ABGrid Engine

Declarative WEB-component
for dealing with complex tabular data

Developer's Guide
Version 1.0
Component version 1.0

Russia, Republic of Buryatia

Contents

Introduction

ABGrid Engine designed to work with complex tabular data in WEB-applications. The basic architecture of the component dates back to 2013, when the author based on the well-known tabular jQueryThe - plugin was developed as a grid component for the language's visual programming environment PHP. During development, the difficulties and need to write monotonous code for use in master part modes, in the mode of displaying some additional data in the same table below the main data line, and some other problems emerged, which prompted the creation of an ultimately simple and logical architecture component that does not require manual repetition of monotonous code. Subsequently, it was decided to create our own tabular one jQueryA plugin that was implemented but used only for its own needs. At the end of 2025, a decision was made to rebrand and translate the component into pure language Javascript and CSS, which was done. Currently ABGrid Engine self-sufficient and does not have any dependencies on external libraries. The product received many new modules for the convenience of developers, such as an internal record editor autocomplete and the ability to connect it to custom ones input, possibility to work only as View, the developer’s use of system dialogues and notifications, workspace management, and much more, but the approach remains the same - at least manual code writing by the end user-programmer.

Author

1. Key features of the product

ABGrid Engine <TAG1> it's universal grida component focused on developing complex business-applications, an engine with a high degree of configurability and minimal user code. The component is built on the basis of a declarative approach to solving complex problems. Most tasks within the framework ABGrid Engine it is solved at the configuration task level and does not require writing additional code.

Key idea ABGrid Engine:
«Complex — simple. You make a configuration — ABGrid Engine everything else».

1.1 What decides ABGrid Engine

ABGrid Engine designed to solve typical and complex problems of working with tabular data:

- downloading data from the server
- display and format strings and cells
- editing data (create / update) by a built-in editor
- customizable validation of editable data to minimize non-crippled server traffic
- deleting entries
- page-by-page server navigation
- working with detailed grids and subgrids in automatic mode
- display tree data structures in automatic mode
- built-in mechanism autocomplete for editing data
- connecting the mechanism autocomplete component to custom input
- central management of access rights and regimes
- managing the space of others ABGrid-of components
- unlimited hierarchy master\detail, subgrid, treegrid and at the same time, the entire process occurs completely automatically.
- provides the user with the ability to use system dialogues in their code
- all system dialogues and notifications ABGrid Engine:
    - isolated by Grid specimens
    - they do not conflict between master/detail/subgrid
    - safe if there are several greades on the page
    - isolated from user code when they are used
- provides the user with the ability to use system message output

1.2 What ABGrid Engine doesn't do it consciously

ABGrid Engine is not:
• ORM-by the system
• replacing server validation
• user and role management system

1.3 Configuration philosophy

IN THE ABGrid Engine no need:
• manually write handlers CRUD-operations
• manage the state of the editing form
• synchronize detailed and subgrid grids manually.
The behavior of a component object is described by the configuration, and all internal state transitions are performed by the engine.

1.4 For which projects is the component intended

ABGrid Engine suitable for:
• administrative panels
• corporate systems
• ERP / CRM decisions
• internal business- interfaces

The component scales from simple tables to complex hierarchical structures with nested grids and detailed tables. All detailed components, etc. subgrids are complete components ABGrid Engine., all relationships between which work automatically based on a given configuration. Detailed tables can have both their own subgrids and their own detailed tables, and there are no restrictions on the level of hierarchy, it all depends on the tasks and imagination of the developer.

1.5 Basic terms

The documentation uses the following terms:

Grid or grid <TAG1> copy ABGrid Engine, related to DOM-container.
Master-grid, master grid — main grid.
Detail-grid <TAG1> detailed grid associated with the selected wizard— grid row.
Subgrid <TAG1> a nested grid that opens inside the string.

TreeGrid-function mode of the component in the display mode of the tree data structures.
Editor <TAG1> record creation or editing form.
Policy <TAG1> access rights policy applied to CRUD-operations.
readOnly <TAG1> grid mode — view only«.

2. Architecture ABGrid Engine

ABGrid Engine built according to modular architecture. The main purpose of such an architecture is — the division of responsibilities between components and the possibility of independent development of each module.

Basic principles of architecture:

• GridCore <TAG1> central grid coordinator
• DataEngine <TAG1> downloading and updating data
• CrudEngine <TAG1> operations create / update / delete
• TreeEngine <TAG1> support for tree structures
• DetailsPanelsEngine <TAG1> detailed panels control
• ToolbarEngine <TAG1> toolbar elements
• EditFormService <TAG1> built-in record editor

GridCore connects all subsystems and provides a single one API for the developer. Each module is responsible only for its own area of functionality.

This architecture allows:
• expand functionality without changing the kernel
• connect new features gradually
• facilitate code testing and support

2.1 Life cycle

The grid works through several stages:

1. Creating a Grid instance
2. Configuration normalization
3. Initializing modules
4. Loading data
5. Rendering of the table
6. Connecting event handlers

This life cycle allows flexible connection of new modules and extensions.

2.2 Role GridCore

GridCore <TAG1> central object of the system.

It performs the following tasks:
• stores configuration options
• manages the state of the grid
• initiates data loading
• manages events
• connects all system modules

Most public methods API they are called through an object grid.

2.3 Configuration parameter schema

Schema defines the grid data structure and is used in several places at once:
• display of columns
• records editor
• data validation
• type conversion
• data display roles

Therefore schema is the central part of the configuration.

2.4 Recommendations for field descriptions

When describing schema recommended:
• use clear field names
• always specify the field type (type)
• use editor only where editing is required
• add hint for tips to the user
• explicitly ask required for required fields

Example:

```
schema: {
  name: {
    type: 'string',
    editor: {type: 'text', requested: true}
```

```
  },
  price: {
    type: 'number',
    editor: {type: 'number', min: 0 }
 }
}
```

2.5 Using modes create / edit

IN THE ABGrid you can set different field settings for create and edit modes.

Example:

```
schema: {
  quantity: {
    type: 'int',
    editor: {
      type: 'int',
      min: 1,
      default: 1,
      create: {visible: true},
      edit: {visible: false}
 }
  }
}
```

This allows flexible management of the editor form behavior.

2.6 Using field roles

Parameter role determines where the field is used.

For example:

```
role: 'grid'
role: 'editor'
role: 'data'
```

This allows:

• hide fields from the table
• use fields only in the editor

• store service data

2.7 General guidelines for circuit design

To keep the grid configuration clear and scalable:

• don't overload schema a lot of logic
• use a single style for describing fields
• group similar fields
• use comments for difficult places
• try to keep the same structure editor for all fields

2.8 Plugin system ABGrid Engine

ABGrid Engine supports functionality expansion via **plugins**.

The plugin — is a regular JavaScript class that accesses a grid instance and can:

- subscribe to events
- add new methods
- interact with data
- change component behavior

Plugins allow you to expand the capabilities of the ABGrid Engine **no change to library source code**.

This means that a developer can create custom extensions with only **collected ABGrid library**.

2.8.1 Plugin connection

The plugin is connected when you create a grid through the parameter `plugins`.

Example:

```
class ExamplePlugin {

    constructor(grid) {
 this.grid = grid;
    }

    init() {
 console.log("ExamplePlugin initialized");
    }

}

const grid = new ABGrid('#grid', {

    plugins: [
 ExamplePlugin
    ]

});
```

When creating the ABGrid:

1. creates an instance of each plugin
2. gives him the object `grid`
3. calls the method `init()`.

2.8.2 Plugin structure

The minimum plugin structure is

```
class MyPlugin {

    constructor(grid) {
 this.grid = grid;
    }

    init() {

 //initialization code

    }
}
```

Element	Appointment
constructor(grid)	receives a copy of the grid
init()	called after initialization of the grid

Through `this.grid` the plugin gains access to all the features of the component.

2.8.3 Use of events

The plugin can subscribe to grid events.

Example:

```
class RowLoggerPlugin {

    constructor(grid) {
 this.grid = grid;
    }

    init() {

 this.grid.on('row:click', ({rowId }) => {
 console.log("Clicked row:", rowId);
        });

    }
}
```

In this way, you can respond to:

- loading data
- selecting strings
- user actions
- CRUD operations

2.8.4 Adding your own API

The plugin can add new methods to the grid instance.

Example:

```
class ExportPlugin {

 constructor(grid) {
 this.grid = grid;
    }

 init() {

 this.grid.exportCSV = () => {

 const rows = this.grid.getRows();

 console.log("Export rows:", rows);

        };
    }
}
```

After that, the method can be used:

```
grid.exportCSV();
```

2.8.5 Multiple plugin usage

You can connect multiple plugins at the same time:

```
const grid = new ABGrid('#grid', {

    plugins: [
 ExamplePlugin,
 RowLoggerPlugin,
 ExportPlugin
    ]

});
```

Plugins are initialized **in the order in which they were connected**.

2.8.6 When to create a plugin

Plugins are recommended for:

- additional interface logic
- integrations with other libraries
- grid API extensions
- automating user actions
- adding new UI modules

This approach allows you to expand the functionality of the component **without changing its source code**.

2.8.7 Recommendations for plugin development

When creating your own plugins for ABGrid Engine, it is recommended to follow several rules to ensure compatibility with future versions of the component and avoid conflicts with other plugins.

Use only public API

The plugin should only communicate with the component via **public Grid methods and events**.

It is not recommended to refer directly to the internal properties of the grid object.

For example:

✓ correct

```
this.grid.on('row:click', handler);
this.grid.getRows();
```

✕ undesirably

```
this.grid._rows
this.grid._dataEngine
```

Internal properties may change between component versions.

Do not change internal data structures

The plugin should not be directly modified:

- internal data sets
- column configuration
- the condition of the grid

All changes must be made through the grid API.

For example:

✓ correct

```
this.grid.reload();
```

✕ undesirably

```
this.grid.data data.rows = [];
```

Use your own namespaces

If the plugin adds methods or properties to the grid object, it is recommended to use unique names.

Example:

✓ recommended

```
this.grid.myPluginExport = function() {... }
```

✕ undesirably

```
this.grid.export = function() {... }
```

This helps avoid conflicts with future versions of ABGrid Engine or other plugins.

Subscribe to events instead of changing behavior

If you need to respond to user actions or data changes, it is better to use an event system.

Example:

```
this.grid.on('data:loaded', () => {
 console.log('Data loaded');
});
```

This approach makes the plugin independent of the internal component implementation.

Avoid change DOM outside the grid container

The plugin should only work inside a DOM container owned by the grid.

It is not recommended to change page elements outside the grid area unless required by application logic.

Minimize dependencies

It is advisable that plugins do not require additional libraries or frameworks.

This makes plugins easier to use and reduces the likelihood of conflicts.

Compatible with future versions

When developing plugins, it is recommended to consider that:

- the internal mechanisms of the ABGrid Engine may change

- the public API remains stable

Therefore, plugins should only rely on **documented capabilities of the component**.

2.8.8 Example of a custom plugin

Below is a simple example of a plugin that adds a new method to the grid API.
The method outputs to the console the number of rows loaded in the table.

This example demonstrates the basic principle of how plugins work — **expanding the capabilities of the Grid instance**.

Plugin implementation

```
class RowsInfoPlugin {

 constructor(grid) {
 this.grid = grid;
    }

 init() {

 this.grid.printRowsCount = () => {

 const rows = this.grid.getRows();
 console.log('Rows count:', rows.length);

        };
    }
}
```

Plugin connection

The plugin is connected when you create a grid through the parameter `plugins`.

```
const grid = new ABGrid('#grid', {

    plugins: [
 RowsInfoPlugin
    ]

});
```

Using the plugin

Once the plugin is connected, the new method is made available through the grid object.

```
grid.printRowsCount();
```

The number of rows loaded into the table will be displayed in the console.

3. Connecting files to HTML-to the page

Recommended connection option:

```
<head>
   <link rel="stylesheet" href="/abgrid-engine/abgrid-structure.css">
  <link rel="stylesheet" href="/abgrid-engine/abgrid-icons.css">
  <link rel="stylesheet" href="/abgrid-engine/themes/abgrid-theme-blue-middle.css">
   <script src="/abgrid-engine/abgrid.min.js" defer></script>
</head>
```

Use of:

```
<script>
 document.addEventListener('DOMContentLoaded', () => {
    const grid = new ABGrid(...);
 });
</script>
```

Example of a minimal connection (ES-modules):

```
<link rel="stylesheet" href="/abgrid-engine/abgrid-structure.css">
<link rel="stylesheet" href="/abgrid-engine/abgrid-icons.css">
<link rel="stylesheet" href="/abgrid-engine/themes/abgrid-theme-blue-middle.css">

<div id="grid"></div>
<script type="module">
   import {ABGrid } from '/abgrid-engine/abgrid.esm.js';

   const grid = new ABGrid('#grid',{
     autoLoad: true,
     data: {url: '/api/items' }
   });
</script>
```

4. Reserved names and service fields

Some of the names are used by the engine for official purposes and are not recommended as custom field names:

- id <TAG1> unique string identifier (primary key), in edit always read-only. Field uniquely identifying the string. If necessary, the server can form this field as a result of concatenation of several primary fields or theirs hash-the result. The component always uses only one field to identify strings.
- __confirmMismatchTpl (message template for confirm-of fields)
- Column type actions (virtual render column)
- policy (in the server's evolution response: data.policy)

5. Data exchange format with the server

ABGrid Engine supports different formats for representing strings when loading and sending data.

- inFormat (data.inFormat):
- auto — define the format automatically
- object — array of objects (recommended)
- aoa <TAG1> Array of Arrays (strings as arrays of values)
- csv — CSV string (delimiter is set to data.csvDelimiter)

outFormat (data.outFormat): object | aoa | csv — Outbound Data Format (CRUD and user requests).

In mode «auto» the input data format is recognized by the first line of the line data.

Exchange formats are specified in the parameter data data. For example,

```
data data:{
        ...
  //string formats
  inFormat: 'auto', //'auto'|'aoa'|'object'|'csv'
 //outgoing data format (CRUD, user requests): 'aoa'|'object'|'csv'
  outFormat: 'object',
     csvDelimiter: ';'
}
```

6. Guidelines for architecture

• Use internal controller for the simple CRUD-of the scripts
• Use external controller for complex forms and business logic
• All business validations must be duplicated on the server
• Do not modify the data grid directly — use API

7. Quick start

The «Quick Start» section demonstrates the minimum set of steps required for initialization and use ABGrid Engine.
The purpose of section — is to show the basic principle of operation of the component without going deeper into advanced settings.

7.1 Minimum initialization

A component instance is created by calling the constructor ABGrid(container, options), where container <TAG1> CSS-selector or DOM-container element. Parameter root inside options not used. It is highly recommended to use the attribute « as a containerid» div-of the element.
To create a grid, just specify DOM-container and configuration.
IN THE ABGrid Engine grid columns are built from schema.fields. It also describes the rules for the editor (required/readOnly by mode, values enum/autocomplete etc.).

Example of minimal initialization:

```
const grid = new ABGrid('#grid', {
 data data: {
 urls: '/api/items'
      },
 schema: {
 fields: {
 id: { type: 'number', ui: { label: 'ID', grid: { visible:
true } } },
 name: { type: 'text',   ui: { label: 'Name', grid: { visible:
true } } }
            },
 order: ['id', 'name']
      }
});
```

7.2 What happens when a grid is created

When creating an instance ABGrid Engine:
• the grid is linked to DOM-to the container
• the internal state is initialized
• data loading mechanisms are being prepared
• request and response interceptors are recorded

The component itself does not send requests to the server until data loading is explicitly called or startup is triggered when the parameter is set autoLoad=true.

7.3 Loading data

A method is used to download data load():
grid.load();

Method load:
• sends a request to the server
• receives data in a standard response format
• updates table content
• sets the current string when data is available
• updates related detailed tables
• sets a policy of rights CRUD if there is permission in the configuration and if there is a policy itself in the server response

7.4 Server response format (briefly)

The server is obliged to transmit the response in format JSON and to comply with the response standard for successful processing by a component, the response must be an object with the following parameters (for example, for a boot operation):

```
{
 success: true | false, //the result of the operation
 message: 'OK',          //message
 data data:{
 rpp:10,   //confirmation requests rpp
 page:1, //confirmation requests page
 totalRecords:97,        //total number of entries
//strings, format is specified by the data.inFormat parameter
 rows: []
   }
}
```

Additional fields (policy (et al) processed automatically if present.

7.5 Minimum life cycle

The type life cycle of a grid is
1. Creating an instance ABGrid Engine
2. The Challenge load()
3. Receiving data from the server
4. Display strings
5. Setting the current line
6. Cleaning and loading of detailed nets

Further actions (editing, detailing, navigation) are described by the configuration and do not require additional code.

8. Field types

IN THE ABGrid Engine field data type**,** field type in editor and field type in grid <TAG1> it is different entities and they ask in the appropriate configuration parameters**.**

8.1 Types fields data (schema.fields.<name>.type)

Defines the type of data.

Type	Appointment	Note:
text	String data	Default
number,int	Numerical data	Reduced to number
bool, boolean	Boolean type	true\false
date	Date	Requires a strict YYYY-MM-DD format
datetime	Date and time	Requires a strict YYYY-MM-DD format HH:mm
json	Arbitrary object	No transformations
password	To hide passwords	ShowPasswordToggle: true option for show\hide

Used **only** for:

- normalization of values
- validations
- exchange formats with the server

✕ **NOT affected** on:

- editor's choice
- display in grid

Important about enum:

- type: "enum" **does NOT select an editor**
- enum <TAG1> it is **type of validation**, not UI
- Valid values are taken from schema.fields.<f>.values.enum

Supported formats values.enum:

1) Object-map (recommended, simplest)

```
values: {
 enum: {
 M: "Male",
 F: "Feminine"
   }
}
```

- key <TAG1> value stored in the data
- value <TAG1> displayed text

2) Array of strings/numbers

```
values: {
 enum: ["M", "F"]
}
```

- both the value and the text displayed = the same
- suitable for simple cases

3) Array of objects {value, text }

```
values: {
 enum: [
      { value: "M", text: "Male" },
 { value: "F", text: "Feminine" }
  ]
}
```

- clear separation value and text
- useful if you need an extensible format

How the system uses it

IN THE editor (editor.type = "select")

- value <TAG1> what is written in the field
- text <TAG1> what the user sees

In the grid

- displayed **text**
- no matter how enum is given

In the validator (type: "enum")

- it is checked that **value included in the list of acceptable ones**

- the comparison is ongoing **by string representation** (even if enum is given by numbers)

In all cases, ABGrid Engine leads enum to a single internal format `{value, text}`.

8.2 Types fields editor (schema.fields.<name>.editor.type)

The field type determines which HTML-control will be created in the built-in editing form. The following are the types supported in the current version.

Type	Appointment
text	<input type="text">
number	<input type="number">
boolean	<input type="checkbox">
date	<input type="date">
datetime	<input type="datetime-local">
email	<input type="email">
file	<input type="file"> Selecting a file to upload to the server
tel	<input type="tel">
textarea	<textarea></textarea>
select	<select> <option value="...">...</option> </select>
autocomplete	<input type="text"> Remote directory search by built-in module Autocomplete
confirm	<input type="text"> Confirmation for the main field, the main field is usually of type string in the schema.fields.<fname>.type

Rules of use:

- Editor **selected by ONLY editor.type**
- values.enum **does not include anything in itself**
- values.autocomplete **does not include anything in itself**
- If editor.type not specified → used text
-

If the field contains a file, the CRUD request is automatically sent as:
multipart/form-data
File transmitted in the field: file_<fieldName>

8.2.1 More about type «select»

It is being created HTML-element «select».

```
<select>
  <option value="...">...</option>
</select>
```

Source data for select — **schema.fields.<field>.values.enum**.

This is a required parameter for editor.type = "select".

```
editor: {type: "select" },
values: {enum:...}
```

Supported formats values.enum

8.2.1.1 Object-map {value: text } (recommended)

```
values: {
 enum: {
 M: "Male",
 F: "Feminine"
  }
}
```

HTML would look like this:

```
<select>
 <option value="M">Male</option>
 <option value="F">Female</option>
</select>
```

- value <TAG1> the value which **saved in the field**
- text <TAG1> text which **the user sees**

✓ Recommended format for documentation and actual projects.

8.2.1.2 Array of objects {value, text }

```
values: {
 enum: [
    { value: "M", text: "Male" },
    { value: "F", text: "Female" }
  ]
}
```

HTML:

```
<select>
 <option value="M">Male</option>
 <option value="F">Male</option>
</select>
```

Used if you need a more extensible format (for example, in the future with additional attributes).

8.2.1.3 Array of strings or numbers

```
values: {
 enum: [1, 2, 3]
}
```

or

```
values: {
 enum: ["A", "B", "C"]
}
```

HTML:

```
<select>
 <option value="1">1</option>
 <option value="2">2</option>
 <option value="3">3</option>
</select>
```

It is important to understand behavior:

- value and text **the same**
- numbers **are given to the string** in HTML
- this is acceptable, but **inconvenient for the user**, if you need human-readable signatures

✓ Suitable for simple technical values
❌ Not recommended for UI with signatures

What happens when you select a value

When selecting an item:

- it is recorded in the field **value**
- the value data type is defined by **schema.fields.<field>.type**

Example:

```
type: "number",
editor: { type: "select" },
values: { enum: [1, 2, 3] }
```

- the user chooses 2
- in the <select> meaning "2" (string)
- the model leads to a value of number
- in the data will be 2 (number)

8.2.1.4 Behaviour in the absence of values.enum

If specified:

```
editor: { type: "select" }
```

but **values.enum missing or empty**, then:

- <select> it will be empty
- no choice possible
- this is considered **configuration error**

It is recommended to explicitly state values.enum always.

8.2.1.5 Relation to validation (values.enum)

If additionally specified, when saving:

- the value is checked for ownership values.enum
- the comparison is performed **by string representation**

doesn't affect on HTML

- **doesn't enable select automatically**
- is used **for validation only**

8.2.2 More about type «autocomplete»

This type of editing in the editor is specified to select values from a remote directory on the server when the previous method becomes extremely inconvenient for the user. As a rule, this happens with a relatively long list of options. In this case, it is more convenient to use this type of editing. The component engine provides a simplified mechanism that provides 2 fields in the data line - one field hidden from the user for an identifier (code), the second is easy for the user to read.

Autocomplete can be specified globally (options.editor.autocomplete) and locally in the field (field.editor.autocomplete). Local setting of a field overrides global for that field only.

The default global configuration is

```
autocomplete: {
 urls: null,
 dataKey: null,          //Query data key for server
 queryParam: 'q',        //search bar
 fieldParam: 'field', //a parameter for specifying a value
search field
 minChars: 3,     //minimum number of characters to trigger the
request
 debounceMs: 250, //protective pause in ms
 limit: 30,          //restricting options (for server)
  strict: true      //prohibit entering values outside the list
 }
```

An example of local configuration:

```
schema: {
 fields: {
 companyId: {
 type: 'text',
 displayField: 'companyName’,
 editor:{
 type: 'autocomplete’,
 autocomplete: {
  enabled: true,
     url: '/api/companies/ac',
          fieldParam: 'id',
     textField: 'name',
  dataKey: 'company'
 }
       }
 }
 }
}
```

Parameter displayField specifies the editor where to write the text value when selecting. If local overrides are not needed, then it is enough to specify displayField and editor:{ type: 'autocomplete' }.

8.2.2.1 Server response format

The server must respond in the following format:

```
{
 success: true,

//at success: true an empty string is allowed
//at the success: false required non-empty parameter to
display to the user
 message: ","
 data: {
 items: [
  {id: 123, value: "OOO Chamomile" },
              ...
 {id: 124, value: "JSC Cornflower" }
 ]
 }
}
```

Important:

- id <TAG1>required (any type but not null/undefined)
- value <TAG1>required (is given to the string)
- any other fields can be added — they are saved as raw, but UI shows value.

If there are no options, the server must install success:false and in message failed search message string (in this case the parameter data data optional), for example

```
{
   success: false,
   message: 'Not really found'
}
```

If the search fails, it is the value of the parameter message displayed to the user and selection is prohibited.

8.2.2.2 Note on autocomplete.dataKey

Parameter dataKey added to the request autocomplete how query-parameter `dataKey`. This allows you to use one endpoint for different sources of hints (for example: companies, users, products).

Example request:
`/api/autocomplete?q=abc&field=companyName&limit=30&dataKey=companies`

8.2.3 More about type «confirm»

ABGrid Engine provides the ability, when editing records by an internal editor in built-in form, to automatically display and process the input value confirmation field in the main field to ensure that the user has actually entered the correct value into the main field and will remember it if necessary. This is used when changing, for example, a password or entering an email value, etc. These fields are of the type type = «confirm». These fields only make sense in the layer UI‘, do not exist in the object of the string and do not participate in any way in the exchange with the server. Fields with type «confirm» inherit the properties of the main field, so in the configuration it is enough to indicate only the unique properties of this particular type of field, namely its type type: 'confirm'and confirmOfп, which must contain the name of the main field, the values of which are confirmed by the « fieldconfirm». When the user enters different values, the engine will not allow saving data from the internal editing form and will display the corresponding message.

Here’s a practical example. In the configuration, we specify 2 fields for changing the password – main field «password» and confirmation field «firmPassword».

Example with a field of type «confirm».

```
schema:{
    fields:{
 password:{
 type: 'text',
            editor: {
                edit: {visible: true, required: true},
                create: {visible: true, required: true},
            }
       },
 passwordConfirm:{
 type: 'confirm',
           firmOf: 'password'
       }
    }
}
```

8.3 Types columns render (fields.<field>.ui.colType)

The type of the render column controls how the value is displayed in the table. It is an independent subsystem from editor.type. The type of a particular column is specified in the property of the field itself in the parameter ui.grid.

Field types:

- text (default)
- tree
- actions (virtual column)
- image
- link
- badge
- select
- autocomplete
- boolean
- progress
- rating
- date
- datetime
- currency

Example of a column render with type boolean:

```
schema:{
      fields:{
 .....
 published:{...., ui:{... grid:{colType: 'boolean'...}}},
 .....
      }
}
```

If the field is set values.enum, ABGrid Engine by default, treats the column as select and displays label instead of code. To show the original value, set ui.grid.colType: "text" or yours ui.grid.renderer.

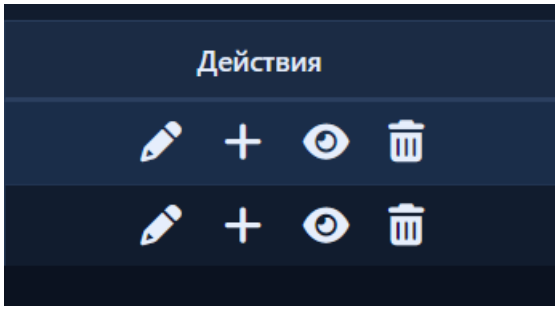
8.3.1 Details of column type «actions»

ABGrid Engine allows you to create custom columns with type colType= «actions». These columns render buttons on each line for some actions on the line, such as editing, deleting, etc. It is important to understand that these are user columns that do not participate in communication with the server, lines do not exist in the object, and when the object is transferred, lines are not transferred to functions. The component engine recognizes them by type colType= «actions» and operates according to its own internal algorithm. The number of columns and their placement order are arbitrary. Actions in the column are rendered as elements «button» according to a certain internal algorithm.

Example of a column with type actions:

```
field_actions:{type:'text',
     ui:{label: appMsg?.['Actions']?? 'Actions',
             grid:{width: '20%', colType: 'actions',
                    actions: {
           edit: {classes: 'fa fa-pencil', 'aria-label':
appMsg?.['EditCategory']?? 'Edit', hint: appMsg?.['Edit']??
'Edit'},
          addsubctg: {classes: 'fa fa-plus', 'aria-label':
appMsg?.['AddSubCategory']?? 'Add subcategory', hint:
appMsg?.['AddSubCategory']?? 'Add subcategory'},
          publish: {classes: 'fa fa-eye', 'aria-label':
appMsg?.['PublishingCategory']?? 'Publish', hint:
appMsg?.['Publish']?? 'Publish'},
          trash: {classes: 'fa fa-trash-can', 'aria-label':
appMsg?.['DeleteCategory']?? 'Delete', hint:
appMsg?.['Delete']?? 'Delete'}
                  }
          }
     }
}
```

Display on page:



The click handler is assigned in software, for example:

```
function onActionClick (action, row, ctx, grid) {
 alert(action);
}
gridUsers.onActionClick(onActionClick);
```

The action name is formed from the first parameter actions.*: {....}, where * is the configuration parameter.
Parameter action <TAG1> this is the property name from the configuration, i.e. (see example above)
«edit», «addsubctg», «publish», «trash».

The rest parameters:

- row <TAG1> the object of the string,
- ctx <TAG1> context
- grid <TAG1> instance of a specific component ABGrid Engine

Properties:

- classes <TAG1> custom icon class to display
- aria-label - Attribute text aria-label action buttons
- hint hovering over the button prompts the text(title)
- raw <TAG1> custom attributes. They rendere as is.

Use the option to specify custom attributes rawп, for example:

```
actions: {
 edit: {
 hint: 'Edit',
 classes: '',
 raw: 'uk-icon="icon: file-edit"'
 },
 delete: {
 hint: 'Delete',
 classes: '',
 raw: 'uk-icon="icon: trash"'
 }
}
```

In this example, instead of icons fontawesome icons are set UIKit.

9. Component configuration

Configuration ABGrid Engine describes the behavior of the grid declaratively. All the main features of the component are configured through the object optionsp, passed to the constructor.

9.1 General configuration structure

Configuration ABGrid Engine consists of several sections. Below is a typical «map» configuration.

```
{
  //Modes
 autoLoad: true,
 readOnly: false,

  //Rights policy (auto-application data data.policy from
server responses)
 policy: { enabled: true },

  //Data and transport
 data data: {... },

  //Final lines
 summary: {... },

// CRUD-engine
 crud: {... },

  //Detailed grids
 detailGrids: [... ],

  //Detailed panels
 detailsPanels: [... ],

 // UI-helpers
 dialogues: true,
 loadingOverlay: true,

 //Localization
 i18n : {....},
```

```
  //Visual settings and UI
  view: {... }

//Editor (global defaults and behavior)
 editor: {... },

  //Data schema and UI (columns, field editor rules)
 schema: { fields: {... } },

}
```

Important:
• Parameter columns the new architecture does not use — columns and editor behavior are described through schema.fields.
• Section view responsible for displaying (heights, tulbar, pager, subGrid et al).

Below is a short parameter assignment:
autoLoad
If true <TAG1> grid automatically calls data loading after initialization. If false <TAG1> data is loaded only by explicit call grid.load(). Detailed meshes are blocked by the master mesh.
detailGrids[]
Array of detailed composites ABGrid (master-detail), which are automatically created and rebooted based on the current wizard line. Each detail is described in terms of { gridId, linkfield, options }, is a full-fledged ABGrid EngineThe - component and is configured exactly the same as the master component.

view
Grid visual configuration: dimensions, title, toolbar, pager, sorting, checkboxes, subgrid, treegrid etc. Section view affects UI, but does not replace permissions checks (they are performed at the time of action).

view.subGrid
Nested grid inside the string (subgrid). Configurable via {enabled, linkfield, options }. options <TAG1> this is the usual grid configuration, i.e. subgrid it is a full copy ABGrid Engine.

9.2 readOnly

Parameter readOnly puts grid into mode «view only».
At readOnly=true all data modification operations are prohibited (create/update/delete), whether the built-in editor form or external controller is used.
ABGrid Engine uses single contract «envelope» (envelope) to download data and CRUD-of operations. The contract can be wrapped in dataKey.

9.3 autoLoad

Parameter autoload controls startup of the component immediately after initialization. For detailed grids, this parameter is forced to be set by the master grid false when creating them. The user may not create detailed grids himself; the component will do this if they are specified.

9.4 Final lines

ABGrid Engine provides the ability to display summary rows at the bottom of the table, which can include information both by page and global totals. Page data is calculated automatically by the component; if this data is not in the server response, global data, naturally, can only be reflected if this data is in the server response. The configuration of the final section is set as follows:

```
summary: {
 enabled: false,
 page: true,
 total: true,
 layout: 'stack',
 pageLabel: 'Σ Page',
 totalLabel: 'Σ Total',
 separator: '',
 emptyText: ''
}
```

Options:

enabled <TAG1> includes the final lines;

page <TAG1> resolves string page-results;

total <TAG1> resolves string total-results;

pageLabel and totalLabel <TAG1> line signatures;

separator <TAG1> separator between multiple units;

emptyText <TAG1> text for empty final cell.

layout - display type. Options: 'stack' – one under the other, 'inline'-in line

The results are set not only globally, but also at the field level through schema.fields.<name>.summary. Options supported:
summary: true
summary: 'sum'
summary: ['sum', 'avg']
summary: { page: ['sum', 'count'], total: ['sum', 'avg', 'max'] }

If summary: true, the set of aggregates is automatically selected: for numeric fields, the default is sum, for non-numerical — count. **Supported units: sum, avg, min, max, count**.

PageThe component calculates the results itself from the lines of the current page. Total-the server can return the results, for example, to data data.summary.total or data data.summaryTotal. Acceptable as complete objects of the form amount: { sum: 1500, avg: 300 }, and an abbreviated entry amount: 1500 if only one operation is expected for the field.

Public API: getSummary() — return the object of the current ones page/total results; setSummary(summary) — manually install summary and redraw the table.

Important: page-results without server work themselves if it’s specified for fields summary; total-results without a server usually won't appear unless explicitly called setSummary(); the final lines are constructed based on schema.fields and are tied to the aliases of the columns.

9.5 Permit Policy

ABGrid Engine allows you to manage permissions to create, edit, and delete data for everyone ABGrid-the componeta. Option policy.enabled controls automatic application only data data.policy from server responses. Manual call setPolicyTree() from the program code works independently of policy.enabled. If the server responses have policy rights in the parameter data data.policy the component cascades it for all its children and for its subgrid. If the server stops sending a policy, it is saved until the page is rebooted or until a new server response containing this policy (at data data.policy = true). The mode is described in more detail in the corresponding section.

9.6 Global policy when responding to a server with 401 status

ABGrid Engine supports global auth-policy for typical script «server returned 401 — user needs to show message and redirect to authorization page».

Configuration:

```
auth: {
    redirectOn401: false,
    loginUrl: '/auth/login',
    redirectDelayMs: 3000,
    redirectOnlyOnce: true,
    showToast: true
}
```

Options:

- redirectOn401 — includes automatic response to HTTP 401;
- loginUrl <TAG1> URL login pages;
- redirectDelayMs <TAG1> delay before transitioning;
- redirectOnlyOnce <TAG1> protection against multiple redirect for multiple parallel requests;
- showToast <TAG1> show the user a notification before the transition.

Behavior: if the server returned 401 and auth.redirectOn401 = true, the component considers this an authorization error; when enabled showToast the user is shown a message from i18n.errmsg.unauthorised or standard text; after delay redirectDelayMs the transition to is performed loginUrl; at redirectOnlyOnce = true repeated 401 will not generate a cascade of identical notifications and repeated transitions.

Important: auth-the policy does not replace server security and does not automatically log in to the user; it only centralizes client security UX-reaction to the 401 that has already happened.

Addition to i18n: for auth-scenario uses the key i18n.errmsg.unauthorised.

The policy can be global and installed on the entire application:

```
//Global policy
//IMPORTANT: call DO new ABGrid(...)
ABGrid.setDefaults({
    auth: {
        redirectOn401: true,
        loginUrl: '/auth/login',
        redirectDelayMs: 3000,
        redirectOnlyOnce: true,
        showToast: true
    },
    debug: {enabled: true}
});
```

For custom queries (for example fetch(download reports, etc.) a public method is provided:

```
await grid.handleUnauthorized(response)
```

Example: downloading a report:

```
const response = await fetch(urls, {...});

if (await grid.handleUnauthorized(response)) {
 return;
}
```

Important: working with Response (Fetch API)

Object `Response` allows you to read the body of the answer **only once**.

❗ Problem

```
await response.json();
await response.text(); // ✕ Error — body already read
```

✓✓ As decided in ABGrid

Method `handleUnauthorized()`:

- uses `response.clone()` for reading the body
- **doesn't spend the main one response**

This allows you to use the answer safely further:

```
if (await grid.handleUnauthorized(response)) {
 return;
}

const blob = await response.blob(); // ✓ works
```

⊘ Limitations

- The method processes **401 status only**
- Other errors (400, 500) shall be processed separately
- You should not read the body again response after blob()/json()/text()

✓ Recommendations

- Use it `grid.handleUnauthorized()` in **all custom ones fetch-inquiries**
- Do not manually duplicate auth logic
- Do not implement custom redirects at 401

🧠 Architectural principle

The whole logic of authorization should be **centralized in ABGrid**e, and not smeared with the application code.

9.7 Section data data

Section data data manages the transport and the way the grid communicates with the application or server.

Key parameters:

- url - endpoint server (if transport="server" is used)
- method - HTTP-method (default POST)
- headers <TAG1> static headers for the server
- filter <TAG1> static data filter on the server
- dataKey is the name of the wrapper field for payload (optional)
- interceptors: request/response/error (e.g., auth/CSRF) hooks. More details in the relevant section.
- controller - processing mode intents: 'internal' or 'external'
- transport - for the internal-controller: 'server' or 'local'. In version 1.0.0 only the value ' is supportedserver'.
- intents - handler intents (intentions) for external-of the controller
- inFormat/outFormat/csvDelimiter - incoming/outgoing data formats

Example of a minimum transport setup:

```
data data: {
 url: '/api/items',
      method: 'POST',
      heads: { 'Content-Type': 'application/json' },
      dataKey: 'data'
}
```

9.7.1 Controller data: internal and external

ABGrid Engine uses a model intents: View generates intentions (tulbar clicks, pagination, sorting), and the controller decides what to do next.

Options controllers:

- internal (default) - built-in controller ABGrid. He calls himself load()/setPage()/CRUD and it can work with the server.
- external - ABGrid only issues intents. Solutions (challenges) load/CRUD(its modals, its requests) accepts the application.

To enable the external controller:

```
data data: {
```

```
 controller: 'external',
  intents: (ctx) => { //ctx.action, ctx.payload, ctx.source,
ctx.grid
     }
}
```

The format of ctx in intents is

- action Is the action string (for example, "toolbar:add" or "pager:page")
- payload Is the object of the action parameters (for example, {page, rpp, rowId, rowIds, sortOrder})
- source - source (for example, "view" or "api")
- grid/api - Grid specimen link

Typical intentsc which generates View:

- toolbar:add | toolbar:edit | toolbar:delete | toolbar:refresh
- pager:page | pager:rpp | pager:refresh
- sort:preview | sort:apply
- filter:apply | filter:clear

Example of an external controller (external application decides what to do):

```
data: {
      controller: 'external',
      intents: async ({ action, payload, grid }) => {
                  if (action === 'toolbar:add') {
 //open your custom form and then save through API
 openMyModalCreate({ onSave: (row) => grid.crud.create(row)
});
                          return;
        }
        if (action === 'toolbar:edit') {
         const rowId = payload.rowId?? grid.getCurrentRowId();
         if (!rowId) return;
         openMyModalEdit(rowId, {onSave: (row) =>
grid.crud.update(rowId, row) });
         return;
    }
    if (action === 'toolbar:delete') {
      const ids = payload.rowIds?? grid.getCheckedRowIds();
      if (!ids?.length) {
        const rid = grid.getCurrentRowId();
        if (rid) ids.push(rid);
 }
 if (!ids.length) return;
      //confirmation can be yours (or use the built-in dialog)
 const ok = await firmApp('Delete entries?');
```

```
      if (!ok) return;
      await grid.crud.deleteRowsByIds(ids);
      return;
    }
    if (action === 'pager:page') {
      grid.setPage(Number(payload.page), {load: true });
      return;
    }
    if (action === 'sort:apply') {
      grid.state.sortOrder = payload.sortOrder;
      await grid.load({ resetPage: true});
      return;
    }
  }
}
```

In mode external ABGrid you can still use built-in CRUD (grid.crud.*) and load(), but the call decision is up to the application.

9.7.2 Format for exchanging with the server

ABGrid Engine allows you to set different exchange formats with the server and set the appropriate parameters for incoming and outgoing traffic. This refers to the data format in data data and first of all, the data of the strings themselves. Formats supported object, aoa, csv.

- In mode object the string consists of field name objects and their values, for example:

```
{
 success:true,
 message: ”OK”,
 data:{
 rpp:10,
 page:1,
 rows: [
     {"id": 1, "name": "Alice", "email": "alice@mail.com"},
    {"id": 2, "name": "Bob", "email": "bob@mail.com"}
 ]
 }
}
```

- Format aoa (Array of array). Recommended format.

```
--------
rows: [
      [1, "Alice", "alice@mail.com"],
   [2, "Bob", "bob@mail.com"]
 ]
```

- Format csv

```
rows: [

        "1;Alice;alice@mail.com",
        "2;Bob;bob@mail.com"

 ],
```

In mode auto the string format is determined automatically by the first line. The outgoing format is similar. Parameter csvDelimiter defines a symbol for separating field values in a string.

9.8 Section crud

Section crud describes the parameters of the built-in one CRUD-engine (operations create/read/update/delete via the server).
Key parameters:

- operParam <TAG1> operation parameter name (default —oper»).
- operCreate /operRead /operUpdate /operDelete — values of the operation parameter for create/read/update/delete.
- deleteBatchSize <TAG1> pack size at mass removal (0 means without limitation).
- deleteProgress <TAG1> show progress when deleting in batches.

9.9 Array of detailed grids detailGrids

The array of detailed (slave) components is given by the array detailGrids[]. There is no need for software initialization of detailed components; the master grid creates them itself if available HTML-of an element from the parameter gridId. Detailed grids are full copies ABGrid Engine.

Every element detailGrids described as an object:

- gridId <TAG1> id DOM-of the container in which the detailed mesh will live (required).
- link:{masterField: '...', detailField: '...'}, where masterField and detailField— names fields communications. The value is taken from the current wizard line from the field masterField and is added to the fine grid filter with a name detailField.
- options <TAG1> configuration ABGrid Engine for detailed grid (parameter autoLoad forced to shut down, the wizard initiates the download).
- cascade <TAG1> if true, after loading the part there is data, the cursor is set to the first line and its own detailed grids are automatically loaded.

Operating principle: the master controls the parts. When changing the current line, the wizard sets the filter in the detailed grid and calls load({ resetPage:true }).
The part filter is formed as merge: existing filter (object or function) + { [detailField]: value from the current wizard line}.

9.10 Array of detailed panels detailsPanels

detailsPanels <TAG1> array of custom UI-of panels linked to the current string. The engine causes them to update when the current line changes. An array consists of objects, each of which specifies the panel itself and its behavior.

Unlike detailGrids:
• detailsPanels they are not grid-components
• the content is completely defined by the user code
• ABGrid Engine controls only the lifecycle and string binding
Setting up detailsPanels executed in grid configuration:

```
detailsPanels: [
  {
    // --- Identification ---
    // (required) unique id panels
 id: "requestDetails"

    //--- Where is DOM panels ---
 // (required) selector | Element | () => Element
    host: "#requestDetailsHost",

    // --- From where take masterId ---
    link: {masterField: "id" }, //default: "id"

    // --- Rules of law auto-binding/cleaning ---
 bindSelector: "[data-panel]", //default: "[data-panel]"
 hintSelector: "[data-panel-hint]", //default: "[data-panel-hint]"
 keepselector: "[data-panel-keep]", //default: "[data-panel-keep]"
 clearValue: "—", //  default: "—"
 emptyHint: "Select an entry...", //default: "" (if empty— hint only
shown/hidden)

 //--- Fully manual mode (override) ---
    //If onShow is set to → autoBind NOT executed
//ctx.masterRow, ctx.masterId, ctx.host, ctx.grid, ctx.reason, ctx.force
    onShow(ctx) {
            //In manual mode, you fill out the DOM yourself.
    },

    //If onClear is set to → autoClear does NOT execute
//ctx.host, ctx.grid, ctx.reason, ctx.prevMasterRow/Id/Field
    onClear(ctx) {
      //In manual mode, you clean the DOM yourself.
    },

    //---- Hooks "after" (for difficult parts) ---
    //Called after autoBind or after onShow
    onAfterBind(ctx) {
      //For example, add a complex block:
```

```
 //renderFiles(ctx.host.querySelector('#filesList'), ctx.masterRow?.files);
 },

    //Called after autoClear or after onClear
    onAfterClear(ctx) {
      //For example, clean up a complex block:
 //ctx.host.querySelector('#filesList').innerHTML = '';
 }
  }
]
```

List of handlers.

Handler	When called	Appointment
`onShow(ctx)`	When currentRow appears	Fully manual mode
`onClear(ctx)`	With currentRow = null	Fully manual mode
`onAfterBind(ctx)`	After autoBind or after onShow	Panel revision
`onAfterClear(ctx)`	After autoClear or after onClear	Cleaning complex blocks

When there is currentRow (current string):

1. If given `onShow` <TAG1>
    - autotiech drawing(autobinding) NOT executed
    - called `onShow`
    - then `onAfterBind` (if any)
2. If `onShow` NOT specified
    - auto-binding is performed
    - then `onAfterBind` (if any)

When currentRow = null:

1. If given `onClear` <TAG1>
    - auto-cleaning is NOT performed
    - called `onClear`
    - then `onAfterClear` (if any)
2. If `onClear` NOT specified
    - auto-cleaning is performed
    - then `onAfterClear` (if any)

The ctx contract

For onShow/onAfterBind

```
{
 id,
 host,
 grid,
 masterRow,
 masterId,
 masterField,
 reason,
 force
}
```

For onClear/onAfterClear

```
{
 id,
 host,
 grid,
 reason,
 prevMasterRow,
 prevMasterId,
 prevMasterField
}
```

Life cycle

onShow(ctx)

Triggers when selecting/changing the current line.

`ctx`:

- `id, host, grid`
- `masterRow, masterId, masterField`
- `reason, force`

onClear(ctx)

Works when `currentRow === null`.

`ctx`:

- `id, host, grid`
- `reason`
- `prevMasterRow, prevMasterId, prevMasterField`

Public API

Available as `grid.panels`.

- `grid.panels.getPanels()`
- `grid.panels.getCurrentRowId()`
- `grid.panels.isEnabled()`
- `grid.panels.setEnabled(enabled, { hideOnDisable=true, refreshOnEnable=true})`
- `grid.panels.refresh({force=false, reason='refresh'})` <TAG1> again apply current a line ko everyone panels
- `grid.panels.showForRow(rowId, {force=false, reason='manual'})` <TAG1> show panels for the specific lines
- `grid.panels.hideAll({reason='manual'})` <TAG1> clear/hide panels (depending on mode)
- `grid.panels.destroy()`

Important: `refresh()` it works **to all panels**.because engine is one and currentRow is one.

Recommendations

- Make a frame in HTML/template, and let the panels only “substitute values”.
- For long texts, fix the block height and enable scrolling (overflow:auto) so that layout doesn’t jump.
- Use it afterClear for complex internal parts of the panel, leave simple fields for auto-cleaning.

Options reason and force

Some methods `grid.panels` and the panel handlers get the following parameters:

- `reason`
- `force`

These parameters are used to control the behavior of the panels and convey the context of the call.

reason

What is it

`reason` <TAG1> string marker for the reason for calling the lifecycle panel.

It does not directly affect the internal logic of the engine, but is transmitted to `ctx` and allows the panel to respond differently depending on the source of the call.

Where used

Transmitted in:

- `onShow(ctx)`
- `onClear(ctx)`
- `onAfterBind(ctx)`
- `onAfterClear(ctx)`

Available as:

`ctx.reason`

Possible values (embedded)

Meaning	When it occurs
`"row-change"`	The user selected a different line
`"refresh"`	Called `grid.panels.refresh()`
`"manual"`	Called `showForRow()` or `hideAll()`
`"disabled"`	Panels were disabled via `setEnabled(false)`
`"enabled"`	The panels were included through `setEnabled(true)`
`"data-empty"`	After downloading the data `currentRow` became null
`"clear-current"`	Called `grid.clearCurrentRow()`

The engine can add new service values in future versions.

Example of use

```
onAfterClear({ reason, host }) {
 if (reason === "data data-empty") {
 host.queryselector('[data data-panel-hint]').textContent
=
```

```
 "No filter data.";
  }
}
```

---

force

What is it

`force` <TAG1> logical flag (`boolean`), indicating that the panel should be updated even if the current line has not changed.

Default:

```
force = false
```

Where used

Transmitted in:

- `onShow(ctx)`
- `onAfterBind(ctx)`

Available as:

```
ctx.force
```

When applied

`force` used in the following methods:

```
grid.panels.refresh({force: true })
grid.panels.showForRow(id, {force: true })
```

Why you need it `force`

By default, the engine may not perform a repeat bind if `currentRowId` hasn't changed.

If you want to force the panel to update (for example):

- the display format has changed
- the language has changed
- updated reference data

- the topic has changed
- the state outside masterRow has changed

— is used by `force: true`.

Example

```
grid.panels.refresh({force: true });
onShow({masterRow, force}) {
 if (force) {
 console.log("Forced update panels");
  }
}
```

Default behavior

If:

- `reason` not transmitted — used `"refresh"` or `"row-change"` depending on the context
- `force` not transferred — is considered `false`

Architectural principle

- `reason` <TAG1> explains *why* a challenge has occurred
- `force` <TAG1> manages *whether the call should occur repeatedly*

They are not required for use, but provide flexibility for complex panels.

Panel marking (required for automatic rendering and cleaning)

Inside `host` mark items that should automatically fill in/clean as follows:

- `data-panel="fieldName"` <TAG1> meaning fields from `masterRow[fieldName]`
- `data-panel-hint` <TAG1> (optional) prompt block, shown in the absence of currentRow

Example:

```
< div id="requestDetailsHost" class="uk-text-small abdp-panel">
 < div class="abdp-line"><b>Company:</b> <span data-panel="company">—
</span></div>
 < div class="abdp-line"><b>Email:</b> <span data-panel="email">—
</span></div>
 < div class="abdp-line"><b>Name:</b> <span data-panel="name">—
</span></div>

 < div class="abdp-line uk-margin-small-top"><b>Message:</b></div>
 < div data-panel="message" class="abdp-msg">—</div>

 < div data-panel-hint class="uk-text-muted uk-margin-small-top">
 Select recording...
 </div>

 <!-- Example of a "complex" part that you clean/draw in onAfter* -->
 < div id="filesList"></div>
</div>
```

If an item should not be automatically cleaned — add an attribute to it `data-panel-keep`.

9.11 UI-helpers

UIThe helpers are set by two parameters and control the permission to use internal system dialogues and the display of a small window showing the execution of an operation by a component, for example, a boot. The settings are specific to each component object ABGrid Engine.

```
dialogues: true | false,
loadingOverlay: true | false
```

9.12 Localization (i18n)

ABGrid Engine uses a centralized set of message texts (i18n) for UI: errors, confirmations, button signatures, etc.

- Use recommendations:
- bring all custom texts to i18n-app dictionary and pass them to grid configuration
- do not hardcode messages in handlers — use keys/templates
- all messages ABGrid must be overridden

Example (conceptually):

```
i18n: {
 common: {
 confirmtitle: 'Confirmation'
  },
 crud: {
 readOnly: 'View-only mode',
 deleteConfirmOne: 'Delete entry?',
 deleteConfirmMany: 'Delete selected entries?'
  },
 validation: {
 required: 'The field is required'
  }
}
```

The user can completely override messages at will in their configuration. The full localization object is shown below:

```
i18n: {
 //CRUD: server response contract error texts (envelope)
 crud: {
 noOpId: 'The server response does not contain the required
opId opcode,
 noRowId: 'The server response does not contain the required
id of the created record'
 completeServerData: 'Incomplete data from the server',
 serverResponse: 'The server's response: ',
 serverRejected: 'The server canceled the operation'
 },
 add: {
 caption: 'Add',
 hint: 'New line'
 },
```

```
edit: {
caption: 'Change',
hint: 'Edit a line'
},
del: {
caption: 'Delete',
hint: 'Delete line(s)'
},
excel: {
caption: 'Excel’,
hint: 'Export in the Excel'
},
word: {
caption: 'Word',
hint: 'Export in the Word'
},
pdf: {
caption: 'Pdf',
hint: 'Export in the Pdf'
},
pager: {
page: "Page. {0},"
from: "from {1},"
first: "First",
prev: "Previous",
next: "Next",
last: "The Last",
refresh: "Update",
rpp: "Line on page:",
recordInfo: "{0} of {1} rows",
emptyTable: "No lines to display"
},
msgAction: "Action",
msgBranch: "Open branch",
msgBranchAria: "Open related strings",
msgLoad: "Download...",
msgSave: "Save...",
msgDelete: "Delete...",
msgDone: "Done",
msgErrorTitle: "Error",
msgNoData: "No data",


confirm: {
```

```
 title: "Confirmation",
 deleteOne: "Delete entry?",
 deleteMany: "Delete selected entries ({0})?",
 deleteLinkedWarn: "Removal of linked data is possible",
 noAskDelete: "No more asking"
 },

 errMsg: {
 readOnly: 'View-only mode’,
 rollbackOperAtServer: "Server operation cancelled!",
 noRowsToDelete: "No lines to delete!",
  noSelectedRowsToDelete: "No ticked lines to delete!",
 crudnotconfigured: "CRUD is not configured. Specify data.url
or connect model.repository."
 },

 editForm: {
 titleEdit: 'Edit an entry'
 titleCreate: 'New entry',
 btnSave: 'Save',
 btnSaveNext: 'Save and continue',
//message template for firm fields (for example, password confirmation)
// {0} — label of the original field
  firmMismatch: 'The value must match the value in the field
«{0}»',
 noChanges: 'No change'
  },

 autocomplete: {
 notFound: 'No matches',
 loading: 'Downloading...',
 error: 'Error'
 },
 buttons: {
 btnClose: "Close"
 btnOK: "OK",
 btnYes: "Yes, yes"
 btnDelete: "Delete"
 btnNo: "No"
 btnCancel: "Cancel"
 }
}
```

9.13 Section editor

Section editor controls the behavior of the built-in form of creating/editing a record (EditForm).
Important: mode «view only» is set by the parameter readOnly at the root options. Editor does not have its own readOnly flag.
Key parameters:

- requestedByDefault <TAG1>definitive fields required in editor (can be overridden at field level in schema).
- exclusiveFields <TAG1> list of fields the editor never shows (service fields).
- confirmDelete / confirmDeleteMany <TAG1> show confirmation of deletion of one/several entries.
- autocomplete <TAG1> default settings for the built-in auto-complete (if enabled at field level).

9.14 Section schema

Section schema it is the central element of the configuration ABGrid Engine.

Through schema described by:
• data fields
• grid's speakers
• display rules
• editor behavior

9.14.1 Order, visibility and width of columns

Configuration option schema.fields contains a description of all fields in the string.
Example:

```
schema: {
 fields: {
 id: {
 type: 'number',
 ui: {
 label: 'ID',
 grid: { visible: true, width: '25%' }
       }
    },
 name: {
 type: 'text',
 ui: {
 label: 'Name',
 grid: { visible: true, width: '15%' },
 editor: { required: true }
       }
    }
  }
}
```

The visibility of the columns is specified by the parameter schema.fields<fname>.ui.grid.visible. The order can be given explicitly in schema.order. If no order is specified, the order in which the fields are declared is used. The field title is specified by the parameter label. Column widths are only set for visible columns and it is recommended to set the width of each column in %, see example above. The width of the columns can be set to `px` by number `width:100`. At the same time `'px'` there is no need to write and it is even harmful. If the table width goes beyond the screen, a horizontal scroll slider will appear. This slider is the same for the table and for the total rows.
Example:

```
schema: {
      order: ['id', 'name']
}
```

9.14.2 Redefining properties in editor

For each field, the default properties of the global one can be overridden editor and set individual properties in editing and recording modes. You can override the obligation, visibility, mode readOnly.

```
fname: {
 editor:{
 edit:{
 visible: true | false,
 readOnly: true | false,
             requested: true | false
 }
    }
}
```

9.15 Configuration debug mode

ABGrid Engine supports lightweight dev/debug-mode for early detection of configuration errors.

Configuration:

```
dev: {
    enabled: false,
    checkCyrillicKeys: true,
    checkUnknownOptions: true
}
```

Let's also say alias debug, which is processed in the same way as dev.

Options: enabled <TAG1>includes checks; checkCyrillicKeys <TAG1> warn if the configuration keys contain Cyrillic; checkUnknownOptions <TAG1> warn about options that the component does not understand.

Mode purpose: quickly catch typos in parameter names; detect random Russian layout in keys; find «extra» options that the developer expects, but ignores.

Important: this is the diagnostic mode, not runtime-protection; it doesn't block the component, but only displays warnings in console.warn; sections schema and i18n they deliberately allow a looser structure and therefore should not be checked as harshly as other configuration sections.

It is recommended to include dev.enabled during the development phase and turn off in production.

9.16 Section view

Section view describes visual configuration and behavior UI grida (heights, tulbar, pager, sorting, checkboxes, subGrid, treeGrid, etc.).
The main parameters are:

- subGrid — nested grid inside the string: {enabled, linkField, options}.
- treegrid — hierarchical string mode: {enabled, column, model, pid, isLeaf, level, levels}.
- tabletitle — table header (enabled, caption, hint, toggle).
- toolBar <TAG1> toolbar setting (buttons, order, custom actions).
- pager <TAG1> page-by-page navigation (enabled, rpp).
- sortable /multiSort / sortOrder — sort and multisort.
- checkbox <TAG1> show string selection checkboxes.
- height / minHeight / maxHeight — dimensions of the grid area.

Section view affects UI, but is not a security mechanism. Server-side validation of rights remains mandatory.

9.16.1 Row selection, dimensions, headings

```
checkbox: true //control the display of the string selection checkbox
height: 400 //height of the entire container of component c px
showHeader: true //manage the rendering of the title of the table itself

//Dimension limitation in the mode of obtaining additional space, px
minHeight: 200
maxHeight: 800

// Title component

tableTitle: {
 enabled: true //permit management
 caption: "Table title" //component header text
//mouseover tooltip (title)
 hint: "Hint",
 toggle: true //allow component collapse to header
  }
```

9.16.2 Sorting and multisorting

ABGrid Engine supports both single-field sorting and multi-sorting (multi-field) at a level UI and formation sortOrder.

The settings are in options.view:

- view.sortable: true|false <TAG1> enable sorting
- view.multiSort: true|false <TAG1> allow multisort (when key held Shift or Ctrl
- view.sortOrder: [] — initial sort order (array {alias, dir})

Example:

```
view: {
        sortable: true,
        multisort: true,
        sortOrder: [
              {alias: 'name', dir: 'asc' },
              {alias: 'createdAt', dir: 'desc' }
        ]
}
```

The user can specify sorting and multisorting by fields according to his preference. Multisorting provides 3 states of each field (except sorting into 1 field) – asc-desc-none. States change in a circle when keys are held Shift or Ctrl. Due to the nature of using a key in browsers Alt it is not used for multisorting and is not processed in any way. The order of fields in multisort is highlighted in each field header. When holding down a key Shift or Ctrl the user can set the sorting to several fields at will as in direction (asc, desc), so is the order itself by fields. When the key is released, a request is sent to the server, in which the sorting order is contained in the parameter array sortOrder., the server is obliged to return the data in accordance with this procedure. Due to the fact that an array is used to set the order of fields by the user and this array is constantly rebuilt under his influence, the order of the selection fields itself is preserved as the user set it. Sorting by 1 field is also available and is triggered when you click on the field title without holding the keys. In this case, the field does not have a third state «none» and it changes in a circle asc-desc, the request to the server is sent immediately by clicking on the column header. The initial sort value is set by the configuration and sent to the server at the first data download, provided that autoloading is allowed.

9.16.3 Toolbar

The toolbar allows you to place custom buttons to perform any actions. Configuration is done in an object toolBar. The rule for setting buttons is very simple - you need to set the action object itself, in which you can write the inscription and hint, for example, like this:

```
toolBar: {
  add: {caption: 'Add', hint: 'New user'},
 //relevant only for treegrid
 addChild: {
     caption: 'Add a child’,
  hint: 'New child string'
 },
 edit:{caption:'Edit',hint:'User Edit'},
 del:{caption:'Delete',hint:'Delete user(s)' }
}
```

Buttons need to be written, existing ones in the default configuration are not automatically added to the toolbar and are rather written there in order not to forget how to set them.

The processor presses buttons one on all buttons, the user can assign his own processor in which to perform certain actions. The operation is identified by passing a parameter whose value matches the name of the button configuration object, i.e. according to the example above, these will be «add», «edit», «del».

```
function toolBarClick(action, ctx, grid) {

     if (action === 'del'){
         grid.deleteCheckedRows();
        return false;
    }
 --------
 }
 usersGrid.onToolBarClick(toolBarClick);
```

Options, transmitted in the handler:

- action <TAG1> name of button (operation)
- ctx <TAG1> context
- grid <TAG1> instance of the component

The user handler is triggered before the built-in one, which must be returned to interrupt operation false, otherwise (if there is no return or return true) he will continue. The product comes with built-in icons to display actions:

- add <TAG1> adding entries
- addChild <TAG1> adding a child entry in mode treegrid
- edit <TAG1> editing the record
- del <TAG1> deleting entries

- excel – export in the Excel
- word – export to Word
- pdf – export to PDF

The built-in editor supports creating, editing, and deleting entries. Export to the above formats is not processed by the engine.

The button handler can be assigned directly in the configuration, for example:

```
toolBar: {
    deleteUsedOrExpired: {
    requestsSelections:false, // see. description below
    requestsData:false,       // see. description below
    caption: 'Cleanse tokens',
 hint: 'Remove used/expired recovery tokens',
 async onClick(grid, payload) {
 payload.preventDefault();
          try {
        //IMPORTANT: url is yours, not options.data.url
          const res = await grid.http.request({
               url: '/admin/api/password-reset-tokens/cleanup',
          //endpoint
     method: 'POST',//or DELETE
     heads: grid.options?.data?.headers || {},
     dataKey: null, //if the API is not a key wrapper
     data: {},          //if you need a request body
               strict: true, //we expect evolution {success,
               message, data}
     unwrapData: true,
     intent: 'toolbar:deleteUsedOrExpired'
      });

     }
             if (res?.success === true) {
               await grid.msgDone(res.message || 'Done');
               await                          grid.refresh();//or
grid.load({resetPage:false})
             } else {
               await   grid.msgError(res?.message   ||   'Token
clearing error');
             }
           } catch (e) {
             console.error(e);
```

```
            await grid.msgError('Inquire to request the
server');
      }
         }
   }
}
```

Also, assign your handler to the arrow function:

```
grid.onToolBarClick((action, ctx, grid) => {
---------
});
```

If standard « buttons are processed in user codeadd», «addChild», «edit», «del» then must be returned false to interrupt the system flow of button press processing. It is recommended to always return false after processing, regardless of the type of buttons.

Important!

When the table is empty, only the «Add» button will be available, provided that the component is not a child. In all other cases, if data is missing, buttons are displayed, but their actions are prohibited.

You can also add « elements to the toolbarselect» with its own buttons for performing actions. Used, for example, for mass data transactions.

Configuration is given as follows (highlighted in bold for select:

toolBar: {

bulkGroup: {
type: 'group', //required parameter
class: 'abgrid-bulk', // CSS-class

bulk: { //group ID (id)
type: 'select', //type, mandatory
caption: 'Operation',
placeholder: '— select —',
options: [// myself select
{value:'delete', caption:'Delete'},
{value:'spam', caption:'Mark how spam' '}
]
},

applyBulk: { // button for the select, id buttons

```
    caption: 'Apply',
    requestsSelection: true
  }
},

 edit: {caption:'Change' } // ordinary button
}
```

Top wrapper (in the name example) bulkGroup«is used to render elements in one container on the page so that when the screen size changes, the action button does not »run away from its own select and the overall layout did not break.
API groups:

- `grid.toolbar.getValue(id)`
- `grid.toolbar.setValue(id, value)`
- `grid.toolbar.clearValue(id)`

As a id for the select serves as a key, in the example it is «bulk», for its button as usual, the key of the button itself, in the example it is «applyBulk».

9.16.4 Toolbar button availability options

requiresSelection: boolean

Determines whether the selected (current) line is required to activate the button.

- `true` <TAG1> button is only active when the selected line is present.
- `false` (default for custom buttons) — line selection is optional.

Typical application:

- `edit`, `delete`, `addChild`
- custom actions on a specific record (cloning, opening a card, etc.)

requestsData: boolean

Determines whether data is required in the table to activate the button.

- `true` <TAG1> the button is only active if the table contains at least one line.
- `false` (default for custom buttons) — data not required.

Typical application:

- export data
- mass operations
- actions that make sense only if records are available

Priority of rules

1. If `requestsSelection: true` <TAG1> the button will be disabled when there is no selected line (even if there is data).
2. If `requestsData: true` <TAG1> button will be disabled when the table is empty.
3. If both parameters `false` or not specified — button is available regardless of the state of the grid.

Default parameter values:

Button	requiresSelection	requestsData
add	false	false
edit	true	false
delete	true	false
addChild	true	false
custom (default)	false	false

9.16.5 Built-in pager

The component has its own built-in pager, which is configured by the object pager, which has only 2 parameter-permit mappings enabled and in an array rpp[], which specifies the parameter for displaying lines on a page with one request.

```
pager: {
 enabled: true ,
 rpp: [10, 20, 30, 50]
}
```

This example shows the default parameters of the built-in pager, which can be overridden by its configuration.

9.16.6 Space management

The engine allows you to control the workspace of others ABGrid Engine-of configuration-based components. The mechanism of action is that when folding the grid to the header (the component itself, not the columns of the table), it calculates its vacated space and distributes it in proportion to the requests among others. When the original size is restored, all consumers are restored to the sizes specified in their configuration.

The configuration is specified by the array parameter view.toggleGrid[].

```
toggleGrids: [
 {
    gridId: 'pointedCompanies', //applicant component (consumer)
   enabled: true,
   share: 100,                  //application as % of available space
       release: true
   },
   .....
]
```

Each array object configures a separate one ABGrid-component. Description of the configuration parameters of one object:

- gridId : string, attribute «id» root div-of the element
- enable : boolean, allow\deny state change
- share: number, in %. Application of each array element for the volume of freed space of the giving component
- reload: boolean. Restart control parameter when increasing space for an array component element. At true the component is set to the maximum possible value rpp (record per page) and an automatic reboot is performed.

AGBrid Engine automatically normalizes (recalculates) the sum of all applications to 100% and then proportionally distributes the free space between them according to the applications and sets it. In version 1.0 of the component, the mechanism operates at the same level according to the principle «one controls many». There is no hierarchical distribution. It works by the minimizing/expanding icon of the «main» component.

9.16.7 Subgrid

Subgrid <TAG1> is a detailed grid displayed inside the row master-grid.
Features:
• subgrid it is being created master-grid dynamically when the line is opened and destroyed when closed
• subgrid always associated with a specific string master
• subgrid uses a separate container within the line
• subgrid can get rights policy from master-grid

Inherently subgrid it is a full-fledged component ABGrid Engine, managed by master-grid and it exists autonomously until the moment of closing; when closing a line with it, it is destroyed.

The configuration is set in the object subGrid and the developer does not need to write any additional code for this mechanism to work. All you need to do is – allow the mode itself, override the communication field in the master grid if necessary, and define the configuration of the subgrid itself as ABGrid Engine <TAG1> component.

```
 subGrid: {
 enabled: false,  //Mode resolution
link:{ masterField: 'id', //communication field in master grid
 detailField: 'id'
        }
options: null   //parameter-configuration of a full ABGrid Engine
}
```

9.16.8 TreeGrid

TreeGrid <TAG1> display mode of hierarchical data in one grid. Used for categories, structures, nested entities. Component version 1.0.0 supports the tree data structure model Adjacency, a model NestedSet can be implemented according to user requests.
TreeGrid not a variety detail-grid. TreeGrid <TAG1> is a mode of displaying hierarchical data within a single grid. The mode is enabled via the configuration view and it doesn't create additional ones grid-instances.

Setting up TreeGrid produced in the section viewthe example includes default parameter values:

```
view: {
        treeGrid: {
              enabled: false,
              model: 'adjacency',
              column:'',
              pid: 'pid',
              isLeaf: 'isLeaf',
              level: 'level',
              levels: 1
       }
}
```

The main parameters of TreeGrid (v1) are

- enabled — allows treegrid mode.
- model <TAG1> tree structure model type.
- column <TAG1> name of the field (column) in which the tree is rendered.
- pid <TAG1> parent node id field (adjacency list).
- isLeaf <TAG1> true\false, a sign that the string is a leaf (true) or branch (false)
- level <TAG1> nesting level field if server sends (optional). If there is no server in the response, the component calculates level automatically.
- levels <TAG1> number of nesting levels to respond from the server per request. Default 1.

Mode TreeGrid it is exclusively a visualization mode. The preparation of the data hierarchy is on the server side or external controller.

9.17 Default configuration (DEFAULT_OPTIONS)

Below is the full default configuration ABGrid Engine

```
export const DEFAULT_OPTIONS = {

  //Manage autoloading of data immediately after initialization,
        //for detailed grids, the master grid forcibly prohibits their creation
  autoLoad: true,

   //Global Policy: View Only mode.
   //Affects ALL write operations (create/update/delete) regardless of whether
   //whether the built-in editor form or the external one is used.
  readOnly: false,

  //Totals/aggregates at the bottom of the grid (per page and/or general).
   // page-the results are calculated on the client according to the current one rows.
   // total the results usually come from the server in data data.summary / data
data.summaryTotal.
  summary: {
          enabled: false,
          page: true,
          total: true,
          pageLabel: 'Σ Page',
          totalLabel: 'Σ Total',
          separator: ' · ',
          emptyText: ''
 //number of decimal places (global adjustment)
  fractionDigits: 2
  },

 //Rights policy (optional).
   //If enabled=true, ABGrid any response the server tries to apply data data.policy through
setPolicyTree().
 //The server is the source of truth: if data data.policy absent, then nothing changes.
  policy: {
  enabled: false
     },

   //Global Authorization/Redirect Policy (Optional).
   //If redirectOn401=true, then at HTTP 401 ABGrid shows error (if UI can)
   //and through redirectDelayMs will make the transition to loginUrl (once).
  auth: {
          redirectOn401: false,
```

```
        loginUrl: '/auth/login',
        redirectDelayMs: 3000,
        redirectOnlyOnce: true,
        showToast: true,
 },
```

```
  // dev/debug mode (minimum configuration checks).
 //Include in development to catch typos in keys and erroneous options faster.
  //Checks:
  // - checkCyrillicKeys: warn if the keys contain Cyrillic (often a typo)
  // - checkUnknownOptions: warn about unsupported options (by known sections)
  //Note: string values (label/caption/hint etc.) are not checked — Russian texts are normal
there.
 dev: {
        enabled: false,
        checkCyrillicKeys: true,
        checkUnknownOptions: true
    },
//transport
    data: {
        url: null,              //url requests
        method: 'POST',         // HTTP-default method
 headers: {},           //static headers (at each request)
 filter: null,          //static filter (at each request)
           //identifier key in centralized requests for one urls
            //if specified, the request to the server turns into an object with this name
 dataKey: null,

 //Interceptors (request/response/error)
// ABGrid Engine knows nothing about auth/CSRF/headers etc., but has a mechanism
//providing the user to modify server requests and responses
//Use acceptors for modification  headers/params/body with every request.
 interpreters: {
      request: null,   //function(ctx) | Array<function(ctx)>
      response: null,  //function(ctx, payload) | Array<function(ctx, payload)>
      error: null      //function(ctx, error) | Array<function(ctx, error)>
 },

    //Data controller
    // internal: ABGrid processes it himself intents and it can work with the server
    // external: ABGrid only issues intents, decisions are made by the application
 controller: 'internal',
```

```
    // transport for the internal controller. reserved for future functionality
 transport: 'server',
    //external intents handler: function(ctx) or object map by action
        intents: null,

    //string formats
        inFormat: 'auto', // 'auto'|'aoa'|'object'|'csv'
//outgoing data format (CRUD, user requests): 'aoa'|'object'|'csv'
 outFormat: 'object',
        csvDelimiter: ';'
    },
 // CRUD (embedded operations)
 crud: {
 operParam: 'oper',          //request parameter names
 operCreate: 'create',
        operRead: 'read',
        operUpdate: 'update',
        operDelete: 'delete',
//limiting the number of lines for deletions in mode batch (0 - no restrictions)
 deleteBatchSize: 0,
//manage progress display at batch-deletions
 deleteProgress: true
 },

 //detail grid array (external containers, mode linked)
// {gridId:'', link:{masterField:'id', detailField:'...'},  options:null}
    detailGrids: [],

 //external UI-panels linked to the current line (NOT grids)
 detailsPanels: [],

 //UI Helpers
    dialogues: true,
    loadingOverlay: true,

 // i18n (localization)
    i18n: {
 // CRUD: server response contract error texts (envelope)
 crud: {
 noOpId: 'The server response does not contain the required
operation code opId',
 noRowId: 'The server response does not contain the required
ID of the created record id',
 completeServerData: 'Incomplete data from the server'
```

```
serverResponse: 'The server's response is: ',
serverRejected: 'The server canceled the operation'
        },
add: {
caption: 'Add',
hint: 'New line'
        },
edit: {
caption: 'Change',
hint: 'Editing a line'
        },
del: {
caption: 'Delete',
hint: 'Delete line(s)'
},
        excel: {
            caption: 'Excel',
            hint: 'Export to Excel'
        },
        word: {
            caption: 'Word',
            hint: 'Export to Word'
        },
        pdf: {
            caption: 'Pdf',
            hint: 'Export to Pdf'
        },

        //Toast (notifications)
        toast: {
            close: 'Close,
            closeAll: 'Close All'
        },

        pager: {
            page: "Page. {0},"
            from: "of {1}",
            first: "First,"
            prev: "Previous",
            next: "Next",
last: "The last one",
refresh: "Update",
rpp: "Line on page:",
recordInfo: "{0} of {1} lines",
```

```
 emptyTable: "No lines to display"
        },
 msgAction: "Action",
 msgBranch: "Open the branch",
 msgBranchAria: "Open related lines",
 msgLoad: "Downloading...",
 msgSave: "Saving...",
 msgDelete: "Deletion...",
 msgDone: "Done",
 msgErrorTitle: "Error",
 msgNoData: "No data",

 confirm: {
 title: "Confirmation",
 deleteOne: "Delete entry?",
 deleteMany: "Delete selected entries ({0})?",
 deleteLinkedWarn: "The linked data can be deleted,"
 noAskDelete: "Don't ask anymore."
        },

 errmsg: {
 readOnly: 'View-only mode’,
 rollbackoperatserver: "Server operation cancelled!",
 noRowsToDelete: "No lines to delete!",
 noSelectedRowsToDelete: "No marked lines to delete!",
 crudnotconfigured: "CRUD not configured. Please indicate data
data.urls or connect model.repository.",
                unauthorised: 'Authorization required'
        },

 editForm: {
 titleEdit: 'Edit entry',
 titleCreate: 'New entry',
 btnSave: 'Save',
 btnSaveNext: 'Save and continue',
            //message template for confirm-fields (such as
password confirmation)
            // {0} — label source field
 confirmMismatch: 'The value must match the value in the field
«{0}»',
 noChanges: 'No change'
        },

 autocomplete: {
```

```
 notFound: 'No match',
 loading: 'Loading...',
 error: 'Error'
        },
 buttons: {
            btnClose: "Close",
            btnOK: "OK",
            btnYes: "Yes,"
            btnDelete: "Delete",
            btnNo: "No",
            btnCancel: "Cancel"
        }
    },
 // view (UI grid as a component/container)
 view: {

 //Manage table header display (THEAD)
  // false: the header is not rendered and is not involved in sorting/"select all" clicks.
    showHeader: true,
             //control the display of the string selection checkbox
       checkbox: true,
 height: null,     //Height of the entire container of the component
//Dimension Limitation
 minHeight: 200,
 maxHeight: 800,

 tableTitle: {  //Component heading
 enabled: true,     //display resolution management
 caption: ","       //component header text
//mousehover tooltip (title)
 hint: ","
 toggle: true  //manage the collapse/expand permission of a component
 },

 sortOrder: [],     //initial sort array
 sortable: true,   //permit management  erasing
 multiSort: true,  //permit management  multisort
 sortIcons: {//sort direction display icons
 asc: '▲',
 desc: '▼',
            none: ''
        },
```

```
 toolBar: {}, //toolbar for displaying action buttons

 //built-in pager
 pager: {
 enabled: true,      //display resolution management
          //array of string count values per page
 rpp: [10, 20, 30, 50]
 },

        //an array of components to control their space when changing their
 toggleGrids: [],

 // subGrid (integrated child grid under the line)
 subGrid: {
//Mode resolution communication master-detail: subgrid.<detailField> =
master.<masterField>
            enabled: false,
            link: {masterField: 'id', detailField: 'id' },
//parameter-configuration of full ABGrid Engine
 options: null
 },

 // tree (UI: hierarchical strings)
 treegrid: {
 enabled: false,
//Which column to display the tree in (main/single tree column).
//It is indicated as alias/field name from schema.
 column: '',
//Wood storage model on server (for selection TreeEngine).
//Now implemented adjacency-list (pid). nestedset reserved for the future
//implementation.
 model: 'adjacency',   //tree data structure model
 pid: 'pid',           //parent field name
 //name of the field of the sign that the string is a "sheet" (not a branch, has no parent)
  isLeaf: 'isLeaf',
 //nesting level: if the server doesn't return, it will automatically change
 level: 'level',
 //request the number of data nesting levels from the server per request
 levels: 1
        },
    },

 // editor (built-in recording editor)
 editor: {
```

```
 requestedByDefault: false,
 //exclude service/editor-only fields from UI editor (insurance)
 exclusiveFields: ['actions', 'extData'],
 //confirmations for deletion
 confirmDelete: true,
        confirmDeleteMany: true,

        autocomplete: {
            url: null,
//default dataKey if options.data.dataKey is empty.
//Query identification key for centralized server-side processing
 dataKey: null,
 queryParam: 'q',      //parameter containing search string
 fieldParam: 'field',  //search Request Field ID
 minChars: 3,          //minimum number of characters to start searching
 debounceMs: 250,      //protective pause in ms
 limit: 30,        //limit on the number of responses from the serer
 strict: true //manage server response (see below)
        }
    },

 // schema (rules). IMPORTANT: The primary key is ALWAYS reserved as a field'id'
 schema: null,

 //defaults of internal status
 page: 1,
 totalRecords: 0
};
```

O parameter strict in the autocomplete.
Parameter strict in the autocomplete determines whether the server must strictly adhere to the response format ABGrid (success, data data, dataKey) or a “soft” mode is allowed in which the component attempts to extract a list of options from any reasonable response format.
In strict mode, any breach of contract is considered a request error, in soft mode — only results in an empty list of options without generating an error.

10. Permission policy (policy) in detail

ABGrid Engine supports a tree-like permission model policy (policyTree) received from the server or installed by software. The component that received the rights policy extends it to its detailed components and to its component subgrid if available. The spread of rights policy occurs in a cascade downward, i.e. detailed grids apply rights policies and move on for their detailed and subgrid components.
Format (example):

```
data data: {
  ...,
 policy: {
     //for the mother component
     allow: { create: true, update: false, delete: false },
     details: {
      //rights policy for detailed grid price
          price: {read: true, write: false }, ...
 },
 // for the yours subnets
 subgrid:{allow:{create:false,update:true,delete:false} }
      }
}
```

Important: option policy.enabled controls automatic application only data data.policy from server responses. Manual call setPolicyTree() works independently of policy.enabled.

11. Editor and schema in detail

Editor responsible for creating and editing records. The editor's behavior is fully described through schema.

11.1 General idea schema

IN THE ABGrid Engine the editor's behavior is described through a single configuration schema.
schema <TAG1> is —the contract«between UI-editor and data: what fields exist, how they are displayed, what values are allowed, and what rules apply when creating and editing.
The key principle is:
• defaults are set at the upper level schema (common to all modes\fields)
• overrides are specified by mode create/edit (only what is different)

11.2 Global settings Editor

Editor <TAG1> it is a subsystem UI to create/edit a record. General editor parameters are specified in options.editor.
The main parameters are:

- requestedByDefault (boolean) — global field binding policy. If true <TAG1> fields count required by default, if false <TAG1> no. On the level schema this can be overridden for a specific field and for modes create/edit.
- visiblebydefault (boolean) — default global field visibility policy in the editor (if the field does not override visible).
- readOnlyByDefault (boolean) — global politics readonly for editor fields by default (if the field does not override readOnly).
- confirmDelete / confirmDeleteMany <TAG1> confirmation of deletion of one record or bundle.
- autocomplete <TAG1> global auto-complete parameters (enabled, debounceMs, limit and the like), which may be locally overridden.

The internal editor opens the automatically created edit form using the «Add» or «Edit» button, the edit entry form also opens by double-clicking on the line.

11.3 Structure schema

In general schema it has the following form:

```
schema: {
  fields: {
    <fieldName>: {
 // defaults fields (general for the create/edit)
       ...
 //redefine for mode only create
 create: {... },
       //redefine for mode only edit
 edit: {... }
 }
 }
}
```

Notes:
• fieldName <TAG1> field key (usually the same as the property name in the data string object).
• Within the field, only “simple” defaults are allowed; anything that depends on the — mode create/edit.
• create/edit only specified properties are overridden, the rest is taken from field and global defaults editor.*ByDefault.

11.4 Fields and their properties

Below is the recommended minimum set of field properties. The specific set depends on the type of editor and UI.

- label <TAG1> human-readable field name (may be i18n-by the key).
- type — Data type/editor widget (text, number, date, select, etc).
- required <TAG1> Mandatory. If not specified — applies editor.requestedByDefault.
- visible <TAG1> Visibility of the field in the form. If not specified — applies editor.visiblebydefault.
- readOnly <TAG1> Reading only (true/false). If not specified — applies editor.readOnlyByDefault.
- validators <TAG1> Set of validators (see section 11.6).
- autocomplete <TAG1> Local auto-complete settings for the field (if the field supports).
- ui <TAG1> UI-field settings (width, classes, layout-options).

11.5 Overrides in modes create/edit

For most projects, the rules in create and edit they differ. IN THE ABGrid Engine this is solved explicitly through nested sections create/edit inside the field description.

Example: field required when created, but not edited after:

```
schema: {
      fields: {
             code: {
                   label: "Code",
                   create: {required: true},
                   edit: {readOnly: true}
             }
 }
}
```

11.6 Validation in the editor

Validation is described at the field level through validators (and/or required).
It is recommended to keep the rules next to the field description, and not blur them by events UI.
Example:

```
schema: {
      fields: {
             email: {
                    label: "Email",
                    type: "text",
                    validators: [
                          {type: "email" },
                          {type: "maxLength", value: 120 }
                   ]
              }
       }
}
```

Important: ABGrid Engine validates the form on the client for UX, but server checks are required in any case. Supported types:

- string
- int
- number
- float
- email

- phone
- date
- enum
- boolean

Important to understand:
The parameters for the validator are
type: ”email”,
requested: true,
minLen: 6,
maxLen: 120,
regex: null

11.6.1 Numerical restrictions

min : number

Minimum acceptable value.

max : number

Maximum permissible value.

messageMin : string

Message in case of violation min.

messageMax : string

Message in case of violation max.

ABGrid Engine uses **strictly fixed set of parameters message***. Arbitrary message parameter names are not supported. Parameter writing rule message : you need to add the parameter itself in style camelCase п, for example
messageRequired: 'Email mandatory',
messageEmail: 'Enter correct email',
messageMinLen: 'Email too much short',
messageMaxLen: 'Email too much long

- Checks are being carried out **only for visible and editable fields**
- If the field **not required** and the value is empty — the other checks are not performed
- All messages message*:
  - they have **fixed names**
  - apply **by type of inspection**, not by field name
- If message* not specified — the engine standard message is used

11.6.2 Checking with a regular expression

regex : RegExp | string | null

Regular expression to check the value.

- Maybe:
  - an object RegExp
  - string (to be converted to RegExp)
  - null <TAG1> validation disabled

messagePattern: string

The full list of acceptable messages is:

- messageType <TAG1> message if the value entered is not of the expected type
- messageMinLen, messageMaxLen <TAG1> string length violation messages
- messagePattern - message when there is a discrepancy regex
- messageMin, messageMax <TAG1> message when number limits are violated
- messageEmail – message if incorrect email
- messagePhone - message if the phone number is incorrect
- messageDate <TAG1> message if date is incorrect
- messageEnum, messageEnumEmpty – message if the value is not included in the list enum or even an empty value

In most cases, declarative parameters are sufficient to validate the field (required, min/max, minLen/maxLen, regex, email, enum etc.). For non-standard scenarios, ABGrid Engine supports custom field validators <TAG1> array of functions that are called after all the built-in checks.

```
schema: {
  fields: {
    email: {
      type: 'email',
      validators: [
        ({value}) => {
          if
(String(value).toLowerCase().endsWith('@example.com')) {
 return 'The domain @example.com is prohibited';
          }
          return true;
        }
      ]
    }
  }
}
```

Operating rules:

- validators <TAG1> array of functions.

- The function should return:
  - true / null / undefined <TAG1> value correct;
  - string <TAG1> error text;
  - {message, code} <TAG1> object of error.
- Validators are performed only for visible and editable fields.
- Custom validators do not replace built-in validators, but **they complement them**.

Cross-field (lower-case) validators are an internal mechanism of the ABGrid Engine and are not a public API.

11.6.3 User validation

If necessary, the developer can write their own check function before saving the record. Mechanism schema.validators.row **complements** built-in validation and execution **after** validate individual fields before sending them to the server.

For example, there are 2 fields, and the situation is such that they or both can be empty, but if one is filled in, then the second should also matter. Then the custom validation could be:

```
validators: {
  row: [
    ({ row }) => {
 const empty = (v) => v == null || (typeof v === 'string' & &
v.trim() === '');
 const a = row.fieldA, b = row.fieldB;
      if ((empty(a) &&!empty(b)) || (!empty(a) & & empty(b)) {
 return 'Fields A and B must be filled in together';
      }
      return true;
    }
  ]
}
```

11.7 Uploading files from the editing form.

ABGrid Engine supports sending files via multipart/form-data data in the create/update scripts if the editing form contains file fields.

Actual multipart-the contract v1: The request is sent as FormData; in part meta transmitted by JSON-metadata of the operation; binary files are passed in separate parts with names of the form file_<fieldName>.

The scheme is something like this:

```
meta = application/json
file_avatar = <binary>
file_document = <binary>
```

Important: at multipartThe -request component should not be forced to specify Content-Type: application/json; boundary generated automatically by browser; internal service fields ABGrid from meta deleted before sending.

Feature v1: in multipart-mode CRUDThe - payload for a single record is sent as a single object row, not as a standard array rows. This is a known feature of the current version v1. Server DTO should expect exactly row in the meta for a single file upload operation.

Practical note for the server side: convenient to accept meta like @RequestPart("meta") DTO/Json and files as separate @RequestPart("file_<alias>") parts.

Example for the Spring Boot (JAVA):

DTO:

```
public class CrudMultipartRequest {
    private String oper;
    private String opId;
    private Map<String, Object> row;

    public String getOper() {
        return oper;
    }

    public void setOper(String oper) {
        this.oper = oper;
    }

    public String getOpId() {
        return opId;
    }
```

```
        public void setOpId(String opId) {
            this.opId = opId;
        }

        public Map<String, Object> getRow() {
            return row;
        }

        public void setRow(Map<String, Object> row) {
            this.row = row;
        }
    }
```

An example of a controller:

```
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;

import java.util.HashMap;
import java.util.Map;

@RestController
@RequestMapping("/api/files")
public class FileCrudController {

    @PostMapping(consumes =
MediaType.MULTIPART_FORM_DATA_VALUE)

    public Map<String, Object> handleCrudWithFile(
            @RequestPart("meta") CrudMultipartRequest meta,
            @RequestPart(value= "file_archive",
required=false) MultipartFile archiveFile,
            @RequestPart(value= "file_avatar", required=false)
MultipartFile avatarFile
    ) throws Exception {

        String oper = meta.getOper();
        String opId = meta.getOpId();
        Map<String, Object> row = meta.getRow();

        if ("create".equalsIgnoreCase(oper)) {
            //1. read data lines from row
 //2. save file (s)
            //3. create an entry in the database
            //4. return id of created record
```

```
            Long createdId = 101L;

 if (archiveFile)!= null & &!archiveFile.isEmpty()) {
                String originalName =
archiveFile.getOriginalFilename();
                long size = archiveFile.getSize();
                // save file
            }

            if (avatarFile!= null & &!avatarFile.isEmpty()) {
                // save second file
            }

            Map<String, Object> data = new HashMap<>();
            data.put("opId", opId);
            data.put("id", createdId);
            data.put("row", Map.of(
                    "id", createdId,
                    "name", row.get("name"),
                    "description", row.get("description")
            ));

            return Map.of(
                    "success", true,
                    "message", "Record created",
                    "data", data
            );
        }

        if ("update".equalsIgnoreCase(oper)) {
            Long id =
Long.valueOf(String.valueOf(row.get("id")));

            if (archiveFile)!= null & &!archiveFile.isEmpty())
{
                // replace/save file
            }

            if (avatarFile!= null & &!avatarFile.isEmpty()) {
                // replace/save file
            }

            Map<String, Object> data = new HashMap<>();
            data.put("opId", opId);
            data.put("row", row);
```

```
            return Map.of(
                    "success", true,
                    "message", "Record updated",
                    "data", data
            );
        }

        return Map.of(
                "success", false,
                "message", "Unsupported operation",
                "data", Map.of("opId", opId)
 );
    }
}
```

11.8 Case study schema

Below is an example of a small set of fields demonstrating defaults and mode overrides:

```
editor: {
 requiredByDefault: false,
 visibleByDefault: true,
 readOnlyByDefault: false
},
schema: {
 fields: {
 id: {
 label: "ID",
 type: "number",
 edit: {readOnly: true},
 create: {visible: false}
    },
 name: {
 label: "Title"
 type: "text",
 requested: true
    },
 price: {
 type: 'number',
 summary: {
 page: ['avg', 'sum'],
 fractionDigits: 3
 }
 }
 createdAt: {
 label: "Created"
 type: "date",
 edit: {readOnly: true},
 create: {visible: false}
    }
  }
}
```

In this example:

• editor.requestedByDefault=false <TAG1> by default, fields are not required
• name.required=true <TAG1> required field
• id in the edit readonly, and in create hidden
• createdAt it is filled with the server and hidden during creation

12. Protocol for exchanging with the server

12.1 dataKey: request wrapper

If parameter options.data data.dataKey given, then requests to the server turn into an object with this key. Example: {"payload": {... } }. The key can be used on the server side to identify requests from different sources (not necessarily) ABGrid Engine) with centralized requests for one urls.

12.2 Data loading (load)

For the load() strict mode is used: the server must return envelope of the form { success:true, message?, data data:{ rows, recordcount?, page?, rpp? } }.

Minimum example answer:

```
{
 success: true,
 message: "OK":
  data: {
 rows: [ {id: 1, name: "A" },... ],
 totalRecords: 10,
 page:1,
    summary: {
      page: {},
      total: {}
  }
}
```

Data fields:

- rows — array of strings.
- recordcount <TAG1> total number of lines (if not specified, used rows.length).
- page / rpp <TAG1> optional: server can return actual values, they are synchronized with grid state.
- summary <TAG1> summary information for the page and for the sample as a whole.(Sum, mean, quantity)

12.3 CRUD-operations and Result

When working with the server, everything CRUD-operations occur using only one method, default POST and they are obliged to return the object Result uniform format (for built-in CrudEngine and for model.repository).

Result <TAG1> single contract between UI, grid and the server. Even with local data, it is recommended to return Result-object.

Result:

{success:boolean, message:string, opId:string|null, data:any, raw:any }

- success <TAG1> the result of the server performing the operation (is also received alias commit=true | false). Recommended use success.
- message <TAG1> text message (for UI/toast and diagnostics).
- opId <TAG1> browser operation id. Required parameter for return by the server to confirm the operation. Used for updating, creating records for status update purposes View client (browser) side. Initially sent by the client to the server for unambiguous identification of the operation.
- data data <TAG1> payload (eg, created/updated string, metadata).
- raw <TAG1> server source response (for server operations).

Important: ABGrid Engine can perform initial data validation, but does not replace server permissions checking and validation. The server remains the only source of truth.

When using an internal editor, the component only sends data to the server when saving data diff <TAG1> i.e. only fields whose values have been changed by the user.

In the new record creation mode, an internal identifier is sent to the server `__clientId` and operation ID `opId`. The server, if the write insertion operation is successful, is obliged to return this one opId, assigned row identifier in the « parameterid» (default). Return __clientId in component version 1.0.0, it is not necessary. This parameter has been entered for future versions for batch create. But already in version 1.0.0, new lines are sent to the server in an array for the same purpose rows.

Example of a request to the server when creating an entry:

```
gridUsers: {
 format: 'object',
   oper: 'create',
   opId:'mks82ecz-1-3gbf8',
 rows:[
     { __clientId: "c_mks7afwn_xbhov4at",
        name: "c_mks7afwn_xbhov4at"
 }
```

```
    ]
}
```

When editing an entry to the __ serverclientIdnaturally, it is not generated or sent; to identify a string on the server side, modified fields and their values are sent, as well as a string identifier.

Example of a request to the server when editing a record:

```
gridUsers: {
 format: 'object',
   oper: 'update',
   opId:'mks82ecz-1-3gbf8cq8',
 rows:[
 { id: "2",
 name: "New name"
     }
   ]
}
```

When records are deleted, an array is sent to the server id deleted strings in the array parameter rowIds, even when deleting just one entry. This is done deliberately to unify the processing of server-side operations and for batch delete.

Example of a request to the server when deleting an entry\entries:

```
gridUsers: {
   oper: 'delete',
   opId:'mks8pnwq-2-5sp493em',
 rowIds:[2,5,9]
}
```

When the server returns modified or inserted lines, the component engine updates them in the internal repository, in the grid itself, and in the form of editing without reloading the entire page.

13. Public API

ABGrid Engine provides stable public API to control the component from the outside. All methods are designed to be used in external code (controllers, forms, pages).

13.1 Life cycle

• destroy() — destroys the grid and frees up resources
• load(opts) — Reloads data (with page save or reset)
• refresh() — redraws UI no data upload

13.2 Working with data

• load(opts) – download data with options
• getRows() - get all lines
• getRowById(id) – retrieve the string object by id
• getCurrentRow() - get the object of the current line
• setCurrentRow(id) – sets the current string to the identifier value id
• getCheckedRowIds() – return an array of strings marked with checkboxes.

13.3 CRUD API

• createRow(data)
• updateRow(id, data)
• deleteRowsByIds(ids)

All CRUD methods return a Result object.

13.4 Work s filters and by sorting

• setFilter(filterObj)
• clearFilter()
• setSort(sortArr)
• clearSort()

13.5 Navigation and UI

• gotoPage(page)
• setPageSize(size)
• showLoader()
• hideLoader()

13.6 Events and handlers

Grid supports registering processors:
• on(event, handler)
• off(event, handler)

Highlights:
• data:loading
• data:loaded
• row:select
• row: deselect
• crud:success
• crud: error

Row events: row:click, row:dblclick, row:current, row:check.

Load events: load:start, data:loaded, load:success, load:error, load:end.

Subgrid events: subgrid:opened, subgrid:closed.

Sorting events and toolbar: sort:changed, toolbar:change.

Events intents: intent, and also itself action-name as event (e.g. sort:apply) if it is emitted through dispatch-intentions.

13.7 Public API (details)

Below is a list of public methods for the Grid instance (GridCore / ABGrid), which are intended to be used from external application code. The table is compiled from the archive source code and reflects the actual signatures, parameters, and returned values.

Notes:
• Methods that return Promise, are asynchronous (they can be await).
• Types are given as JavaScript/DOM types (string/number/boolean/object/Array/Function/HTMLInputElement/AbortSignal etc.).
• If the parameter allows multiple formats (for example, an object or serverRow-array), this is stated in the description.

Method	Parameters (type, description)	Returns
on(name, fn)	name: string — Event name. fn: (payload:any)=>any — Event handler. If he returns it false and payload supports preventDefault(), the default action will be canceled.	Function — unsubscribe function (() => void).
off(name, fn)	name: string <TAG1> Event name. fn: Function <TAG1> Same handler as previously signed.	void — returns nothing.
msg(text, title = '', variant = 'info')	text: any — Message text (will be brought to the line). title: string — Title	Promise<void> — will resolve after closing the dialogue.

	(optional). variant: string — Message variant/type (for example 'info'/'success'/'error').	
firm(text, title = '')	text: any <TAG1> Question text (will be brought to the line). title: string <TAG1> Heading (optional).	Promise<'yes'|'no'|'cancel'> — result of selection in confirmation dialog.
async request({url = null, method = null, data = null, heads = null, params = null, body = null, signal = null, dataKey = null, strict = false, unwrapData = true, intent = 'user:request', acceptors = null} = {})	args: object — Query options (see fields below). args.url: string|null — URL. If not specified — default from HttpClient/settings can be used. args.method: string|null — HTTP method (for example 'GET', 'POST'). args.data: object|null — JSON payload (if used). args.headers: object|null — HTTP headers. args.params: object|null — Query parameters will be added to the	Promise<any> — strict: returns the data object; non-strict: returns «expanded» payload/result.

	URL. args.body: any|null — Raw body (if need to send non-JSON). args.signal: AbortSignal|null — AbortController.signal to cancel the request. args.dataKey: string|null — Key for unwrap data (overrides dataKey by default). args.strict: boolean — If true — contract {success/commit, message, data} is expected and data is returned. args.unwrapData: boolean — If true — with non-strict dataKey can be extracted. args.intent: string|null — String identifier of the intent/context of the request (for logs/interceptors). args.interceptors: object|null — Interceptors (before/after/error, etc.), will be	

	smerjen with default.	
async load(opts = {})	opts: object — Download options. opts.resetPage: boolean — If true — resets page to 1 before loading (useful for parts/subgrids). opts.preserveTree /opts.preserveTreeState: boolean — If true — does not reset tree state when rebooted (treeGrid).	Promise<void> — performs data loading and updates status/render (result via events load:*).
refresh()	—	Promise<void> — same as load({resetPage:false}).
setReadOnly(readOnly)	readOnly: boolean — readOnly mode: blocks write actions (create/update/delete).	this <TAG1> returns a copy of the grid (for chaining).
getReadOnly()	—	boolean — current readOnly flag (considers options.readOnly).
setData(data)	data data: object <TAG1> Data for local installation without network request. Format: {rows:Array, page?:number, rpp?:number,	void <TAG1> installs data and redraws if necessary.

	recordCount?:number}.	
getData()	—	object — data/paging snapshot. If pager is disabled: {rows}. Otherwise: {rpp,page,rows,recordCount}.
setServerData(rows, total)	rows: Array — Strings in server-format (array) or external-format (object) — will be normalized. total: number — Total number of entries (recordCount).	void <TAG1> installs locally server data data (no request).
resetRowFormat()	—	void <TAG1> resets formatting/collation serverRow (used when changing schema/indices).
setCurrentRow(rowId, {emit = true} = {})	rowId: string|number — String ID (PK value). options: object — Options. options.emit: boolean — If true — emits the current string selection event.	void — sets the current string.
clearCurrentRow({emit = true} = {})	options: object — Options. options.emit: boolean — If true — emits the reset	void — clears the current string.

	event of the current line.	
setFilter(filterObjOrFn)	filterObjOrFn: object|Function — Or object filter, or function (grid)=>object to calculate the filter on the fly. The custom function to set the filter must return the filter object.	this <TAG1> returns a Grid instance (for chaining).
setPage(page, {load = true} = {})	page: number — Page number (1..totalPages). options: object — Options. options.load: boolean — If true — automatically calls grid.load().	void.
setRpp(rpp, {load = true} = {})	rpp: number — Rows per page. options: object — Options. options.load: boolean — If true — automatically calls grid.load().	void (also resets page=1).
getTotalPages()	—	number <TAG1> number of pages (ceil(total/rpp), minimum 1).
getVisibleColumnCount()	—	number — number of visible columns.

getCellValueByAlias(rowId, alias)	rowId: string|number — String ID (PK). alias: string — Alias (name) fields/columns.	any|null <TAG1> cell value or null, if row/column not found.
getRowObjectById(rowId)	rowId: string|number — String ID (PK).	object|null — string object in external-format or null.
buildRequestData(overrides = {})	overrides: object <TAG1> Partial overrides for the resulting query object.	object — requestData for query (oper,page,rpp,sortOrder,filter,treeLevels?, ... + overrides).
buildRequestPayload(overrides = {})	overrides: object — Same as buildRequestData().	object — request payload (in the current version it matches buildRequestData).
render()	—	void <TAG1> redraws grid (render DOM).
renderTreeCell({value, column, rowObject})	args: object — Render arguments. args.value: any — Cell value. args.column: object — Column description (column config). args.rowObject: object — String object (external-format).	string <TAG1> HTML/cell text for tree (if treegrid off — returns value as string/value).
bindAutocomplete(inputEl, options = {})	inputEl: HTMLInputElement — The input to which	object — handle {destroy, open, close, clear, setValue, getValue}.

	autocomplete connects. options: object — Autocomplete options (url, debounceMs, limit, minChars, mapping, etc).	
deleteRowById(rowId)	rowId: string|number — String ID (PK).	boolean <TAG1> true if the string was actually deleted locally.
deleteRowsByIds(rowIds)	rowIds: Array<string|number> — String Identifier (PK) Array.	number <TAG1> number of lines actually deleted.
deleteCheckedRows()	—	Array<string> — array rowId actually deleted lines (by status checked).
updateRowById(rowId, rowObjectOrServerRow)	rowId: string|number — String ID (PK). rowobjectOrServerRow: object|Array — Either a string object (external-format) or serverRow (array).	boolean <TAG1> true if the string is found and updated locally.
createRow(rowObjectOrServerRow, opts = {})	rowobjectOrServerRow: object|Array — Either a string object (external-format) or serverRow (array). opts: object — Insertion options. opts.position:	string|null <TAG1> rowId added line or null, if nothing is added.

	'top'|'bottom'|'afterCurrent' — Where to add string (default — bottom).	
createRows(rows, opts = {})	rows: Array<object|Array> — Array of strings (external-format) or serverRows (array). opts: object — Insertion options. opts.position: 'top'|'bottom'|'afterCurrent' — Where to add lines.	Array<string> — array rowId added lines.
onToolBar(handler)	action,ctx,grid	action <string> - the key(s) of the action(s), ctx - context, grid <TAG1> instance of the component
destroy()	—	void <TAG1> frees resources: subgrids, parts, handlers DOM/events, etc.
getDetailGrid(containerId)	containerId – container string without #detail grid or number - array index with detail grid configuration detailGrids[]	grid-instanas detailed grid

getCheckedRowIds()		Returns an array of marked strings or an empty array if none are present
handleUnauthorized(response)	response	Global processing of server response with 401 status

14. Interceptors (request/response/error)

The section describes the mechanism of the interceptors ABGrid Engine: intercepting internal requests and responses HTTP-client to implement end-to-end logic (eg automatic application policy).

Interceptors c ABGrid Engine <TAG1> this is a mechanism for intercepting and processing requests to the server and responses from the server at the object engine level ABGrid Engine. They allow centralized implementation of end-to-end logic without being tied to specific operations (load, create, update, delete). Interceptors are specified by an array (or object if there is only one interceptor of each type). If necessary, the user can write many spoilers of the same type with different tasks, and all of them will be processed within one request/response in the order of priority specified when they are specified in the array.

Why do we need interceptors.

Interceptors are used for:

• automatic rights policy processing (policy)
• centralized analysis of server responses
• implementation of general rules of conduct for all grids
• reducing code duplication c CRUD-operations

The interceptors operate transparently to the user of the component and do not require explicit invocation in the configuration of each operation. Interceptor configuration is carried out in the parameter data data and in general it looks like this:

```
data: {
   acceptors:{
        request: [fn1,fn2,....],
        response: [fn10,fn11,....],
        error: [fn20,fn21,....]
 }
}
```

where fn*-user functions triggered in the order in which they were written

Interface interceptors:

```
interface Interceptor {
  request?: (ctx: RequestContext) => void | false;
  response?: (ctx: RequestContext, payload: any) => any;
  error?: (ctx: RequestContext, error: any) => void | any;
}
```

14.1 Request Interceptors (request)

The request interceptor is called before sending HTTP-request to the server.

In the current architecture ABGrid Engine request interceptors:
• do not change the logic of access rights
• do not add service parameters associated with policy

This is done intentionally: the server decides when and what rights to return to the client. ABGrid Engine does not signal the server to send a policy.

Example of practical application.
The task: The form contains a global filter, for example, the choice of organization, its type, etc. Each time you request, you must transfer the values of this global filter to the server (including when initializing the page and when the grid component is autoloading).
The decision:
Global filter data collection function:

```
function buildUsersFilter(ctx) {
 if (ctx.data === null) return;
 const filter = {};
 .....
 filter.ctype = selectedValueOf('cmbCompanyType');
 filter.company = selectedValueOf('cmbCompany');
 filter.urole = selectedValueOf('cmbUserRoles');
    .....
 if (ctx.data.filter === null) ctx.data.filter = filter;
 else ctx.data.filter = {...ctx.data.filter, ...filter};
 }
```

Configuration component:

```
data: {
        interpreters: {
          request: buildUsersFilter,
      },
     }
```

More example of use (we send the checkbox status to the server):

```
const grid = new ABGrid({
 .....
 data: {
 interpreters: {
 request:(ctx) => {
 if (ctx.data === null) return;
 ctx.data.extData = {};
 ctx.data.extData["customresult"] = toggle.checked;
 }
     }
});
```

14.2 Response Interceptors (response)

The response interceptor is called for EACH server response received through the internal one HTTP-grid's client. The main task of the response interceptor is — analysis of the standard envelope server response and application of global logic.

Parameter payload this type of interceptor is transmitted by value and can be replaced in the user code.

```
response(ctx, payload) {
     ...
     return payload;// ✓ it is possible replace
}
```

Automatic application of rights policy
If the grid configuration has an option enabled `policy:{ enabled: true }`, the answer spoofer performs the following logic:
1. Checks the presence of an object data data.policy in the server response
2. If policy present — calls grid.setPolicyTree(data data.policy)
3. The policy is preserved entirely in the internal state of the grid
4. The policy automatically applies to:
   • master grid
   • detailed grids
   • subgridam

If `data data.policy` none, no rights changes occur.
Manual policy setting

Method `setPolicyTree(policyTree)` is public API and can be called manually from user code.

Important:
• `setPolicyTree()` works regardless of option policy.enabled
• `policy.enabled` controls ONLY automatic policy application from server responses
Why policy.enabled may be disabled.
Option policy.enabled can be disabled in the following cases:
• the politics of rights comes from another API (for example, /me or JWT)
• a mock server or offline mode is used
• the rights policy is too voluminous and should not be conveyed in every response
• integration requires complete control by the application

At `policy.enabled = false` grid completely ignores `data data.policy` from server responses.
Interceptor coupling with readOnly
Important: Interceptors do not control the mode readOnly.
Mode readOnly:
• is the local mode of operation of the grid
• managed through configuration or methods `setReadOnly()/getReadOnly()`
• takes precedence over `policy`

Even if `policy` allows CRUD-operations, `readOnly=true` prohibits any changes to data.
So:
• the server remains the source of truth
• the client stores the rights status independently
• the rights policy is applied uniformly and centrally

14.3 Error Interceptors(error)

This type of interceptors are called **in case of an error in performing the operation**, after attempts to execute via the controller (built-in or external), but **to the** standard Grid Reaction (messages, rollback, completion of the operation). The user can handle query errors by installing one and more interceptors with type «error». 2 parameters –context are passed to the handler ctx and the object of the error error. Parameter ctx it is passed by link and can be analyzed, logged, and extended. Parameter error not strictly standardized, which gives greater flexibility to the user.

The order of execution is as follows:

1. controller (internal/external)
2. error receptors
3. standard Grid reaction (toast/msg/rollback)

14.4 Interceptors error. Recommendations

Definitely

- there is a server
- there are access rights
- there are business mistakes
- there are different types of users

Very useful

- centralized error processing
- single UX
- logging
- analytics

You don't have to use it

- if the data is local
- if everything is «simple»
- if there are enough default messages

What? not worth it do in error-interceptor

❌ Attempt to re-execute CRUD
❌ Manually change grid data
❌ Do UI actions at «maybe»
❌ Rely on a specific format error no checks

Examples are use of

- logging

```
error(ctx, err) {
  console.error('[ABGrid error]', {
    op: ctx.op,
    rowId: ctx.rowId,
 error: err
  });
}
```

- Custom processing HTTP-of codes

```
error(ctx, err) {
  if (err?.status === 401) {
    auth.logout();
    router.go('/login');
  }

  if (err?.status === 403) {
    grid.msgError(Not enough right');
  }
}
```

- Server error spoofing\normalization

```
error(ctx, err) {
  if (err?.code === 'VALIDATION_ERROR') {
 return {
      message: 'Data validation error',
      details: err.details
    };
  }
}
```

- Global UX-behavior

```
error(ctx, err) {
  if (ctx.op === 'load') {
 grid.ui.toast.show('Failed to load data', 'error');
  }
}
```

- Suppression of standard behavior

```
error(ctx, err) {
  return false; //say to the grid: "I processed the error myself."
}
```

14.5 Result

Interceptors c ABGrid Engine provide a clean and scalable architecture, being the only point of user intervention in server requests and responses. This allows you to use ABGrid Engine both in simple projects and in complex ones enterprise-integrations.

15. Mode Master / Detail more details mode

ABGrid Engine supports complex hierarchical data display scenarios:

master–detail, subgrid (detailed data in the line), treeGrid, detailsPanels. All modes can be combined with each other. There are no restrictions on the complexity of the hierarchy; the component supports all the necessary functionality automatically.

Master-grid manages detailed grids. When changing the current line master:
• retrieves the value masterField from the current line
• installs the filter in a detailed grid with parameter detailField with field value masterField
• initiates loading of detailed grid data

The detailed grid does not subscribe to the master's events on its own. The download initiator is always master-grid. The detailed grid does the same with its detailed grids (if specified), the process is processed in cascade and stopped, provided that the detailed grids do not have their own detailed grids. The process is fully automatic, defined solely by configuration, and does not require writing any additional code to control it. The master grid itself creates detailed grids from configuration parameters, however div-items should already be on the page. It is not prohibited to create detailed grids before creating a master grid, but it is also not recommended. When creating detailed ones, the master component checks their existence and does not create new ones if they are available.

15.1 Combination of modes

ABGrid Engine allows combinations of modes:
• treegrid + detailGrids
• master/detail + subgrid
• treegrid + subgrid

The data loading logic remains the same:
• master initiates loading of dependent grids
• each one grid manages only its direct child components

16. Mode ReadOnly

Mode ReadOnly designed to translate an object ABGrid Engine to state «view only». In this mode, the user can view data, navigate pages, sort and filter records, but any data modification operations are prohibited.

16.1 Purpose

ReadOnly in the ABGrid Engine is a global component policy. It does not refer to a single editor or form, but describes the policy of data change operations as a whole.
Key idea:
• the data can be read
• data cannot be created, modified or deleted

16.2 How to enable ReadOnly

Mode ReadOnly it is set in the root of the grid configuration:

```
const gridUsers = new ABGrid({
       readOnly: true,
        ...
});
```

It is the single source of truth for the entire grid, including
• built-in editor
• toolbar
• public CRUD-methods
• external controller

16.3 What is allowed in the mode ReadOnly

In mode ReadOnly the following actions are allowed:

- downloading and updating data (load, reload)
- navigation by pages
- sorting and filtering
- string selection (current/checked)
- opening forms in view mode

16.4 What is prohibited in the regime ReadOnly

In mode ReadOnly any data modification operations are prohibited:

- create (creating new records)
- update (editing existing records)
- delete / deleteMany (deletion of one or more entries)

When attempting a prohibited operation:
• no request is sent to the server
• the operation terminates locally
• the object is returned Result with an error sign

16.5 ReadOnly and UI

Mode ReadOnly doesn't change the structure UI automatically.
Buttons toolbar may remain visible, but actions will be blocked.

This approach allows:
• use one layout for different roles
• manage button visibility separately from data policy

16.6 ReadOnly and Editor

Built-in editor obeys the regime readOnly:
• forms can be opened for viewing
• save buttons do not perform actions
• fields can be translated into readonly/disabled

16.7 ReadOnly and an external controller

All public CRUD-component methods take into account mode readOnly. This means that the external controller cannot bypass the restrictions even if it does create/update/delete directly.

16.8 Recommendations for use

Recommended application scenarios ReadOnly:

- directory View Mode

- limited access for users without editing rights
- temporary blocking of changes (maintenance mode)
- using a component object as view-of the component

17. UI-intents

UI-intents – is how the View component module interacts with the application. Intent — is a declarative notice of user intent.
ABGrid does not decide, *what to do*, he reports, *what the user wanted to do*.

Important:

- intents always generated
- controller mode only affects processing, not for generation

General scheme:

UI action
<TAG1>
intent { action, payload }
↓
controller (internal or external)
<TAG1>
(optional) server
<TAG1>
update data/refresh

17.1 List standard intents

This is **very important**. Example:

- Toolbar:
  - `toolbar: add`
  - `toolbar:edit`
  - `toolbar: delete`
  - `toolbar: refresh`
- Pager:
  - `pager: page`
  - `pager:rpp`
  - `pager: refresh`
- Sort:
  - `sort: preview`
  - `sort: apply`
- Filter:
  - `filter: apply`
  - `filter: clear`

Important:

The intents list is part of the public API component.

Example 1. External controller

```
data: {
  controller: 'external',
  intents: ({ action, payload, grid }) => {
    if (action === 'pager:page') {
      controllerLoadPage(payload.page);
 }
  }
}
```

ABGrid it does not download data itself, but notifies the application of the user's intention.

Example 2. Clean View.

```
data: {
  controller: 'external',
  intents: (e) => {
    presenter.handleIntent(e);
  }
}
presenter.handleIntent = async ({ action, payload}) => {
  if (action === 'sort:apply') {
    const rows = sortInMemory(payload.sortOrder);
    grid.setData(rows);
    grid.refresh();
 }
};
```

17.2 What intents they don't

Important:

- intent ≠ HTTP request
- intent ≠ CRUD operation
- intent ≠ business logic

18. Global statistics helpers

ABGrid provides several static ones helper-methods and properties.

Version: ABGrid.VERSION, ABGrid.version, ABGrid.getVersion().

Date/time assistants:

ABGrid.today() // YYYY-MM-DD,

ABGrid.now()//ISO datetime,

ABGrid.timestamp()//Unix timestamp in milliseconds.

Static autocomplete API:

ABGrid.autocomplete.bind(inputEl, options), ABGrid.autocomplete.attach(inputOrSelector, options), ABGrid.autocomplete.bindGrid(grid, inputEl, options).

Purpose: quickly get a version of the connected assembly; use single ones helper-date functions in configuration and forms; connect autocomplete without creating a built-in one editor.

Example from the component site admin panel.

In the editor, install in the field date current values

```
validFrom: {
 type: 'date',
 ui: {label: 'The beginning actions',

 grid: {visible: true, width: '10%', colType: 'date'}},
 editor: { type: 'date', default: ABGrid.today,
   edit: {required: true, readOnly: false},
   create: {required: true, readOnly: false}
 }
},
```

19. Typical use cases

This section provides practical use cases ABGrid Engine.
The examples are focused on real-world application problems and demonstrate recommended architectural approaches.

19.1 Master + Detail (classic reference book)

Scenario:
• master-grid displays a list of entities (such as orders)
• detail-grid displays related data (order positions)
• master manages downloading detail

```
const ordersGrid = new ABGrid('#orders',
 {
  autoLoad: true,
       data: {url: '/api/orders' },
     detailGrids: [{
         gridId: 'orderItems',
 link:{masterField: 'id',detailField: 'ordersId'
         options: {
  gridId: 'orderItems',
  data: {url: '/api/order-items' }
  }
  }]
 });
```

When changing the current line to ordersGrid:
• the value of the field is extracted id
• in detail-grid a filter is installed
• detail-grid reboots

19.2 Master + external form of editing

Scenario:
• the grid is used only to select an entry
• editing is done in an external form
• external code controls CRUD

```
ordersGrid.on('row:select', (row) => {
  loadform(row.id);
});

function saveForm(data) {
  ordersGrid.updateRow(data.id, data)
    .then(result => {
      if (result.success) ordersGrid.reload();
    });
}
```

19.3 Use external controller

Scenario:
• ABGrid does not directly execute requests
• the whole logic of loading and saving data is in the external code

```
const grid = new ABGrid({
  data: {
    controller: 'external'
  }
});

grid.on('intent', (intent) => {
  if (intent.type === 'load') {
    fetchData(intent.params).then(rows => {
      grid.setData(rows);
    });
  }
});
```

19.4 Grid in mode ReadOnly

Scenario:
• the grid is used as a viewing component
• changes to data are prohibited

```
const gridUsers = new ABGrid({
       readOnly: true,
       data: {url: '/api/logs' }
});
```

19.5 TreeGrid (hierarchical reference book)

Scenario:
• one grid displays hierarchical data (categories, structures, divisions)
• mode is used view.treegrid for visualizing a tree
• the data source can be either flat (parentId), as well as a pre-prepared server

Example (adjacency list/parentId):

```
const categoriesGrid = new ABGrid('#categories', {
        autoLoad: true,
        data: {
                url: '/api/categories'
         },
        view: {
        treeGrid: {
                enabled: true,
                pid: 'pid',
                column: 'name',
           }
  }
});
```

Notes:
• TreeGrid <TAG1> is a single table display mode, not detail-grid.
• Loading children can be executed by:
 <TAG1> with one request (all nodes at once)
 <TAG1> lazy (by opening the node) if the server supports separate endpoint.
The specific mode depends on the implementation data data.controller and yours API.
TreeGrid can be combined with detailGrids:
• master (treegrid) selects a node
• detail-grid shows the associated records of the selected node

19.6 Using system dialogues in user code

ABGrid Engine provides unified system dialogs that can be called from user code. They are used for:

- display messages to the user;
- request confirmation of actions;
- UI Unification (Single Style, i18n, Behavior);
- works in asynchronous scenarios (via Promise).

System dialogues are accessible through Grid instance methods.

19.6.1 Message dialogue grid.msg()

Appointment

Method `msg()` used to display informational, warning and erroneous messages to the user.

Dialogue **blocks execution of user script** until closing and returns `Promise`.

Signature

```
grid.msg(text, title?, variant?) : Promise<void>
```

Options

- text <TAG1> message text (any type, will be brought to the line)
- title *(optional)* <TAG1> dialogue title
- variant *(optional)* <TAG1> message type
  (for example: "info", "success", "warning", "error")

Example 1. Simple information message

```
grid.msg('Data saved successfully');
```

Example 2. Message with header

```
grid.msg(
 'The changes will take effect after the page is rebooted',
 'Attention'
);
```

Example 3. Error message

```
grid.msg(
 'Error saving data',
```

```
 'Error',
 'error'
);
```

Example 4. Use with await

```
await grid.msg(
 'The operation is completed',
 'Done',
 'success'
);

//the code will only execute after the dialog is closed
console.log('The dialogue is closed');
```

19.6.2 Confirmation dialogue grid.confirm()

Appointment

Method `confirm()` used to request confirmation of an action from the user (deletion, mass operations, irreversible changes, etc.).

Method **always asynchronous** and returns the user's selection result.

Signature

```
grid.confirm(text, title?) : Promise<'yes' | 'no' | 'cancel'>
```

Return values

- 'yes' <TAG1> user confirmed the action
- 'no' <TAG1> the user refused
- 'cancel' <TAG1> user closed dialog or clicked — Cancel«

Example 1. Simple confirmation of action

```
const result = await grid.confirm('Delete the selected
entry?');

if (result === 'yes') {
    grid.deleteCheckedRows();
}
```

Example 2. Confirmation with header

```
const answer = await grid.confirm(
 'All selected entries will be deleted without the possibility
of recovery.',
 'Confirmation of removal'
);

if (answer !== 'yes') {
 return;
}
```

//continuation of the operation

Example 3. Use of the toolbar button in the handler

```
grid.on('toolbar:click', async (payload) => {
 if (payload.action !== 'delete') {
 return;
    }

 const res = await grid.confirm(
 'Are you sure you want to delete the entries?',
 'Deletion'
    );

 if (res !== 'yes') {
 //cancel default action
 payload.preventDefault();
 return false;
    }
});
```

Example 4. Confirmation before user request

```
async function deleteWithConfirm(rowId) {
 const res = await grid.confirm(
 'Delete entry?',
 'Confirmation'
    );

 if (res !== 'yes') {
 return;
    }

 await grid.request({
 urls: '/api/delete',
 method: 'POST',
 data data: { id: rowId },
 strict: true
    });

    grid.msg('The entry has been deleted', 'Done', 'success');
}
```

19.6.3 Confirmation dialogue grid.confirmEx()

Appointment

Method `confirmEx()` used to request confirmation of an action from the user (deletion, mass operations, irreversible changes, etc.). Different from confirm() advanced settings.

Allows:

- set dialogue style (`variant`)
- fully configure the buttons
- manage focus
- add a checkbox (for example: “Do not ask anymore”)
- get a structured result

Method **always asynchronous** and returns the user's selection result as an object.

Signature

```
await grid.confirmEx(message, title?, options?)
```

Parameter	Type	Description of the description
`message`	`string`	Message text
`title`	`string`	Dialogue title
`options`	`object`	Additional parameters

Options options

Parameter	Type	Description of the description
`variant`	'default'	'danger'
`buttons`	`Array`	Array of buttons
`focus`	'yes'	'cancel'`
`checkbox`	`object`	Checkbox setup

Parameter **variant** controls the visual style of the confirmation window.
The following values are supported:

Meaning	Appointment
`'default'`	Normal neutral dialogue
`'danger'`	Destructive effect (removal, cleaning, discharge)
`'warning'`	Potentially dangerous action
`'info'`	Information confirmation

Button format:

```
{
  label: 'Delete’,
  value: 'yes',
  className: 'abgrid-btn-danger'
}
```

Format checkbox:

```
{
 text: 'Don't ask anymore,
 checked: false,
 position: 'buttons-left', //optional
  onChange: (checked) => {}
}
```

Return value – object:

```
{
  value: 'yes' | 'cancel' | 'no',
  checked: boolean
}
```

Minimal call

Example 1:

```
await grid.confirmEx('Delete entry?');
```

Example 2 (indicating the variant):

```
await grid.confirmEx('Delete entry?', 'Confirmation', {
variant: 'danger'});
```

Default behavior

If in parameter options not specified buttons, focus or checkbox, the component automatically applies the standard configuration.

Default buttons

- 2 buttons are created:

  Confirmation button — value: 'yes'
  Cancel button — value: 'dancel'

Button texts are taken from options.i18n.buttons (btnYes, btnCancel). If i18n not configured, built-in values are used.

Classes for buttons:

- **`primary`** <TAG1> it is *role/emphasis* (can change color depending on variant)
- **`abgrid-btn-*`** <TAG1> this is *semantic class buttons* (danger/primary/neutral)
- For destructive dialogues:
  - Confirm = `abgrid-btn-danger`
  - Cancel = `abgrid-btn-primary` (without `primary`)

Example 3. Simple confirmation of action

```
const r = await grid.confirmEx(
 'Delete 10 entries?',
  'Confirmation',
  {
    variant: 'danger',
 buttons: [
      {label: 'Delete', value: 'yes', className: 'abgrid-btn-
danger'},
      {label: 'Cancellation', value: 'cancel'}
    ],
    focus: 'cancel'
 }
);
if (r?.value === 'yes') {
  // action
}
```

Example 4. Example s checkbox

```
const r = await grid.confirmEx(
 'Delete all entries?',
  'Confirmation',
 {
    variant: 'danger',
    checkbox: {
      text: 'Don't ask anymore,
      checked: false
    }
  }
);

if (r?.value === 'yes') {
  if (r.checked) {
 //save the "don't ask" setting
  }
}
```

If built-in UI-dialogue is not available, confirmEx() uses it automatically grid.confirm().

19.6.4 Recommendations for use

- ❗ **Always use await** or .then() when working with msg() and confirm(), confirmEx()
- ❗ Do not use window.alert() and window.confirm() <TAG1> system dialogues:
  - support i18n;
  - consistent in style;
  - they work correctly with asynchronous grid scenarios
- ✅ Use it confirm(), confirmEx() **before performing actions**поданид поданид поданининининиковининининининиковинининининини□
- ✅ In event handlers, you can undo an action via payload.preventDefault()

19.7 Notifications (toast)

ABGrid Engine supports non-blocking notifications (toast messages) designed to inform the user about the progress and result of operations no user script interrupt**.**

Toast notifications are used when:

- no confirmation of action required;

- you can’t or shouldn’t block the interface;
- it is important to show the status of the operation (success, error, warning, information).

Notifications are displayed on top of the interface and automatically hidden after a specified time interval.

19.7.1 Method grid.toast()

Appointment

Method `toast()` displays a short notification to the user and **does not block code execution**.

Signature

```
grid.toast(text, variant?, options?): void
```

Options

- text <TAG1> notification text
  (any type will be reduced to a string)
- variant *(optional)* <TAG1> notification type
  Possible values (recommended) are
  - "info"
  - "success"
  - "error"
- options *(optional)* <TAG1>display settings object:
  - timeout: number <TAG1> display time in milliseconds
    (default is global value)
  - closeable: boolean <TAG1> display close button
  - position: string <TAG1> position on screen
    (for example: "top-right", "bottom-left" etc.)

19.7.2 Examples of use

Example 1. Information notice

```
grid.toast('Data is loaded');
```

Example 2. Notification of a successful operation

```
grid.toast(
 'Record successfully saved',
 'success'
);
```

Example 3. Error notification

```
grid.toast(
 'Error while performing operation',
 'error'
);
```

Example 4. Notification with custom options

```
grid.toast(
 'Changes saved',
 'success',
    {
 timeout: 5000,
 closeable: true,
 position: 'bottom-right'
 }
);
```

19.7.3 Use in asynchronous scenarios

Toast notifications are convenient to use in asynchronous operation chains where UI blocking is not desired.

```
try {
 await grid.request({
 urls: '/api/save',
 method: 'POST',
 data data: formData,
 strict: true
    });

    grid.toast('Data saved', 'success');

} catch (e) {
    grid.toast('Error saving', 'error');
}
```

19.7.4 Use in conjunction with grid events

```
grid.on('load: success', () => {
    grid.toast('Data successfully downloaded', 'info');
});

grid.on('load: error', () => {
    grid.toast('Data loading error', 'error');
});
```

19.8 When to use toast, and when msg / confirm

The script	Recommended tool
Action confirmation required	confirm()
You need to block the script	msg()
Informing without blocking	toast()
Background operation error	toast('...', 'error')
Critical error	msg('...', 'Error', 'error')

19.8.1 Dialogues and notifications: UX-recommendations

- ✅ Use it toast() for the **status and background messages**
- ❗ Do not use toast() for actions requiring informed confirmation
- ✅ For CRUD errors, the following combination is often better:
  - toast() <TAG1> for a brief notice
  - msg() <TAG1>for detailed description (as required)
- ❗ Do not overuse the number of notifications — they should not «spam» the user

• confirm() <TAG1> only before actions that modify data
• msg() <TAG1> for blocking and critical messages
• toast() <TAG1> for background and status notifications
• error-interceptors — for centralized UX

Do not use toast for confirmations and do not block UI unnecessarily.

19.8.2 Result

Toast notifications in ABGrid Engine:

- do not block UI;
- easily embedded into asynchronous code;
- support the same style and i18n;
- perfectly complement system dialogues (msg, confirm).

19.9 Autocomplete for custom input

ABGrid Engine allows the use of subsystem «autocomplete» not only in the built-in editor, but also connect it to arbitrary custom ones input-to elements outside the grid. This is especially useful when using external forms and custom forms UI, when the developer wants to reuse the mechanics of finding and selecting values. By default, ABGrid Autocomplete expects the standard server response format: items: [{id, value}]. When used autocomplete for custom input it is possible to specify mapping of response fields (valueField, textField) if the server returns data in an arbitrary format.

• autocomplete in the ABGrid Engine not strictly tied to the internal editor
• any input can be registered as a source autocomplete
• a single request mechanism is used debounce and processing the response

Example of a connection autocomplete to the custom one input:

```
<input type="text" id="companyInput" placeholder="Enter
company name">
const input = document.getElementById('companyInput');

grid.bindAutocomplete(input, {
      url: '/api/companies/autocomplete',
      valueField: 'id',
      textField: 'name',
      minLength: 3
});
```

or

```
ABGrid.autocomplete.attachStandalone('#companyInput', {
      url: '/api/autocomplete/cities',
       valueField: 'id',
       textField: 'name'
});
```

IN THE this an example:
• user introduces text in the ordinary input
• ABGrid Engine executes request for request autocomplete
• at choice values input it's filling up text meaning
• id can be saved developer separately (data-attribute, hidden input and t.p.)
Autocompleteи, connected to the user’s input:
• does not depend on schema
• does not require availability editor
• can be used in conjunction with an external controller

20. Work c internal HTTP-the client

ABGrid Engine allows arbitrary operations to be sent to the server via an embedded HTTP client.

This is used for:

- bulk operations
- custom actions of tulbar
- server commands (export, change of status, etc.)

20.1 Sending a request via grid.data data.request()

Recommended way — to use the internal DataEngine mechanism:

```
const ids = grid.getCheckedRowIds();

if (!ids.length) {
 grid.toast('No selected lines', 'error');
 return false;
}

const operParam = grid.options?.crud?.operParam || 'oper';

const response = await grid.data data.request({
 requestData: {
 [operParam]: 'bulkDeleteSpam', //name of operation
 ids                            //array of id strings
 },

 unwrapData:false, //redefine, see below
 strict: false,
 intent: 'user:bulkDeleteSpam'
});

grid.toast ('The operation is completed', 'success');
```

- takes it `urls` from `options.data.url`
- uses internal HttpClient
- automatically applies:
  - headers
  - dataKey
  - unwrapData
- passes through the acceptors
- supports loading/strict mode

UnwrapData parameter can be overridden as false and then response will contain more than just data.but also success and message.

20.2 Use buildRequestPayload()

If the operation is to take into account:

- current filter
- sorting
- pagination
- treegrid levels

you can add:

```
const requestdata = {
  [operParam]: 'bulkSetStatus',
  ids,
  ...grid.buildRequestPayload()
};
```

20.3 Alternative way — grid.request()

If required:

- another URL
- another HTTP method
- separate endpoint

can be used:

```
await grid.request({
  url: '/api/admin/bulk',
  method: 'POST',
  data: {
    oper: 'bulkDelete',
    ids: grid.getCheckedRowIds()
  },
  intent: 'user:bulkDelete'
});
```

20.4 Parameter strict

Strict parameter controls the how hard the server response is checked**.**

Used for regular use `load()`.

Then:

- the server is expected to return:

    - `success`
    - `rows`
    - `total`
    - possibly `message`

- if the structure does not match — will be considered an error
- DataEngine applies the following standard logic:

    - updates rows
    - updates total
    - starts redrawing
    - shows messages

This is “strict data download contract”.

If `strict: false` (custom operations)

Used for:

- bulk operations
- export
- changes in status
- server commands
- any non-standard calls

In this mode:

- the server response is NOT required to contain `rows`
- NOT checked for availability `total`
- DataEngine does NOT attempt to redraw the table
- payload returns “as is”

What happens if strict leave it true for the bulk?

DataEngine can:

- expect rows
- try to update the data
- react like a regular load
- show “response structure error”

That is, the behavior will become incorrect. In short, then strict <TAG1> whether or not this is a standard data download.

A summary table for setting this option:

The script	strict
load/reload	true
bulk operations	false
export	false
free command	false
the server returns new rows	true

21. Working in mode View

In View mode, ABGrid Engine acts solely as a visual component that generates intents, but does not make decisions about loading and saving data.

In this mode, ABGrid Engine performs **only a performance**: component **doesn't solve anything about loading/CRUD**a, but only:

- renders data,
- stores UI state (page, rpp, sort, filter),
- **issuer "intent" (intent)** about user actions.

All decisions (what to load, how to sort, what to do when tulbar clicks) remain in yours **Controller/Presenter**.

21.1 Key mode settings View

To turn a grid into "clean View", in configuration you need:

1. **Disable startup**

```
autoLoad: false
```

2. Switch Intent Controller to **external**

```
data data: {
 controller: 'external',
 intents: ... //your handler
}
```

3. (Recommended) Enable global mode "read-only" so that any write attempts are prohibited by grid policy:

```
readOnly: true
```

Important: in ABGrid external mode **does not make network requests on its own**, until you call grid.load()/grid.refresh()/grid.request() by hand. Therefore, for View mode it is usually enough autoLoad: False and don't bother loading methods.

21.2 How to give data to a grid from Controller

The controller in your application receives data in any way (API, store, cache, calculations) and **updates View** through public methods:

- grid.setData(data) <TAG1> install data directly (locally).
- grid.setServerData(rows, total) <TAG1> set lines + total (convenient for server pagination).

Example (MVC):

```
import { ABGrid } from './path/to/grid'; // example

const grid = new ABGrid('#users', {
 autoLoad: false,
 readOnly: true,

 data data: {
 controller: 'external',

//Single Intent Point by View
intents: ({action, payload, grid }) => {
 switch (action) {

// pagination
 case 'pager: page': {
 const page = Number(payload.page);
//let's update the UI state (without downloading!)
grid.setPage(page, { load: false });
 // further decides Controller:
 controllerLoadPage(page);
 break;
        }

 case 'pager:rpp': {
 const rpp = Number(payload.rpp);
          grid.setRpp(rpp, { load: false });
 controllerChangeRpp(rpp);
 break;
        }

// sorting
 case 'sort: preview': {
// preview <TAG1>only update sort indicators (no download)
if (payload.sortOrder !== undefined) {
            grid.state.sortOrder = payload.sortOrder;
          }
 break;
        }
```

```
 case 'sort: apply': {
//apply — Controller decides how reload/recalculate data
 controllerApplySort(payload.sortOrder);
 break;
        }

// filter
case 'filter: apply': {
 controllerApplyFilter(payload.filter);
 break;
        }

 case 'filter: clear': {
 controllerClearFilter();
 break;
        }

 //toolbar
 case 'toolbar: refresh': {
 controllerReload();
 break;
 }

//In "clean view" mode, these actions are usually ignored
//or you make your reaction (for example, open an external dialogue).
 case 'toolbar: add':
 case 'toolbar:edit':
 case 'toolbar: delete':
 default:
 break;
 }
    }
  },

// ...other settings of columns/render/UI
});


//Controller: primary rendering
async function controllerInit() {
 const {rows, total } = await apiFetchUsers({ page: 1, rpp: 20
});
  grid.setServerData(rows, total);
  grid.setPage(1, { load: false });
  grid.setRpp(20, { load: false });
}

controllerInit();
```

21.3 List intents, which issues View (and which you intercept)

The current implementation provides for the following action:

Toolbar

- `toolbar: add`
- `toolbar:edit`
- `toolbar: delete`
- `toolbar: refresh`

Pager

- `pager: page`
- `pager:rpp`
- `pager: refresh`

Sort

- `sort: preview` (previews/indicators only)
- `sort: apply` (sorting confirmation)

Filter

- `filter: apply`
- `filter: clear`

21.4 Practical recommendations for clean View

- If you need it **strict View-only**, put `readOnly: true` and in the external controller **don't process it** `toolbar: add/edit/delete` (or show your toast “view only”).
- Always use: for server pagination
  - grid.setServerData(rows, total) + grid.setPage(page, {load:false})
- Don't call `grid.refresh()/grid.load()` in this mode, if you don’t want network activity from the grid (your Controller should work with the network).

22. ABGrid Engine — Events, Public API and Patterns of Use

22.1 ABGrid Event System (Event System)

ABGrid uses a built-in event system that allows you to respond to user actions and changes in the state of the grid.

The event is subscribed through the following method:

grid.on(eventName, handler)

Example:

```
grid.on('row:current', ({rowId, rowData}) => {
  console.log('Current string:', rowId);
});
```

22.2 Main events

row:current <TAG1> change the current line
data data:loaded <TAG1> data loaded
data data:error <TAG1> data loading error
crud:success <TAG1> successful operation CRUD
crud:error <TAG1> error CRUD operations
grid:ready <TAG1> grid fully initialized

22.3 Public API Grid

Basic public methods API grida:

22.3.1 Methods of working with data

grid.reload() — data reboot

grid.getRowData(rowId) — fetching string data

grid.getCurrentRowId() — get the current line

grid.getCheckedRowIds() — get selected lines

22.3.2 Toolbar methods

grid.toolbar.getValue(id) — get the value of the element
grid.toolbar.setValue(id, value) — set the element value

22.3.3 Methods of working with parts

grid.details.showForRow(rowId) — show panels for line
grid.details.clear() — clear panels

22.4 Patterns use of ABGrid

Master-Detail

```
const masterGrid = new ABGrid({container: '#orders' });
const detailGrid = new ABGrid({container: '#orderItems' });
masterGrid.on('row:current', ({rowId}) => {
  detailGrid.reload({orderId: rowId});
});
```

22.5 TreeGrid

Used to display hierarchical data.
Structure:
id
pid
name

22.6 Admin panel pattern

Typical structure:

- grid with list of entries
- toolbar with actions
- built-in record editor
- master-detail communications

23. ABGrid Engine — example implementations: Master → Detail → Detail

23.1 General interface architecture

This example demonstrates the typical structure of an administrative panel where several related grid are used.

Structure:

Users (master)
  <TAG1>
Licenses (detail)
↓
Product Versions (detail-detail)

Selecting a line in the parent grid automatically updates the child grid.

23.2 Master Grid — Users

```
const usersGrid = new ABGrid({
  container: '#usersGrid',
  schema: {
    id: {type: 'long', role: 'data' },
    email: {type: 'string', role: 'grid'},
    role: {type: 'string', role: 'grid'}
  },
  data: {
    request: {
      url: '/api/users'
    }
  }
});
```

23.3 Detail Grid — Licenses

```
const licensesGrid = new ABGrid({
  container: '#licensesGrid',
  schema: {
    id: {type: 'long', role: 'data' },
    userId: {type: 'long', role: 'data'},
    licenseKey: {type: 'string', role: 'grid'},
    status: {type: 'string', role: 'grid' }
  },
  data: {
```

```
    request: {
      url: '/api/licenses',
      params: (grid) => ({
        userid: usersGrid.getCurrentRowId()
      })
    }
  }
});
```

23.4 Detail-Detail Grid — Product Versions

```
const versionsGrid = new ABGrid({
  container: '#versionsGrid',
  schema: {
    id: {type: 'long', role: 'data' },
    licenseId: {type: 'long', role: 'data' },
    version: {type: 'string', role: 'grid'},
    releaseDate: {type: 'date', role: 'grid'}
  },
  data: {
    request: {
      url: '/api/productVersions',
      params: () => ({
        licenseId: licensesGrid.getCurrentRowId()
      })
    }
  }
});
```

23.5 Binding of the Grids (Update Cascade)

```
usersGrid.on('row:current', ({rowId}) => {
  licensesGrid.reload();
});

licensesGrid.on('row:current', ({rowId}) => {
  versionsGrid.reload();
});
```

23.6 Cascade cleaning of parts

When changing a line, it is important to clear the lower levels so that they are not displayed
outdated data.

Example:

```
usersGrid.on('row:current', ({ rowId }) => {
 licensesGrid.reload();
 versionsGrid.clear();
});
```

23.7 Result

This approach allows you to build powerful administrative interfaces:

• user management
• license management
• available versions of the product
• nested data dependencies

Important:
ABGrid automatically handles line selection, fine-grained mesh cleaning, and data loading, making complex interfaces much easier to implement.

24. Integration with React

24.1 Overview

ABGrid Engine provides the official adapter to work with React applications.

React binding is implemented as **thin wrapper**, which:

- mounts ABGrid in DOM
- manages the life cycle (init / destroy)
- allows you to update the configuration
- grants access to a Grid instance

Important:
ABGrid Engine remains **independent stateful engine**, and React is used only as an integration layer.

24.2 Installation

```
npm install abgrid-engine
```

24.3 Import

```
import ABGridReact from 'abgrid-engine/react';
```

24.4 Basic use

```
import React from 'react';
import ABGridReact from 'abgrid-engine/react';
const config = {
  columns: [
    {name: 'id', header: 'ID' },
    {name: 'name', header: 'Name' },
    {name: 'price', header: 'Price', summary: { page: ['avg'],
fractionDigits: 2 } }
  ],
  data: [
    {id: 1, name: 'Item 1', price: 10 },
    {id: 2, name: 'Item 2', price: 20 }
  ]
};
export default function App() {
 return (
 < ABGridReact config={config} />
  );
}
```

24.5 Receiving a specimen of the grid

```
import React, { useRef } from 'react';
import ABGridReact from 'abgrid-engine/react';

export default function App() {
 const gridRef = useRef(null);

 const handleReady = (grid) => {
 console.log('Grid ready:', grid);
  };

 return (
 < ABGridReact
 ref={gridRef}
 config={config}
 onReady={handleReady}
 />
  );
}
```

24.6 Update of data

If transmitted `data data`the component automatically updates the grid data:

```
< ABGridReact config={config} data data={rows} />
```

24.7 Life cycle

React adapter automatically:

- creates grid at mount
- destroys (destroy) at unmount
- recreates grid when changed config

24.8 Events

Events can be processed through the following:

```
< ABGridReact
 config={config}
 onReady={(grid) => {}}
 onRowClick={(e) => {}}
/>
```

(Supported events are consistent with ABGrid Engine events)

24.9 Important remarks

- Do not use React state to control each cell — ABGrid already manages the state
- Use it ref to call Grid methods
- Avoid frequent re-creation config (it is advisable to memoize)

25. Integration with Vue

25.1 Overview

ABGrid Engine provides the official adapter for Vue applications.

Vue harness:

- mounts grid to component
- manages the life cycle
- monitors configuration changes
- grants access to a Grid instance

25.2 Installation

```
npm install abgrid-engine
```

25.3 Import

```
import ABGridVue from 'abgrid-engine/vue';
```

25.4 Basic use

```
<template>
 < ABGridVue :config="config" />
</template>

<script>
import ABGridVue from 'abgrid-engine/vue';

export default {
  components: {ABGridVue},

  data() {
 return {
      config: {
        columns: [
          {name: 'id', header: 'ID' },
          {name: 'name', header: 'Name' },
          {name: 'price', header: 'Price', summary: { page:
['avg'], fractionDigits: 2 } }
```

```
        ],
        data: [
          {id: 1, name: 'Item 1', price: 10 },
          {id: 2, name: 'Item 2', price: 20 }
        ]
      }
    };
  }
};
</script>
```

25.5 Obtaining a copy of the grid

```
<template>
 < ABGridVue :config="config" @ready="onReady" />
</template>

<script>
export default {
  methods: {
    onReady(grid) {
 console.log('Grid instance:', grid);
    }
  }
};
</script>
```

25.6 Update of data

```
< ABGridVue :config="config" :data data="rows" />
```

25.7 Life cycle

VUE adapter automatically:

- creates grid at `mounted`
- destroys at `beforeUnmount`
- recreates when changed `config`

25.8 Events

```
< ABGridVue
  :config="config"
  @rowClick="onRowClick"
/>
```

25.9 Important remarks

- ABGrid manages his fortune, Vue <TAG1> shell only
- no need to try completely «react» grid
- use events and instance API

26 General recommendations

26.1 Architectural principle

React and Vue adapters are:

thin wrappers over ABGrid Engine

They:

- they do not duplicate business logic
- do not replace DataEngine
- do not interfere with the internal state

26.2 When to use adapters

Use React/Vue adapters if:

- the project has already been built on React or Vue
- quick integration without manual is required DOM-of the code

26.3 When to use vanilla ABGrid

Use ABGrid Engine directly if

- we need complete control
- no dependency on frameworks
- maximum performance required

27. Color palette

ABGrid Engine comes with 7 professional color themes. Users can develop and connect their own based on them. All color design is located in the folder themes. There are also global ones there CSS-files for everything html-pages that blend harmoniously with the color scheme of the component.

28. Delivery kit

The shipment includes all the necessary files and does not require any external libraries. ABGrid Engine distributed in several formats:

- ESM <TAG1> for modern projects and assemblers,
- UMD <TAG1> for universal connection,
- IIFE (abgrid.min.js) —for direct connection via <script> no pickers.

They are also located in the root folder logo files, icon file CSS.
In a folder themes 7 professional topics `abgrid-theme-*.csss` and 7 harmonious themes for the pages as a whole `page-skin-*.csss`

Listed below are the structure and delivery files.

1) Component files:

```
abgrid-engine/
   dist/
      abgrid.js
      abgrid.min.js
      abgrid.esm.js
      abgrid.umd.js
      abgrid-react.min.js
      abgrid-react.esm.js
      abgrid-react.umd.js
      abgrid-vue.min.js
      abgrid-vue.esm.js
      abgrid-vue.umd.js
      abgrid-structure.css
      abgid-icons.css
      abgrid-big-logo.png
      abgrid-big-short-logo.png
      abgrid-small-short-logo.png
  themes/
      abgrid-theme-blue-light.css
      abgrid-theme-blue-middle.css
      abgrid-theme-blue-pro.css
      abgrid-theme-dark-light.css
      abgrid-theme-dark-pro.css
      abgrid-theme-green-light.css
      abgrid-theme-green-pro.css
      page-skin-blue-light.css
      page-skin-blue-middle.css
      page-skin-blue-pro.css
```

```
    page-skin-dark-light.css
    page-skin-dark-pro.css
    page-skin-green-light.css
    page-skin-green-pro.css
  package.json
  README.ru.md
  README.en.md
  LICENSE.ru.md
  LICENSE.en.md
```

2) Developer's Guide (this document)