «render».

Apart from the properties and methods inherited from the Component abstract class, the Form has its own properties and methods.

Properties of the Form

property	comments

e	Array of input elements
b	Array of buttons
mode	Current input mode
apk	Array of primary keys of processing rows
foreign_key	Foreign key value for detail table
primary_key	Primary key of the Form's base table
select	Clause select of SQL query for a row retrieval
from	Clause from of SQL query for a row retrieval
irow	Index of the current row in the "apk" array
err	Array of input errors
query	SQL query for a row retrieval
row	Array of the current row data
aSession	Array of the Form state's parameters that depends on input mode
reference	Array of fields that are foreign keys corresponding to the Form base table's primary key
descendant	Array of fields that are foreign keys corresponding to the Form base table's primary key and the Form base table's type equal to "master table"
dependent	Array of fields that are foreign keys corresponding to the Form base table's primary key and the Form base table's type equal to "reference table"
eventTrigger	Array of events' triggers
failure	Sign of input error
affected_rows	Number of affected rows
formValue	Array of input components' values
type	The Form component's type. It is equal to 'form'

The Form uses the "reference", " descendant" ," dependent" attributes to provide referential integrity for database management systems (DBMS) which does not support this type of data validation before it is entered into the database.

If a value of a table's primary key changed then all correspondent foreign keys have to be changed too. If the value of a Form's base's table primary key changed then the "reference" attribute is used to cascade updating of all correspondent foreign keys.

If a row of master table deleted then all correspondent rows of the detail table have to be deleted too. If of a Form's base's table is the master table then if a row of this table deleted then the "descendant" attribute is used to cascade deleting of correspondent rows of detail tables.

Before removing a row of a reference table, there have to be made checking existence of rows, which refer to the removing row. If such rows were found then removing has to be cancelled. If of a Form's base's table is a reference table then if a row of this table is to be deleted then the " dependent" attribute is used to check existence of rows, which refer to the removing row.

Cascade updating and deleting and checking existence is made by the DBMS driver. If DBMS supports referential integrity then correspondent methods of the driver do not have to execute these operations.

Main methods of the Form

method	Comments
__construct($unit,$sid,$hide)	Class constructor $unit –Unit id $sid- input form id $hide – parameter of show/hide logic
initState($s=null)	Method to set initiate properties' values $s – attribute show/hide and its initial value
saveState()	Method to save the Form's state
restoreState()	Method to restore the Form's state
replaceConfig($sid)	Method to replace the Form's specification by specification with the $sid id
row($ipk)	Method to load a row with the $ipk index
go($args)	Method to handle a request to process a row from a row set $args – parameters of a row to process
processEvent($event,$args=null)	Method to process events $event – parameters of a event to process $args – additional parameters of a event
subscribe()	Method to register internal possible request and events
processRequest($args)	Method to process requests $args – parameters of a request
fire($obj)	Method to generate an event $obj- an instance of the InputButton component
initElements()	Method to initiate input components' values
validate()	Method to validate input components' values
ApplyRequestValues()	Applying of input values
seek($args=null)	Method to process the request to seek a row $args – parameters of the request
insert($args=null)	Method to process the request to insert a row $args- parameters of the request
update($args=null)	Method to process the request to update a row $args- parameters of the request
remove($args=null)	Method to process the request to remove a row $args- parameters of the request
multi_remove($args=null)	Method to process the request to remove a row set $args- parameters of the request
hasReferences($i)	Method to check existence of rows which refer to removing row with index $i
removeDescendants($i)	Method to remove rows of detail tables which correspond to a removing row of a master table with the $i index of the removing row
setQuery()	Method to construct a SQL query for a row retrieval
before_seek()	Method to be performed before processing of the request to a row seeking
before_insert()	Method to be performed before processing of the request to a row inserting
before_update()	Method to be performed before processing of the request to a row updating

before_remove()	Method to be performed before processing of the request to a row removing
after_seek()	Method to be performed after processing of the request to a row seeking
after_insert()	Method to be performed after processing of the request to a row inserting
after_update()	Method to be performed after processing of the request to a row updating
after_remove()	Method to be performed after processing of the request to a row removing

The Grid component

The **Grid** component has HTML element «table» as analog and is used mainly to display database table rows in HTML table. The database table is called the base table. The **Grid** component let the user to navigate through rows, change order and direction of rows and choose actions for a single row or a set of selected rows.

In common case, the Grid components consists of columns to display or input fields, columns of actions for a single row, a column to select rows via marking a checkbox and action buttons on selected rows. There are pages' hyperlinks, input fields to change position within the set of selected rows for the first retrieving row and quantity of rows per page

The **Grid** component can have components of 6 types:

- Component to display or to input field's value (**Column**)
- Component to choose actions for a single row (**RowAction**)
- Component to choose actions for a set of selected rows (**RowSetAction**)
- Component to choose a row (**RowSelector**)
- Component to input a field's value (**InputColumn**)
- Button (**GridButton**)

To simplify configuration of displaying data, there are following Column components in the Framework:

component	Comments
ColumnText	To output a field's value without changing
ColumnDate	To output values of the "date" type in local format
ColumnDatetime	To output values of the "datetime" type in local format
ColumnLink	To output a field's value as a hyperlink
ColumnLookup	To output a field's value with a list

The **Confugurator** exports all these components into the same component the **Column** with appropriate renderer.

Components for fields output or input

The **ColumnLink** component must always have the “renderer” attribute. This attribute must be equal to a name of the class with the «render» static method which constructs the hyperlink.

The **RowSelector** component is rendered as checkboxes (one for a row) and is intended to choose a table row. Updating, removing or viewing of selected rows can be done by checking these rows’ checkboxes and clicking by mouse on an appropriate **RowSetAction** component.

The **RowAction** component is usually rendered as an icon - hyperlink which has the “key” and action name as its parameters. The “key” parameter has to be equal to the row primary key and action name to the row number.

If a primary key consists of several fields then these fields’ values are concatenated with underscore.

The **RowSetAction** component is rendered as a submit button of the HTML form, which includes also checkboxes of the **RowSelector** component.

The **RowAction** or **RowSetAction** component can have as an attribute a condition which validates of the component action. To check the condition the Grid component uses the ConditionTester or Valdator for non – standard conditions. The ConditionTester or Valdator receives the condition’s parameters and the tested row’s data.

The **RowAction** and **RowSetAction** components are used to send selected action and parameters of selected rows to the Form component. However, it’s possible to configure these components to perform actions by the Grid component itself. For example, it’s possible to perform removing rows by the Form component or by the Grid component. In first case the Form displays a removing row data and the user can remove the row by pressing on the button “Remove”. In second case, the user clicks on a row’s icon “Remove” and this row’s removing executes straight away.

For setting any number of displaying rows on a page and any starting number of rows an auxiliary HTML form is used.

In accordance with accepted request processing algorithm, all HTML forms and hyperlinks related to the **Grid** component must send fgsc, fgsu, fgst variables as parameters. The fgsw variable is an optional parameter and it should be sent only if necessary. To get HTML code of these parameters for a HTML form, you have to call the method fgsHidden of the Registry with the common **Unit** id and value «grid» as arguments. To get HTML code of these fields for a hyperlink, you have to call the “fgsRef” method of the **Registry** with the same arguments.

The **Grid** component registers **RowAction**, **RowSetAction**, **GridButton** components as it’s possible request actions and registers requests to change attributes offset, pagesize, order and direction.

Apart from displaying rows, the Grid component can be used to perform rows’ editing of two types.

To perform editing rows of the first type, you have to double click by mouse on desired field of a desired row. Such editing is known as inline editing. In inline editing saving of updated data is performed after pressing key Enter in a input field with type “text” or after selecting an option in input field of “select” type.

Editing of the second type does not require any action from the user. The user can edit data in a spreadsheet-like format. To save updated data the user has to press on a button of the Grid component (GridButton).

The Grid is a part of a Unit component. Usually after the user chose an action on a single row or a set of rows the Grid component generates the appropriate event. The detail Grid is one of handlers of the «detail» event generated by the master Grid. The Grid component also processes the «search» event generated by the Search component after setting a new or dropping a row filter.

To load displaying rows, the Grid component uses row filters, which are set both by the Dataset and the Search component.

Sometimes a SQL query has to select data from several tables by using tables' join. For this purpose, there is the pseudo component ColumnTableJoiner in the Framework. The Configurator only uses this component to set the property «joins» of the Grid. In turn the Grid component uses the «joins» property to do tables' join for the target SQL query. Unlike of the Form component, it's necessary to set a type of table join in a ColumnTableJoiner's specification.

Usually, to display fields with the ColumnLookup using a dynamic list component, the Grid component does left join with the list's table. But if there are reasons not to do so then you can prohibit the table join change for the Grid component.

The Grid component permits forming multirows headers. To do this, you should configure the «Headerspan» property of the Grid component and the «Header» and the «Span» properties of its columns.

To get sum of columns' values and to display columns' totals, it's necessary to set the "Calculate" property of columns during configuration of them.

Rows' sorting can be done by fields of the base or non-base tables. Besides, sorting can be done with a permanent part, which is added at the beginning or ending part of the clause "order by" in a SQL query for rows' retrieval.

Apart from properties and methods inherited from the **Component,** the Grid has got its own properties and methods.

Properties related to the Grid component's state

property	Comments
pagesize	Number of displaying rows per page
direction	Current direction of rows' sorting
order	Current value of sorting
offset	Current position within the set of selected rows for the first retrieving row
hide	Sign to hide the Grid after the user chooses an action on a single row to be performed by the Form component
multimode_hide	Sign to hide the Grid after the user choose an action on a set of rows to be performed by the Form component
display	Sign to display the Grid

apk	Array of primary keys' values for displayed rows
adk	Array of additional fields' values keys for displayed rows
foreign_key	Foreign key's value for a detail table

Using an array of primary keys' values for displayed rows permits the user to do an action only on the displayed rows.

To make the Grid to add a column field's value into an array of additional fields' values keys for displayed rows, it's necessary to set the "Save" attribute of the column. The "adk" array is necessary to send additional parameters of events generated by the Grid component.

Foreign key's value for a detail table is necessary to add the following condition to a SQL query for retrieval of displaying rows:

Foreign key of the detail table=primary key of the master table.

Properties not related to the Grid component's state

property	comments
c	Collection of the Column components
ra	Collection of the RowAction components and компонент
rsa	Collection of the RowSetAction components
b	Collection of the GridButton components
nrows	Number of displaying rows on the current page
total	Total number of rows to display
query	SQL query for retrieval of displaying rows on the current page
set	Collection of displaying rows in the current page
err	Collection of input errors
editable	Sign that there are editable columns in a Grid
type	Type of the Grid component. It is equal to 'grid'

Methods of the Grid component

method	comments
__construct($unit,$sid,$hide)	Class constructor $unit –Unit id $sid- input form id $hide – parameter of show/hide logic
initState($s=null)	Method to initiate properties' values $s – attribute show/hide and its initial values
saveState()	Method to save the Grid's state
restoreState()	Method to restore the Grid's state
row($irow=0)	Mthod to load the row with the $ipk index in the $apk array
page($saveKeyes=false,$rowPK=null)	Method to load a set of displaying rows on the current page $saveKeyes – sign to create arrays $apk and $adk $rowPK – condition for primary key's value
saveKeyes($setGlobal=false)	Method to create arrays $apk and $adk for displaying rows on the current page

	$setGlobal- sign to store some fields' values into the "globals" array of the Registry component
processEvent($event,$args=null)	Method to process events $event – parameters of a event to process $args – additional parameters of a event
subscribe()	Method to register internal possible request and events
processRequest($args)	Method to process requests $args – parameters of a request
RowAction($args)	Method to process a requests generated by the RowAction component $args – parameters of a request
isValidAction($args)	Validation method of a requests generated by the RowAction component $args – parameters of a request
RowSetAction($args)	Method to process a requests generated by the RowSetAction component $args – parameters of a request
isValidRowSetAction($args)	Validation method of a requests generated by the RowSetAction component $args – parameters of a request
setQuery($search_filter)	Method of creation a SQL query for retrieval of displaying rows on the current page $search_filter- sign of a row filter
setOffset($args)	Method to process a requests to set the «offset" attribute $args – parameters of a request
setPagesize($args)	Method to process a requests to set the «pagesize" attribute $args – parameters of a request
setOrder($args)	Method to process a requests to set the «order" attribute $args – parameters of a request
getOrder()	Method of creation the clause "order by" of a SQL query for retrieval of displaying rows
save()	Method to process a requests to save updated data

The Search component

The Search component is an analog of HTML element «form» and enables the user to input row filters for the Grid component. Such form is called a search form. A row filter itself can consists of several search conditions or predicates for fields of different tables, one of which is the base table of the Search component.

In common case a predicate consists of a connector, a comparison operator and an argument. Structure of a predicate is as follows:

<Field of a database table> <comparison operator> <argument>

However, it's possible to create a predicate of the type:

<SQL function (field 1 + field 2)> <comparison operator> <argument>

A connector is used to connect a predicate with a previous one:

<Predicate> <connector> <predicate>

All comparison operators can be divided into 5 groups:

- Comparison test
- Pattern matching test
- Range test
- Set inclusion test
- NULL TEST

The Search component is a container of two kinds of components:

- Button
- Components to input predicates

Buttons of the Search component

A button of the Search component corresponds to an InputButton component.

The Search component registers actions of its InputButton components as possible request.

Components to input predicates can be of two kinds: Predicate and PredicateRange.

The Predicate component serves to input predicates with all comparison operators excluding those of the group “Range test” and is a container of three input elements:

An input element of a predicate’s connector
An input element of a predicate’s comparison operator
An input element of a predicate’s argument

The PredicateRange component serves to input predicates with comparison operators of the group “Range test” and is a container of four input elements:

An input element of a predicate’s connector
An input element of a predicate’s comparison operator
An input element of a predicate range bottom limit or ArgumentMin
An input element of a predicate range top limit or ArgumentMax

An input element of a predicate’s connector is a SelectOneRadio input component and can have only two values «AND» or «OR»

An input element of a predicate’s comparison operator is a SelectOneMenu input component.

If a predicate uses comparison operators of the group “NULL TEST” then the input element of the predicate’s argument is not used.

Values of input elements of a predicate’s connector, comparison operator, argument are stored in session variables.

Predicate components can be united in groups by setting the «fieldset» property. In such case, the connector of the first predicate component in the group serves as the group connector with the previous predicate component or the previous group of predicate components.

The Search component permits to construct predicates with SQL functions.

The Search component uses the PredicateBuilder component to construct SQL code of row filters from data entered by the user.

To use nonstandard method of a predicate’s constructing, it’s necessary to set the custom component’s name as value of the “Custom predicate” attribute in specification of the predicate. The custom component must have the “getPredicate” static method which constructs SQL code of the predicate.

The Search component is a part of a Unit component. The Search component generates the «search» event after setting a new row filter or dropping a row filter. The Search component stores a row filter in the Registry just after setting a new row filter or after restoring its state.

In accordance with accepted request processing algorithm, the Search component’s rendering must include adding hidden fields fgsw, fgsc, fgsu and fgst. To get HTML code of these fields, you have to call the “fgsHidden” method of the Registry with the Unit id and value « search» as arguments.

Apart from properties and methods inherited from the **Component** abstract class the Search has got its own properties and methods.

Properties of the Search

Property	comments
P	Collection of predicate components
B	Collection of buttons
aliases	Collection of field’s aliases
formValue	Collection of predicate arguments’ values
type	The Search component’s type. It is equal to “search”

Methods of the Search component

метод	Comments
__construct($unit,$sid,$hide)	Class constructor $unit –Unit id $sid- input form id $hide – parameter of show/hide logic
initState($s=null)	Method to initiate properties’ values

	$s – attribute show/hide and its initial value
saveState()	Method to save the Search's state
restoreState()	Method to restore the Search's state
processEvent($event,$args=null)	Method to process events $event – parameters of a event to process $args – additional parameters of a event
subscribe()	Method to register internal possible request and events
processRequest($args)	Method to process requests $args – parameters of a request
setRegistry()	Storing a row filter in the Registry component
formFilter($having)	Construction of SQL code from predicate components for clause "where" or "having" which depends of $having argument
getFieldsetPredicate($aSet)	Construction SQL code for a group of predicates
validate()	Method to validate input components' values
validInput()	Method to check input components' values. This method is needed after restoring of the Search component's state
ApplyRequestValues()	Applying input values
setFilter()	Method to process a request to set a new row filter
dropFilter()	Method process a request to drop a row filter

The Search component permits the user to construct very complex row filters.

The Dataset component

The Dataset is nonvisual component and serves to set hidden row filters for Grid components or lists.

Unlike of the Form, Grid and Search component the Dataset component does not have properties to store in session variables.

The Dataset component consists of the predicates which are known or can become known only at run time.

The Dataset uses the PredicateBuilder component to construct SQL code of a row filter using its predicates.

To use nonstandard method of a predicate's constructing, it's necessary to set the custom component's name as value of the "Custom predicate" attribute in the predicate's specification. The custom component must have the "getPredicate" static method which constructs SQL code of the predicate.

The Dataset component stores a row filter in the Registry just after its own creation.

To use a custom class of the Dataset component, it's necessary to set the custom class name in specification of an appropriate Unit component.

Apart from properties inherited from the **Component** abstract class**,** the Dataset has got the $predicates property which is an array of predicates and the only method “setRegistry” to store a row filter in the Registry component.

The Crud, MasterDetail, UnitSet controllers

The **Crud**, **MasterDetail**, **UnitSet** are descendants of the **Controller** abstract controller.

Common properties of controllers

property	Comments
sid	Controller’s system id
title	Controller’s name
initial	Array of initial values of a controller’s properties
session	Array of a controller state’s properties
acl	Access Control List – an array or user roles that have access to a controller
template	Template to render a controller’s page
type	Component’s type. It is equal to 'controller'

The Crud controller has got only one Unit (property”unit”).

The MasterDetail has got the master Unit (property “master”) and the detail Unit (property “detail”). The MasterDetail can be in two modes (property “mode”): master or detail.

The UnitSet controller has got the master Unit (property “master)” and collections of non-master Units (property “units”). The UnitSet can be in several modes (property “mode”): master or a mode equal to a non-master Unit’s system id.

Controller’s method

method	comments
__construct($sid,$config,$template)	Class constructor $sid- controller’s system id $config-controller’s specification $template- template
initState()	Method to initiate properties’ values
saveState()	Method to save a controller’s state
restoreState()	Method to restore a controller’s state
subscribe()	Method to register possible request and events
processRequest($args)	Method to process requests $args – parameters of a request
processEvent($event,$args)	Method to process events $event – parameters of a event to process $args – additional parameters of a event
loadData()	Method to load data for a page rendering
renderJavaScript()	Method to form data to be transferred for JavaScript

The PredicateBuilder component

The PredicateBuilder component serves to construct SQL code of a row filter from received predicates.

The Search, Dataset and ListLoader components use the PredicateBuilder to construct SQL code of a row filter.

The PredicateBuilder can return:
-empty string that means that there is no any row filter
- SQL code of a row filter
- The string «empty» means that the row filter is evaluated to false or, in other words, that there are no rows to satisfy the row filter.

There can be used SQL functions in predicates. If a SQL function is aggregate then SQL code of the predicate is a part of the clause “having” of the SQL query. If a SQL function is not aggregate then SQL code of the predicate is a part of the clause “where” of the SQL query.
So the PredicateBuilder returns SQL code of a row filter as an associative array of SQL code for the clauses “having” and “where” of the SQL query.

The PredicateBuilder constructs SQL code for the clause “having” and for the clause “where” of the SQL query with the same “getFilter” method.

The PredicateBuilder has got only static methods and does not have properties.

The PredicateBuilder’s methods

method	comments
getFilter($predicates)	Method to construct SQL code of a row filter from the $predicates received array of predicates
formFilter($predicates,$having)	Method to construct SQL code for the clause “having” or “where» of a SQL query from the $predicates received array of predicates and the $having sign of a clause type
getFieldsetPredicate($aSet)	Method to construct common SQL code from a group of predicates
getPredicate($a)	Method to construct SQL code from the $a received predicate

Helper components

There are helper components, the **debug** and **Timer,** in the Framework.

The **debug** component serves to collect and display the following debug information:

- Values of debug variables
- Results of SQL queries’ execution
- Calls of the RequestHandler and EventHandler methods
- Initiated classes

- Logging of specifications' exporting

Depending on settings, the **debug** component stores debug information in the "fgs_debug" system table of in arrays of the Registry component.

The debug component has got only static methods. Each kind of debug information is stored by a separate method of the Registry.

method	comments
Write	To store values of a debug variable parameters $value –value of a debug variable $var –name of a debug variable $point – optional point of debugging $error – optional sign of an error to mark it during output
writeSql	To store result of a SQL query's execution parameters $q –SQL query $result – result of execution $type – type of a SQL query
writeListeners	To store a call of the RequestHandler or EventHandler method parameters $args –received array of arguments $type – type of a call
writeClass	To store a initiated class parameters $class –class name $path – path to a class's file
writeExport	To store an error in a specification parameters $component – type of a exporting component $sid – system id of a exporting component $error – an error code код ошибки экспорта $detail – optional type of child element $detail_sid optional code of a child element
render	Method of debug information's rendering

To use the **debug** component debug mode has to be set.

The **Timer** component is developed with using Singleton design pattern and serves for time profiling of scripts' execution. To get statistics of SQL queries's execution, all methods of the database driver that manipulates data has to include program code of storing methods' execution time.

method	comments
start	To start the Timer parameter

	$point–abbreviation of the starting point
mark	Adding of a profiling point parameter $point– abbreviation of a profiling point
addSql	Adding time of a SQL query's execution parameter $time – time of a SQL query's execution in microseconds
render	Method of time profiling rendering

The Framework's templates

There are three templates in the Framework:

admin_crud.php- the template for Crud controllers
admin_md.php- the template for MasterDeatil controllers
admin_universal.php- the template for UnitSet controllers

These templates have to be used with the application.php control scrip. These templates use the **svm** class to create a two level vertical menu.