ABGrid Engine

Декларативный WEB-компонент
для работы со сложными табличными данными

Руководство разработчика
Версия 1.0
Версия компонента 1.0

Россия, Республика Бурятия
(c) А. Батурин, 2026г

Содержание

Введение 7
1. Ключевые особенности продукта 8
 1.1 Что решает ABGrid Engine 8
 1.2 Чего ABGrid Engine осознанно не делает 9
 1.4 Философия конфигурации 9
 1.4 Для каких проектов предназначен компонент 9
 1.5 Базовые термины 10
2. Архитектура ABGrid Engine 11
 2.1 Жизненный цикл 11
 2.2 Роль GridCore 12
 2.3 Параметр конфигурации schema 12
 2.4 Рекомендации по описанию полей 12
 2.5 Использование режимов create\edit 13
 2.6 Использование ролей полей 13
 2.7 Общие рекомендации по проектированию системы 14
 2.8 Система плагинов ABGrid Engine 15
  2.8.1 Подключение плагина 15
  2.8.2 Структура плагинов 16
  2.8.3 Использование событий 16
  2.8.4 Добавление собственного API 17
  2.8.5 Использование нескольких плагинов 18
  2.8.6 Когда стоит создавать плагин 18
  2.8.7 Рекомендации по разработке плагинов 18
  2.8.8 Пример пользовательского плагина 21
3. Подключение файлов на HTML-странице 23
4. Зарезервированные имена и служебные поля 24
5. Формат обмена данными с сервером 25
6. Рекомендации по архитектуре 26
7. Быстрый старт 27
 7.1 Минимальная инициализация 27
 7.2 Что происходит при создании грида 28
 7.3 Загрузка данных 29
 7.4 Формат ответа сервера (кратко) 29
 7.5 Минимальный жизненный цикл 30
8. Типы полей 31
 8.1 Типы полей 31
 8.2 Типы полей редактора 33
  8.2.1 Подробней о типе «select» 33
  8.2.2 Подробней о типе «autocomplete» 34
   8.2.2.1 Формат ответа сервера 38
   8.2.2.2 Примечение по autocomplete.dataKey 39
  8.2.3 Подробнее о типе «confirm» 40

8.3 Типы колонок рендера 41
8.3.1 Подробно о типе колонок «actions» 42
9. Конфигурация компонента 44
9.1 Общая структура конфигурации 44
9.2 readOnly 46
9.3 autoLoad 46
9.4 Итоговые строки 46
9.5 Политика разрешений 48
9.6 Глобальная политика при ответе сервера со статусом 401 48
9.7 Раздел data 50
9.7.1 Контроллер данных: internal и external 50
9.7.2 Формат обмена с сервером 53
9.8 Раздел crud 54
9.9 Массив детальных сеток detailGrids 54
9.10 Массив детальных панелей detailsPanels 55
9.11 UI-хелперы 63
9.12 Локализация (i18n) 64
9.13 Раздел editor 67
9.14 Раздел schema 67
9.14.1 Порядок, видимость и ширина колонок 68
9.14.2 Переопредление свойств в editor 69
9.15 Режим отладки конфигурации 69
9.16 Раздел view 70
9.16.1 Выбор строк, размеры, заголовок 71
9.16.2 Сортировка и мультисортировка 72
9.16.3 Панель инструментов 73
9.16.4 Параметры доступности кнопок панели инструментов 77
9.16.5 Встроенный пейджер 79
9.16.6 Управление простарнством 80
9.16.7 Subgrid 81
9.16.8 TreeGrid 82
9.17 Конфигурация по умолчанию (DEFAULT_options0 83
10. Политика разрешений (policy) подробно 92
11. Editor и schema подробно 93
11.1 Общая идея schema 93
11.2 Глобальные настройки Editor 93
11.3 Структура schema 94
11.4 Поля и их свойства 94
11.5 Переопределение в режимах create\edit 95
11.6 Валидация в редакторе 95
11.6.1 Числовые ограничения 96
11.6.2 Проверка по регулярному выражению 97
11.6.3 Пользовательская валидация 99

11.7 Загрузка файлов из формы редактирования 100
11.8 Практический пример schema 104
12. Протокол обмена с сервером 105
12.1 dataKey:обёртка запроса 105
12.2 Загрузка данных (load) 105
12.3 CRUD-операции и Result 106
13. Публичный API 108
13.1 Жизненный цикл 108
13.2 Работа с данными 108
13.3 CRUD API 108
13.4 Работа с фильтрами и сортировкой 108
13.5 Навигация и UI 109
13.6 События и обработчики 109
13.7 Публичный API (подробно) 110
14. Интерцепторы (request\response\error) 119
14.1 Интерцепторы запросов (request) 120
14.2 Интерцепторы ответов (response) 121
14.3 Интерцепторы ошибок (error) 123
14.4 Интерцепторы error. Рекомендации 123
14.5 Итог 125
15. Режим Master / Detail подробней 126
15.1 Сочетание режимов 126
16. Режим ReadOnly 127
16.1 Назначение 127
16.2 Как включить ReadOnly 127
16.3 Что разрешено в режиме ReadOnly 127
16.4 Что запрещено в режиме ReadOnly 128
16.5 ReadOnly и UI 128
16.6 ReadOnly и Editor 128
16.8 Рекомендации использования 128
17. UI-intents 130
17.1 Список стандартных intents 130
17.2 Чего intents не делают 131
18. Глобальные статические helpers 132
19. Типовые сценарии использования 133
19.1 Master + Detail (классический справочник) 133
19.2 Master + внешняя форма редкатирования 133
19.3 Использование external controller 134
19.4 Грид в режиме ReadOnly 134
19.5 TreeGrid (иерархический справочник) 135
19.6 Использование системных диалогов в пользовательском коде 136
19.6.1 Диалог сообщений msg() 136
19.6.2 Диалог подтверждения confirm() 138

19.6.3 Диалог подтверждения confirmEx() 140
19.6.4 Рекомендации по использованию 144
19.7 Уведомления (toast) 145
19.7.1 Метод toast() 145
19.7.2 Примеры использования 146
19.7.3 Использование в асинхронных сценариях 146
19.7.4 Использование совместно с событиями грида 147
19.8 Когда использовать toast(), а когда msg\confirm 148
19.8.1 Диалоги и уведомления: UX-рекомендации 148
19.8.2 Итог 148
19.9 Autocomplete для пользовательских input 149
20. Работа с внутренним HTTP-клиентом 151
20.1 Отправка запроса через grid.data.request() 151
20.2 Использование buildRequestPayload() 152
20.3 Альтернативный способ – grid.request() 152
21. Работа в режиме View 155
21.1 Ключевые настройки режима View 155
21.2 Как отдавать данные гриду из Controller 156
21.3 Список intens, которые эмитит View 158
21.4 Практические рекомендации для чистого View 158
22. ABGrid Engine-Events,Public API и Patterns использования 159
22.1 Система событий ABGrid (Event System) 159
22.2 Основные события 159
22.3 Public API Grid 159
22.3.1 Методы работы с данными 159
22.3.2 Методы панели инструментов 160
22.3.3 Методы работы с деталями 160
22.4 Patterns использования ABGrid 160
22.5 TreeGrid 160
22.6 Admin panel pattern 160
23. ABGrid Engine-пример реализации Master-Detail-Detail 161
23.1 Общая архитектура интерфейса 161
23.2 Master Grid - Users 161
23.3 Detail Grid - Licenses 161
23.4 Detail-Detail Grid – Product Verdions 162
23.5 Связывание гридов (каскад обновления) 162
23.6 Каскадная очистка деталей 162
23.7 Результат 163
24. Интеграция с React 164
24.1 Обзор 164
24.2 Установка 164
24.3 Импорт 164
24.4 Базовое использование 164

24.5 Получение экхемпляра грида 165
24.6 Обновление данных 165
24.7 Жизненный цикл 165
24.8 События 165
24.9 Важные замечания 166
25. Интеграция с Vue 167
25.1 Обзор 167
25.2 Установка 167
25.3 Импорт 167
25.4 Базовое использование 167
25.5 Получение экземпляра грида 168
25.6 Обновление данных 168
25.7 Жизненный цикл 168
25.8 События 169
25.9 Важные замечания 169
26. Общие рекомендации 170
26.1 Архитектурный принцип 170
26.2 Когда использовать адаптеры 170
26.3 Когда использовать vanilla ABGrid 170
27. Цветовая палитра 171
28. Комплект поставки 172

Введение

ABGrid Engine предназначен для работы со сложными табличными данными в WEB-приложениях. Базовая архитектура компонента берёт свое начало в 2013г, когда автором на основе известного табличного jQuery-плагина был разработан грид-компонент для визуальной среды программирования на языке PHP. Во время разработки выявились сложности и необходимость написания однообразного кода для использования в режимах мастер-деталь, в режиме отображения каких-то дополнительных данных в той же таблице ниже основной строки данных и некоторые другие проблемы, что побудило создать в конечном итоге простую и логичную архитектуру компонента, не требующего ручного повторения однообразного кода. В дальнейшем было принято решение о создании собственного табличного jQuery-плагина, который был реализован, но использовался только для собственных нужд. В конце 2025г принято решение о ребрендинге и переводе компонента на чистый язык Javascript и CSS, что и было сделано. В настоящее время ABGrid Engine самодостаточен и не имеет каких-либо зависимостей от внешних библиотек. Продукт получил немало новых модулей для удобства разработчиков, таких как внутренний редактор записей, autocomplete и возможность его подключения к пользовательским input, возможность работы только как View, использование разработчиком системных диалогов и уведомлений, управление рабочим пространством и многое другое, однако подход остался прежний - минимум ручного написания кода конечным пользователем-программистом.

Автор

1. Ключевые особенности продукта

ABGrid Engine — это универсальный grid-компонент, ориентированный на разработку сложных бизнес-приложений, движок с высокой степенью конфигурируемости и минимальным объёмом пользовательского кода. Компонент построен на основе декларативного подхода к решению сложных задач. Большинство задач в рамках ABGrid Engine решается на уровне задания конфигурации и не требует написания дополнительного кода.

Ключевая идея ABGrid Engine:
«Сложное — просто. Вы делаете конфигурацию — ABGrid Engine всё остальное».

1.1 Что решает ABGrid Engine

ABGrid Engine предназначен для решения типовых и сложных задач работы с табличными данными:

- загрузка данных с сервера
- отображение и форматирование строк и ячеек
- редактирование данных (create / update) встроенным редактором
- настраиваемая валидация редактируемых данных для минимизации невалидного серверного трафика
- удаление записей
- постраничная серверная навигация
- работа с детальными сетками и субгридами в автоматическом режиме
- отображение древовидных структур данных в автоматическом режиме
- встроенный механизм autocomplete для редактирования данных
- подключение механизма autocomplete компонента к пользовательским input
- централизованное управление правами и режимами доступа
- управление пространством других ABGrid-компонентов
- неограниченная иерархия master\detail, subgrid, treegrid и при этом весь процесс происходит полностью в автоматическом режиме.
- предоставляет пользователю возможность использования системных диалогов в своем коде
- все системные диалоги и уведомления ABGrid Engine:
    - изолированы по экземплярам грида
    - не конфликтуют между master/detail/subgrid
    - безопасны при наличии нескольких гридов на странице
    - изолированы от пользовательского кода при их применении

- предоставляет пользователю возможность использования системного вывода сообщений

1.2 Чего ABGrid Engine осознанно не делает

ABGrid Engine не является:
• ORM-системой
• заменой серверной валидации
• системой управления пользователями и ролями

1.3 Философия конфигурации

В ABGrid Engine отсутствует необходимость:
• вручную писать обработчики CRUD-операций
• управлять состоянием формы редактирования
• синхронизировать детальные и субгрид-сетки вручную.
Поведение объекта компонента описывается конфигурацией, а все внутренние переходы состояний выполняются движком.

1.4 Для каких проектов предназначен компонент

ABGrid Engine подходит для:
• административных панелей
• корпоративных систем
• ERP / CRM решений
• внутренних бизнес-интерфейсов

Компонент масштабируется от простых таблиц до сложных иерархических структур с вложенными сетками и детальными таблицами. Все детальные компоненты и т.н. субгриды являются полноценными компонентами ABGrid Engine, все взаимосвязи между которыми работают автоматически на основе заданной конфигурации. Детальные таблицы могут иметь как свои субгриды, так и свои детальные таблицы, при этом никаких ограничений на уровень иерархии нет, всё зависит от задач и фантазии разработчика.

1.5 Базовые термины

В документации используются следующие термины:

Грид или grid — экземпляр ABGrid Engine, связанный с DOM-контейнером.
Master-grid, мастер-сетка — основная сетка.
Detail-grid — детальная сетка, связанная с выбранной строкой мастер-сетки.
Subgrid — вложенная сетка, открываемая внутри строки.
TreeGrid-режим работы компонента в режиме отображения древовидных структур данных.
Editor — форма создания или редактирования записи.
Policy — политика прав доступа, применяемая к CRUD-операциям.
readOnly — режим работы грида «только просмотр».

2. Архитектура ABGrid Engine

ABGrid Engine построен по модульной архитектуре. Основная цель такой архитектуры — разделение ответственности между компонентами и возможность независимого развития каждого модуля.

Основные принципы архитектуры:

• GridCore — центральный координатор грида
• DataEngine — загрузка и обновление данных
• CrudEngine — операции create / update / delete
• TreeEngine — поддержка древовидных структур
• DetailsPanelsEngine — управление детальными панелями
• ToolbarEngine — элементы панели инструментов
• EditFormService — встроенный редактор записей

GridCore связывает все подсистемы и обеспечивает единый API для разработчика.
Каждый модуль отвечает только за свою область функциональности.

Такая архитектура позволяет:
• расширять функциональность без изменения ядра
• подключать новые возможности постепенно
• облегчать тестирование и поддержку кода

2.1 Жизненный цикл

Работа грида проходит через несколько этапов:

1. Создание экземпляра грида
2. Нормализация конфигурации
3. Инициализация модулей
4. Загрузка данных
5. Рендеринг таблицы
6. Подключение обработчиков событий

Этот жизненный цикл позволяет гибко подключать новые модули и расширения.

2.2 Роль GridCore

GridCore — центральный объект системы.

Он выполняет следующие задачи:
• хранит конфигурацию options
• управляет состоянием грида
• инициирует загрузку данных
• управляет событиями
• связывает все модули системы

Большинство публичных методов API вызываются именно через объект grid.

2.3 Параметр конфигурации schema

Schema определяет структуру данных грида и используется сразу в нескольких местах:
• отображение колонок
• редактор записей
• валидация данных
• преобразование типов
• роли отображения данных

Поэтому schema является центральной частью конфигурации.

2.4 Рекомендации по описанию полей

При описании schema рекомендуется:
• использовать понятные имена полей
• всегда указывать тип поля (type)
• использовать editor только там, где требуется редактирование
• добавлять hint для подсказок пользователю
• явно задавать required для обязательных полей

Пример:

```
schema: {
  name: {
    type: 'string',
```

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

2.5 Использование режимов create / edit

В ABGrid можно задавать разные настройки поля для режимов создания и редактирования.

Пример:

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

Это позволяет гибко управлять поведением формы редактора.

2.6 Использование ролей полей

Параметр role определяет, где используется поле.

Например:

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

Это позволяет:

• скрывать поля из таблицы
• использовать поля только в редакторе
• хранить служебные данные

2.7 Общие рекомендации по проектированию схемы

Чтобы конфигурация грида оставалась понятной и масштабируемой:

• не перегружайте schema большим количеством логики
• используйте единый стиль описания полей
• группируйте похожие поля
• используйте комментарии для сложных мест
• старайтесь сохранять одинаковую структуру editor для всех полей

2.8 Система плагинов ABGrid Engine

ABGrid Engine поддерживает расширение функциональности через **плагины**.

Плагин — это обычный JavaScript-класс, который получает доступ к экземпляру грида и может:

- подписываться на события
- добавлять новые методы
- взаимодействовать с данными
- изменять поведение компонента

Плагины позволяют расширять возможности ABGrid Engine **без изменения исходного кода библиотеки**.

Это означает, что разработчик может создавать собственные расширения, имея только **собранную библиотеку ABGrid**.

2.8.1 Подключение плагина

Плагин подключается при создании грида через параметр `plugins`.

Пример:

```
class ExamplePlugin {

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

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

}

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

    plugins: [
        ExamplePlugin
    ]

});
```

При создании грида ABGrid:

1. создаёт экземпляр каждого плагина
2. передаёт ему объект `grid`
3. вызывает метод `init()`.

2.8.2 Структура плагина

Минимальная структура плагина:

```
class MyPlugin {

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

    init() {

        // код инициализации

    }
}
```

Элемент	Назначение
constructor(grid)	получает экземпляр грида
init()	вызывается после инициализации грида

Через `this.grid` плагин получает доступ ко всем возможностям компонента.

2.8.3 Использование событий

Плагин может подписываться на события грида.

Пример:

```
class RowLoggerPlugin {

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

    init() {

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

    }
}
```

Таким образом можно реагировать на:

- загрузку данных
- выбор строк
- действия пользователя
- CRUD-операции

2.8.4 Добавление собственного API

Плагин может добавлять новые методы в экземпляр грида.

Пример:

```
class ExportPlugin {

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

    init() {

        this.grid.exportCSV = () => {

            const rows = this.grid.getRows();

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

        };
    }
}
```

После этого метод можно использовать:

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

2.8.5 Использование нескольких плагинов

Можно подключить несколько плагинов одновременно:

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

    plugins: [
        ExamplePlugin,
        RowLoggerPlugin,
        ExportPlugin
    ]

});
```

Плагины инициализируются **в порядке их подключения**.

2.8.6 Когда стоит создавать плагин

Плагины рекомендуется использовать для:

- дополнительной логики интерфейса
- интеграции с другими библиотеками
- расширения API грида
- автоматизации действий пользователя
- добавления новых UI-модулей

Такой подход позволяет расширять функциональность компонента **без изменения его исходного кода**.

2.8.7 Рекомендации по разработке плагинов

При создании собственных плагинов для ABGrid Engine рекомендуется соблюдать несколько правил, чтобы обеспечить совместимость с будущими версиями компонента и избежать конфликтов с другими плагинами.

Используйте только публичный API

Плагин должен взаимодействовать с компонентом только через **публичные методы и события грида**.

Не рекомендуется обращаться напрямую к внутренним свойствам объекта grid.

Например:

✓ корректно

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

✕ нежелательно

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

Внутренние свойства могут изменяться между версиями компонента.

Не изменяйте внутренние структуры данных

Плагин не должен напрямую модифицировать:

- внутренние массивы данных
- конфигурацию колонок
- состояние грида

Все изменения должны выполняться через API грида.

Например:

✓ корректно

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

✕ нежелательно

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

Используйте собственные пространства имён

Если плагин добавляет методы или свойства в объект grid, рекомендуется использовать уникальные имена.

Пример:

✓ рекомендуется

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

✕ нежелательно

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

Это помогает избежать конфликтов с будущими версиями ABGrid Engine или другими плагинами.

Подписывайтесь на события вместо изменения поведения

Если необходимо реагировать на действия пользователя или изменения данных, лучше использовать систему событий.

Пример:

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

Такой подход делает плагин независимым от внутренней реализации компонента.

Избегайте изменения DOM вне контейнера грида

Плагин должен работать только внутри DOM-контейнера, принадлежащего гриду.

Не рекомендуется изменять элементы страницы вне области грида, если это не требуется логикой приложения.

Минимизируйте зависимости

Желательно, чтобы плагины не требовали подключения дополнительных библиотек или фреймворков.

Это упрощает использование плагинов и уменьшает вероятность конфликтов.

Совместимость с будущими версиями

При разработке плагинов рекомендуется учитывать, что:

- внутренние механизмы ABGrid Engine могут изменяться
- публичный API остаётся стабильным

Поэтому плагины должны опираться только на **документированные возможности компонента**.

2.8.8 Пример пользовательского плагина

Ниже приведён простой пример плагина, который добавляет новый метод в API грида.
Метод выводит в консоль количество строк, загруженных в таблицу.

Этот пример демонстрирует основной принцип работы плагинов — **расширение возможностей экземпляра грида**.

Реализация плагина

```
class RowsInfoPlugin {

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

    init() {

        this.grid.printRowsCount = () => {

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

        };
    }
}
```

Подключение плагина

Плагин подключается при создании грида через параметр `plugins`.

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

    plugins: [
        RowsInfoPlugin
```

```
    ]
});
```

Использование плагина

После подключения плагина новый метод становится доступен через объект грида.

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

В консоль будет выведено количество строк, загруженных в таблицу.

3. Подключение файлов на HTML-странице

Рекомендуемый вариант подключения:

```
<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>
```

Использование:

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

Пример минимального подключения (ES-модули):

```
<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. Зарезервированные имена и служебные поля

Некоторые имена используются движком для служебных целей и не рекомендуются как пользовательские имена полей:

- id – уникальный идентификатор строки (первичный ключ), в edit всегда read-only. Поле, однозначно идентифицирующее строку. При необходимости сервер может сформировать данное поле как результат конкатенации нескольких первичных полей или их hash-результат. Компонент всегда использует для идентификации строк только одно это поле.
- __confirmMismatchTpl (шаблон сообщения для confirm-полей)
- Тип колонки actions (виртуальная колонка рендера)
- policy (в envelope ответа сервера: data.policy)

5. Формат обмена данными с сервером

ABGrid Engine поддерживает разные форматы представления строк при загрузке и отправке данных.

- inFormat (data.inFormat):
- auto — определить формат автоматически
- object — массив объектов (рекомендуется)
- aoa — Array of Arrays (строки как массивы значений)
- csv — CSV-строка (delimiter задаётся data.csvDelimiter)

outFormat (data.outFormat): object | aoa | csv — формат исходящих данных (CRUD и пользовательские запросы).

В режиме «auto» формат входных данных распознается по первой строке данных строк.

Форматы обмена задаются в параметре data. Например,

```
data:{
        …
  // форматы строк
    inFormat: 'auto', // 'auto'|'aoa'|'object'|'csv'
  // формат исходящих данных (CRUD, пользовательские запросы): 'aoa'|'object'|'csv'
    outFormat: 'object',
    csvDelimiter: ';'
}
```

6. Рекомендации по архитектуре

• Используйте internal controller для простых CRUD-сценариев
• Используйте external controller для сложных форм и бизнес-логики
• Все бизнес-валидации должны дублироваться на сервере
• Не модифицируйте данные grid напрямую — используйте API

7. Быстрый старт

Раздел «Быстрый старт» демонстрирует минимальный набор шагов, необходимых для инициализации и использования ABGrid Engine. Цель раздела — показать базовый принцип работы компонента без углубления в расширенные настройки.

7.1 Минимальная инициализация

Экземпляр компонента создаётся вызовом конструктора ABGrid(container, options), где container — CSS-селектор или DOM-элемент контейнера. Параметр root внутри options не используется. В качестве контейнера настоятельно рекомендуется использовать аттрибут «id» div-элемента.
Для создания грида достаточно указать DOM-контейнер и конфигурацию. В ABGrid Engine колонки грида строятся из schema.fields. Там же описываются правила для редактора (required/readOnly по режимам, значения enum/autocomplete и т.д.).

Пример минимальной инициализации:

```
const grid = new ABGrid('#grid', {
      data: {
        url: '/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 Что происходит при создании грида

При создании экземпляра ABGrid Engine:
- происходит привязка грида к DOM-контейнеру
- инициализируется внутреннее состояние
- подготавливаются механизмы загрузки данных
- регистрируются интерцепторы запросов и ответов

Сам компонент не отправляет запросы на сервер, пока не будет явно вызвана загрузка данных или не сработает автозагрузка при установленном параметре autoLoad=true.

7.3 Загрузка данных

Для загрузки данных используется метод load():
grid.load();

Метод load:
• отправляет запрос на сервер
• получает данные в стандартном формате ответа
• обновляет содержимое таблицы
• устанавливает текущую строку при наличии данных
• обновляет связанные детальные таблицы
• устанавливает политику прав CRUD при наличии разрешения в конфигурации и при наличии самой политики в ответе сервера

7.4 Формат ответа сервера (кратко)

Сервер обязан передавать ответ в формате JSON и соблюдать стандарт ответа для успешной обработки его компонентом, ответ должен быть объектом со следующими параметрами (например, для операции загрузки):

```
{
    success: true | false,   // результат выполнения операции
    message: ' OK',          // сообщение
    data:{
        rpp:10,              // подтверждние запроса rpp
        page:1,              // подтверждние запроса page
        totalRecords:97,     // общее количество записей
// строки, формат задается параметром data.inFormat
        rows: []
    }
}
```

Дополнительные поля (policy и др.) обрабатываются автоматически, если присутствуют.

7.5 Минимальный жизненный цикл

Типовой жизненный цикл грида:
1. Создание экземпляра ABGrid Engine
2. Вызов load()
3. Получение данных с сервера
4. Отображение строк
5. Установка текущей строки
6. Очистка и загрузка детальных сеток

Дальнейшие действия (редактирование, детализация, навигация) описываются конфигурацией и не требуют дополнительного кода.

8. Типы полей

В ABGrid Engine тип данных поля, тип поля в редакторе и тип поля в гриде — это разные сущности и задаются в соответствующих параметрах конфигурации.

8.1 Типы полей данных (schema.fields.<name>.type)

Определяет тип данных.

Тип	Назначение	Примечание
text	Строковые данные	По умолчанию
number,int	Числовые данные	Приводится к number
bool,boolean	Логический тип	true\false
date	Дата	Требует строгий формат YYYY-MM-DD
datetime	Дата и время	Требует строгий формат YYYY-MM-DD HH:mm
json	Произвольный объект	Без преобразований
password	Для скрытия паролей	Параметр showPasswordToggle: true для возможности показа\скрытия

Используется **только** для:

- нормализации значений
- валидации
- форматов обмена с сервером

❌ **НЕ влияет** на:

- выбор редактора
- отображение в гриде

Важно о enum:

- type: "enum" **НЕ выбирает редактор**
- enum — это **тип валидации**, а не UI
- Допустимые значения берутся из schema.fields.<f>.values.enum

Поддерживаемые форматы values.enum:

1) Object-map (рекомендуемый, самый простой)

```
values: {
   enum: {
      M: "Мужской",
      F: "Женский"
   }
}
```

- key → значение, которое хранится в данных
- value → отображаемый текст

2) Массив строк / чисел

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

- и значение, и отображаемый текст = одно и то же
- подходит для простых случаев

3) Массив объектов { value, text }

```
values: {
   enum: [
      { value: "M", text: "Мужской" },
      { value: "F", text: "Женский" }
   ]
}
```

- явное разделение value и text
- полезно, если нужен расширяемый формат

Как это используется системой

В редакторе (editor.type = "select")

- value → то, что записывается в поле
- text → то, что видит пользователь

В гриде

- отображается **text**
- независимо от того, как задан enum

В валидаторе (type: "enum")

- проверяется, что **value входит в список допустимых**
- сравнение идёт **по строковому представлению** (даже если enum задан числами)

Во всех случаях ABGrid Engine приводит enum к единому внутреннему формату `{ value, text }`.

8.2 Типы полей редактора (schema.fields.<name>.editor.type)

Тип поля определяет, какой HTML-контрол будет создан во встроенной форме редактирования. Ниже перечислены типы, поддерживаемые в текущей версии.

Тип	Назначение
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"> Выбор файла для загрузки на сервер
tel	<input type="tel">
textarea	<textarea></textarea>
select	<select> <option value="...">...</option> </select>
autocomplete	<input type="text"> Поиск по удаленному справочнику встроенным модулем Autocomplete
confirm	<input type="text"> Подтверждение для основного поля, основное поле как правило с типом string в schema.fields.<fname>.type

Правила использования:

- Редактор **выбирается ТОЛЬКО по editor.type**
- values.enum **сам по себе ничего не включает**
- values.autocomplete **сам по себе ничего не включает**
- Если editor.type не указан → используется text
-

Если поле содержит файл, CRUD-запрос автоматически отправляется как: multipart/form-data
Файл передается в поле: file_<fieldName>

8.2.1 Подробнее о типе «select»

Создается HTML-элемент «select».

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

Источник данных для select — **schema.fields.<field>.values.enum**.

Это обязательный параметр для editor.type = "select".

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

Поддерживаемые форматы values.enum

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

```
values: {
  enum: {
    M: "Мужской",
    F: "Женский"
  }
}
```

HTML будет выглядеть так:

```
<select>
  <option value="M">Мужской</option>
  <option value="F">Женский</option>
</select>
```

- value → значение, которое **сохраняется в поле**
- text → текст, который **видит пользователь**

✔ Рекомендуемый формат для документации и реальных проектов.

8.2.1.2 Массив объектов { value, text }

```
values: {
  enum: [
    { value: "M", text: "Мужской" },
    { value: "F", text: "Женский" }
  ]
}
```

HTML:

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

Используется, если нужен более расширяемый формат (например, в будущем с доп. атрибутами).

8.2.1.3 Массив строк или чисел

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

или

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

HTML:

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

Важно понимать поведение:

- value и text **одинаковы**
- числа **приводятся к строке** в HTML
- это допустимо, но **неудобно для пользователя**, если нужны человекочитаемые подписи

✓ Подходит для простых технических значений
✕ Не рекомендуется для UI с подписями

Что происходит при выборе значения

При выборе пункта:

- в поле записывается **value**
- тип данных значения определяется **schema.fields.<field>.type**

Пример:

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

- пользователь выбирает 2
- в <select> значение "2" (строка)
- модель приводит значение к number
- в данных будет 2 (число)

8.2.1.4 Поведение при отсутствии values.enum

Если указано:

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

но **values.enum отсутствует или пуст**, то:

- <select> будет пустым
- выбор невозможен
- это считается **ошибкой конфигурации**

Рекомендуется явно указывать values.enum всегда.

8.2.1.5 Связь с валидацией (values.enum)

Если дополнительно указано,то при сохранении:

- значение проверяется на принадлежность values.enum
- сравнение выполняется **по строковому представлению**

не влияет на HTML

- **не включает select автоматически**
- используется **только для валидации**

8.2.2 Подробнее о типе «autocomplete»

Данный тип редактирования в редакторе указывается для выбора значений из удаленного справочника на сервере, когда предыдущий способ становится крайне неудобным для пользователя. Как правило, это происходит уже при относительно длинном списке вариантов. В таком случае удобнее использовать данный тип редактирования. Движок компонента предоставляет упрощенный механизм, предусматривающий наличие в строке данных 2-х полей - одно скрытое от пользователя поле для идентификатора (кода), второе - удобное для прочтения пользователем.

Autocomplete может быть задан глобально (options.editor.autocomplete) и локально в поле (field.editor.autocomplete). Локальная настройка поля переопределяет глобальную только для этого поля.

Глобальная конфигурация по умолчанию:

```
autocomplete: {
  url: null,
  dataKey: null,        //  Ключ данных запроса для сервера
  queryParam: 'q',      //  строка поиска
  fieldParam: 'field',  //  параметр для указания поля поиска
значения
  minChars: 3,      //  минимальное кол-во символов для
срабатывания запроса
  debounceMs: 250,   //  защитная пауза в мс
  limit: 30,         //  ограничение вариантов (для сервера)
  strict: true      //  запрещать ввод значений вне списка
 }
```

Пример локальной настройки:

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

Параметр displayField указывает редактору место, куда надо прописать текстовое значение при выборе. Если не нужны локальные переопределения, то достаточно указать displayField и editor:{ type: 'autocomplete' }.

8.2.2.1 Формат ответа сервера

Сервер обязан отдавать ответ в следующем формате:

```
{
   success: true,

// при success: true допускается пустая строка,
//при success: false обязательный непустой параметр для
отображения пользователю
   message: "",
   data: {
     items: [
       { id: 123, value: "ООО Ромашка" },
              ….
       { id: 124, value: "АО Василёк" }
     ]
   }
}
```

Важно:

- id — обязателен (любой тип, но не null/undefined)
- value — обязателен (приводится к строке)
- любые другие поля можно добавлять — они сохраняются как raw, но UI показывает value.

При отсутствии вариантов сервер обязан установить success:false и в message строку сообщения о неудачном поиске (в этом случае параметр data необязателен), например,

```
{
   success: false,
   message: 'Не найдено'
}
```

При неудачном поиске именно значение параметра message отображается пользователю и при этом выбор запрещен.

8.2.2.2 Примечание по autocomplete.dataKey

Параметр dataKey добавляется к запросу autocomplete как query-параметр `dataKey`. Это позволяет использовать один endpoint для разных источников подсказок (например: companies, users, products).

Пример запроса:

```
/api/autocomplete?q=abc&field=companyName&limit=30&dataKey=com
panies
```

8.2.3 Подробнее о типе «confirm»

ABGrid Engine предоставляет возможность при редактировании записей внутренним редактором во встроенной форме автоматически отображать и обрабатывать поле подтверждения введенного значения в основном поле для уверенности в том, что пользователь действительно ввел правильное значение в основное поле и запомнит его при необходимости. Это используется при изменении, например, пароля или ввода значения электронной почты и т.д. Эти поля имеют тип type = «confirm». Данные поля имеют смысл только в слое UI, не существуют в объекте строки и никак не участвуют в обмене с сервером. Поля с типом «confirm» наследуют свойства основного поля, поэтому в конфигурации достаточно указать только уникальные свойства именно такого типа поля, а именно его тип type: 'confirm' и confirmOf, которое должно содержать имя основного поля, значения в котором и подтверждается полем «confirm». При вводе пользователем разных значений движок не позволит сохранить данные внутренней формы редактирования и отобразит соответствующее сообщение.

Приведем практический пример. В конфигурации прописываем 2 поля для изменения пароля – основное поле «password» и поле подтверждения «confirmPassword».

Пример с полем типа «confirm».

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

8.3 Типы колонок рендера (fields.<field>.ui.colType)

Тип колонки рендера управляет тем, как значение отображается в таблице. Это независимая подсистема от editor.type. Тип конкретного столбца указывается в свойстве самого поля в параметре ui.grid.

Типы полей:

- text (по умолчанию)
- tree
- actions (виртуальная колонка)
- image
- link
- badge
- select
- autocomplete
- boolean
- progress
- rating
- date
- datetime
- currency

Пример рендера колонки с типом boolean:

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

Если у поля задан values.enum, ABGrid Engine по умолчанию трактует колонку как select и отображает label вместо кода. Чтобы показывать исходное значение, задайте ui.grid.colType: "text" или свой ui.grid.renderer.

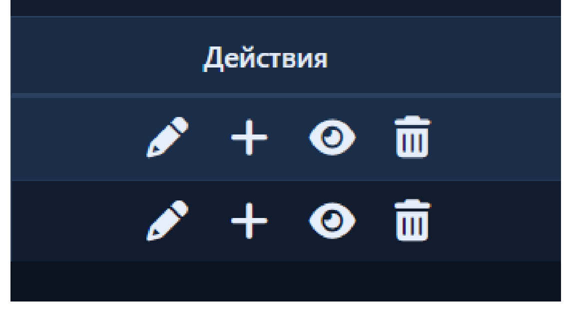
8.3.1 Подробно о типе колонок «actions»

ABGrid Engine позволяет создавать пользовательские колонки с типом colType= «actions». В таких столбцах рендерятся кнопки на каждой строке для каких-либо действий по строке, например, для редактирования, удаления и т.д. Важно понимать, что это именно пользовательские колонки, в обмене с сервером не участвуют, в объекте строки не существуют и при передаче объекта строки в функции не передаются. Движок компонента их распознает по типу colType= «actions» и действует по своему внутреннему алгоритму. Количество колонок и их порядок размещения произвольный. Действия в колонке рендерятся как элементы «button» по определённому внутреннему алгоритму.

Пример колонки с типом 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'}
                }
          }
     }
}
```

Отображение на странице:



Обработчик клика назначается программно, например, так:

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

Имя действия формируется из первого параметра actions.*: { …..}, где * - параметр конфигурации.
Параметр action – это имя свойства из конфигурации, т.е. (см. пример выше) «edit», «addsubctg», «publish», «trash».

Остальные параметры:

- row – объект строки,
- ctx – контекст
- grid – инстанс конкретного компонента ABGrid Engine

Свойства:

- classes – пользовательский класс иконки для отображения
- aria-label - текст для аттрибута aria-label кнопки действия
- hint - текст подсказки при наведении курсора мыши на кнопку(title)
- raw – пользовательские аттрибуты. Рендерятся как есть.

Для задания пользовательских аттрибутов используйте параметр raw, например:

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

В этом примере вместо иконок fontawesome задаются иконки UIKit.

9. Конфигурация компонента

Конфигурация ABGrid Engine описывает поведение грида декларативно. Все основные возможности компонента настраиваются через объект options, передаваемый в конструктор.

9.1 Общая структура конфигурации

Конфигурация ABGrid Engine состоит из нескольких разделов. Ниже приведена типовая «карта» конфигурации.

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

  // Политика прав (автоприменение data.policy из ответов
сервера)
  policy: { enabled: true },

  // Данные и транспорт
  data: { ... },

  // Итоговые строки
  summary: { ... },

// CRUD-движок
  crud: { ... },

  // Детальные сетки
  detailGrids: [ ... ],

  // Детальные панели
  detailsPanels: [ ... ],

    // UI-хелперы
  dialogs: true,
  loadingOverlay: true,

    // Локализация
   i18n : {….},
```

```
  // Визуальные настройки и UI
  view: { ... }

// Редактор (глобальные дефолты и поведение)
  editor: { ... },

  // Схема данных и UI (колонки, правила редактора по полям)
  schema: { fields: { ... } },

}
```

Важно:
• Параметр columns в новой архитектуре не используется — колонки и поведение редактора описываются через schema.fields.
• Раздел view отвечает за отображение (высоты, тулбар, pager, subGrid и др.).

Ниже — краткое назначение параметров:

autoLoad

Если true — грид автоматически вызывает загрузку данных после инициализации. Если false — данные загружаются только явным вызовом grid.load(). У детальных сеток блокируется мастер-сеткой.

detailGrids[]

Массив детальных компонетов ABGrid (master-detail), которые автоматически создаются и перезагружаются на основе текущей строки мастера. Каждая деталь описывается через { gridId, linkField, options }, представляет собой полноценный ABGrid Engine-компонент и конфигурируется точно так же, как и мастер-компонент.

view

Визуальная конфигурация грида: размеры, заголовок, тулбар, pager, сортировка, чекбоксы, subgrid, treeGrid и т.п. Раздел view влияет на UI, но не заменяет проверки прав (они выполняются в момент действия).

view.subGrid

Вложенная сетка внутри строки (subgrid). Настраивается через {enabled, linkField, options }. options — это обычная конфигурация грида, т.е. subgrid является полноценным экземпляром ABGrid Engine.

9.2 readOnly

Параметр readOnly переводит грид в режим «только просмотр».
При readOnly=true запрещены все операции изменения данных (create/update/delete), независимо от того, используется ли встроенная форма редактора или внешний контроллер.
ABGrid Engine использует единый контракт «конверта» (envelope) для загрузки данных и CRUD-операций. Контракт может быть обернут в dataKey.

9.3 autoLoad

Параметр autoload управлет автозагрузкой компонента сразу после инициализации. У детальных сеток данный параметр принудительно устанавливается мастер-сеткой принудительно в false при их создании. Детальные сетки пользователь может не создавать сам, это сделает компонент, если они заданы.

9.4 Итоговые строки

ABGrid Engine предоставляет возможность отображать в нижней части таблицы итоговые строки, которые могут включать информацию как по странице, так и глобальные итоги. Данные по странице рассчитываются компонентом автоматически, если этих данных нет в ответе сервера, глобальные данные, естесственно, могут быть отражены только если эти данные есть в ответе сервера.
Конфигурация итогового раздела задается следующим образом:

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

Параметры:

enabled — включает итоговые строки;

page — разрешает строку page-итогов;

total — разрешает строку total-итогов;

pageLabel и totalLabel — подписи строк;

separator — разделитель между несколькими агрегатами;

emptyText — текст для пустой итоговой ячейки.

layout - тип отображения. Варианты: 'stack' – одно под другим, 'inline'-в строку

Итоги задаются не только глобально, но и на уровне поля через schema.fields.<name>.summary. Поддерживаются варианты:
summary: true
summary: 'sum'
summary: ['sum', 'avg']
summary: { page: ['sum', 'count'], total: ['sum', 'avg', 'max'] }

Если summary: true, набор агрегатов выбирается автоматически: для числовых полей по умолчанию используется sum, для нечисловых — count.
Поддерживаемые агрегаты: sum, avg, min, max, count.

Page-итоги компонент вычисляет сам по строкам текущей страницы. Total-итоги сервер может вернуть, например, в data.summary.total или data.summaryTotal. Допустимы как полные объекты вида amount: { sum: 1500, avg: 300 }, так и сокращённая запись amount: 1500, если для поля ожидается только одна операция.

Публичный API: getSummary() — вернуть объект текущих page/total итогов; setSummary(summary) — вручную установить summary и перерисовать таблицу.

Важно: page-итоги без сервера работают сами, если для полей прописан summary; total-итоги без сервера обычно не появятся, если явно не вызвать setSummary(); итоговые строки строятся на основе schema.fields и привязаны к алиасам колонок.

9.5 Политика разрешений

ABGrid Engine позволяет управлять разрешениями создания, редактирования и удаления данных для каждого ABGrid-компонета. Опция policy.enabled управляет только автоматическим применением data.policy из ответов сервера. Ручной вызов setPolicyTree() из программного кода работает независимо от policy.enabled. При наличии в ответах сервера политики прав в параметре data.policy компонент каскадно применяет ее для всех своих дочерних компонентов и для своего субгрида. Если сервер перестал отправлять политику, то она сохраняется до перезагрузки страницы или до нового ответа сервера, содержащего эту политику (при data.policy = true). Более подробно режим описан в соответствующем разделе.

9.6 Глобальная политика при ответе сервера со статусом 401

ABGrid Engine поддерживает глобальную auth-политику для типового сценария «сервер вернул 401 — пользователю надо показать сообщение и перенаправить его на страницу авторизации».

Конфигурация:

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

Параметры:

- redirectOn401 — включает автоматическую реакцию на HTTP 401;
- loginUrl — URL страницы входа;
- redirectDelayMs — задержка перед переходом;
- redirectOnlyOnce — защита от многократных redirect при нескольких параллельных запросах;
- showToast — показывать пользователю уведомление перед переходом.

Поведение: если сервер вернул 401 и auth.redirectOn401 = true, компонент считает это ошибкой авторизации; при включённом showToast пользователю показывается сообщение из i18n.errMsg.unauthorized или стандартный текст; после задержки redirectDelayMs выполняется переход на loginUrl; при

redirectOnlyOnce = true повторные 401 не будут порождать каскад одинаковых уведомлений и повторных переходов.

Важно: auth-политика не заменяет серверную безопасность и не логинит пользователя автоматически; она лишь централизует клиентскую UX-реакцию на уже случившийся 401.

Дополнение по i18n: для auth-сценария используется ключ i18n.errMsg.unauthorized.

Политика может быть глобальной и установлена на все приложение:

```
// Глобальная политика
// ВАЖНО: вызвать ДО new ABGrid(...)
ABGrid.setDefaults({
    auth: {
        redirectOn401: true,
        loginUrl: '/auth/login',
        redirectDelayMs: 3000,
        redirectOnlyOnce: true,
        showToast: true
    },
    debug: {enabled: true}
});
```

Для кастомных запросов (например, fetch, скачивание отчетов и т.п.) предусмотрен публичный метод:

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

Пример: скачивание отчета:

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

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

Важно: работа с Response (Fetch API)

Объект `Response` позволяет прочитать тело ответа **только один раз**.

❗ Проблема

```
await response.json();
await response.text(); // ❌ Ошибка — тело уже прочитано
```

✅ Как решено в ABGrid

Метод `handleUnauthorized()`:

- использует `response.clone()` для чтения тела
- **не расходует основной response**

Это позволяет безопасно использовать ответ дальше:

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

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

⊘ Ограничения

- Метод обрабатывает **только статус 401**
- Другие ошибки (400, 500) должны обрабатываться отдельно
- Не следует повторно читать тело response после blob()/json()/text()

✅ Рекомендации

- Используйте `grid.handleUnauthorized()` во **всех кастомных fetch-запросах**
- Не дублируйте auth-логику вручную
- Не реализуйте собственные редиректы при 401

🧠 Архитектурный принцип

Вся логика авторизации должна быть **централизована в ABGrid**, а не размазана по коду приложения.

9.7 Раздел data

Раздел data управляет транспортом и способом, которым грид обменивается данными с приложением или сервером.

Ключевые параметры:

- url - endpoint сервера (если используется transport="server")
- method - HTTP-метод (по умолчанию POST)
- headers – статические заголовки для сервера
- filter – статический фильтр данных на сервере
- dataKey - имя поля-обертки для payload (опционально)
- interceptors - перехватчики request/response/error (например, auth/CSRF). Подробнее в соответствующем разделе.
- controller - режим обработки intents: 'internal' или 'external'
- transport - для internal-контроллера: 'server' или 'local'. В версии 1.0.0 поддерживается только значение 'server'.
- intents - обработчик intents (намерений) для external-контроллера
- inFormat/outFormat/csvDelimiter - форматы входящих/исходящих данных

Пример минимальной настройки транспорта:

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

9.7.1 Контроллер данных: internal и external

ABGrid Engine использует модель intents: View генерирует намерения (клики тулбара, пагинация, сортировка), а контроллер решает, что делать дальше.

Варианты контроллеров:

- internal (по умолчанию) - встроенный контроллер ABGrid. Он сам вызывает load()/setPage()/CRUD и может работать с сервером.
- external - ABGrid только эмитит intents. Решения (вызовы load/CRUD, свои модалки, свои запросы) принимает приложение.

Чтобы включить внешний контроллер:

```
data: {
```

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

Формат ctx в intents:

- action - строка действия (например, "toolbar:add" или "pager:page")
- payload - объект параметров действия (например, {page, rpp, rowId, rowIds, sortOrder})
- source - источник (например, "view" или "api")
- grid/api - ссылка на экземпляр грида

Типовые intents, которые генерирует View:

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

Пример внешнего контроллера (внешнее приложение решает что делать):

```
data: {
      controller: 'external',
      intents: async ({ action, payload, grid }) => {
                  if (action === 'toolbar:add') {
 // открыть вашу кастомную форму и затем сохранить через 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;
```

```
      // подтверждение может быть вашим (или используйте
встроенный диалог)
      const ok = await confirmApp('Удалить записи?');
      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;
    }
  }
}
```

В режиме external ABGrid по-прежнему можно использовать встроенный CRUD (grid.crud.*) и load(), но решение о вызове остается за приложением.

9.7.2 Формат обмена с сервером

ABGrid Engine позволяет задавать задавать различные форматы обмена с сервером и задаются соответствующими параметрами для входящего и исходящего трафика. Имеется в виду формат данных в data и в первую очередь данные самих строк. Поддерживаются форматы object, aoa, csv.

- В режиме object строка состоит из объектов имен полей и их значений, например:

```
{
  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"}
    ]
  }
}
```

- Формат aoa (Array of array). Рекомендуемый формат.

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

- Формат csv

```
rows: [

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

],
```

В режиме auto формат строк определяется автоматически по первой строке. Исходящий формат аналогичен. Параметр csvDelimiter определяет символ разделения значений полей в строке.

9.8 Раздел crud

Раздел crud описывает параметры встроенного CRUD-движка (операции create/read/update/delete через сервер).
Ключевые параметры:

- operParam — имя параметра операции (по умолчанию «oper»).
- operCreate / operRead / operUpdate / operDelete — значения параметра операции для create/read/update/delete.
- deleteBatchSize — размер пачки при массовом удалении (0 означает без ограничения).
- deleteProgress — показывать прогресс при удалении пачками.

9.9 Массив детальных сеток detailGrids

Массив детальных (подчиненных) компонентов задаются массивом detailGrids[]. Необходимости программной инициализации детальных компонентов нет, мастер-сетка создает их сама при наличии HTML-элемента из параметра gridId. Детальные сетки являются полноценными экземплярами ABGrid Engine.

Каждый элемент detailGrids описывается как объект:

- gridId — id DOM-контейнера, в котором будет жить детальная сетка (обязателен).
- link:{masterField: '…', detailField: '…'}, где masterField и detailField— имена полей связи. Значение берется из текущей строки мастера из поля masterField и добавляется в фильтр детальной сетки с именем detailField.
- options — конфигурация ABGrid Engine для детальной сетки (параметр autoLoad принудительно отключается, загрузку инициирует мастер).
- cascade — если true, после загрузки детали есть данные, курсор устанавливается на первую строку и автоматически загружаются её собственные детальные сетки.

Принцип работы: мастер управляет деталями. При смене текущей строки мастер устанавливает фильтр в детальной сетке и вызывает load({ resetPage:true }).
Фильтр детали формируется как merge: существующий filter (объект или функция) + { [detailField]: значение из текущей строки мастера }.

9.10 Массив детальных панелей detailsPanels

detailsPanels — массив пользовательских UI-панелей, привязанных к текущей строке. Движок вызывает их обновление при смене текущей строки. Массив состоит из объектов, каждый из которых задает саму панель и её поведение.

В отличие от detailGrids:
• detailsPanels не являются grid-компонентами
• содержимое полностью определяется пользовательским кодом
• ABGrid Engine управляет только жизненным циклом и привязкой к строке
Настройка detailsPanels выполняется в конфигурации грида:

```
detailsPanels: [
  {
    // --- Идентификация ---
    // (required) уникальный id панели
    id: "requestDetails",

    // --- Где находится DOM панели ---
    // (required) selector | Element | () => Element
    host: "#requestDetailsHost",

    // --- Откуда брать masterId  ---
    link: { masterField: "id" }, //  default: "id"

    // --- Правила авто-биндинга/очистки ---
    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: "Выберите запись…",  //default: "" (если пусто — hint только
показывается/скрывается)

    // --- Полностью ручной режим (override) ---
    // Если onShow задан → autoBind НЕ выполняется
// ctx.masterRow, ctx.masterId, ctx.host, ctx.grid, ctx.reason, ctx.force
    onShow(ctx) {
            // В ручном режиме вы сами заполняете DOM.
    },

    // Если onClear задан → autoClear НЕ выполняется
// ctx.host, ctx.grid, ctx.reason, ctx.prevMasterRow/Id/Field
    onClear(ctx) {
      // В ручном режиме вы сами очищаете DOM.
    },

    // --- Хуки “после” (для сложных частей) ---
    // Вызывается после autoBind или после onShow
```

```
    onAfterBind(ctx) {
      // Например, дорисовать сложный блок:
      // renderFiles(ctx.host.querySelector('#filesList'),
ctx.masterRow?.files);
    },

    // Вызывается после autoClear или после onClear
    onAfterClear(ctx) {
      // Например, дочистить сложный блок:
      // ctx.host.querySelector('#filesList').innerHTML = '';
    }
  }
]
```

Список обработчиков.

Обработчик	Когда вызывается	Назначение
`onShow(ctx)`	При появлении currentRow	Полностью ручной режим
`onClear(ctx)`	При currentRow = null	Полностью ручной режим
`onAfterBind(ctx)`	После autoBind или после onShow	Доработка панели
`onAfterClear(ctx)`	После autoClear или после onClear	Дочистка сложных блоков

Когда есть currentRow (текущая строка):

1. Если задан `onShow` →
   - автоматиечская прорисовка(autobinding) НЕ выполняется
   - вызывается `onShow`
   - затем `onAfterBind` (если есть)
2. Если `onShow` НЕ задан
   - выполняется авто-биндинг
   - затем `onAfterBind` (если есть)

Когда currentRow = null:

1. Если задан `onClear` →
   - автоочистка НЕ выполняется
   - вызывается `onClear`
   - затем `onAfterClear` (если есть)
2. Если `onClear` НЕ задан
   - выполняется авто-очистка
   - затем `onAfterClear` (если есть)

Контракт ctx

Для onShow / onAfterBind

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

Для onClear / onAfterClear

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

Жизненный цикл

onShow(ctx)

Срабатывает при выборе/смене текущей строки.

`ctx:`

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

onClear(ctx)

Срабатывает, когда `currentRow === null`.

`ctx:`

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

Публичный API

Доступен как `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' })` — повторно применить текущую строку ко всем панелям
- `grid.panels.showForRow(rowId, { force=false, reason='manual' })` — показать панели для конкретной строки
- `grid.panels.hideAll({ reason='manual' })` — очистить/скрыть панели (в зависимости от режима)
- `grid.panels.destroy()`

Важно: `refresh()` действует **на все панели**, потому что engine один и currentRow один.

Рекомендации

- Делайте каркас в HTML/шаблоне, а панели пусть только “подставляют значения”.
- Для длинных текстов фиксируйте высоту блока и включайте прокрутку (overflow:auto) чтобы layout не прыгал.
- Используйте afterClear для сложных внутренних частей панели, а простые поля оставьте на авто-очистку.

Параметры reason и force

Некоторые методы `grid.panels` и обработчики панели получают параметры:

- `reason`
- `force`

Эти параметры используются для управления поведением панелей и передачи контекста вызова.

---

reason

Что это

`reason` — строковый маркер причины вызова lifecycle панели.

Он не влияет на внутреннюю логику движка напрямую, но передаётся в `ctx` и позволяет панели реагировать по-разному в зависимости от источника вызова.

Где используется

Передаётся в:

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

Доступен как:

```
ctx.reason
```

Возможные значения (встроенные)

Значение	Когда возникает
`"row-change"`	Пользователь выбрал другую строку
`"refresh"`	Вызван `grid.panels.refresh()`
`"manual"`	Вызван `showForRow()` или `hideAll()`
`"disabled"`	Панели были отключены через `setEnabled(false)`
`"enabled"`	Панели были включены через `setEnabled(true)`
`"data-empty"`	После загрузки данных `currentRow` стал null
`"clear-current"`	Вызван `grid.clearCurrentRow()`

Движок может добавлять новые служебные значения в будущих версиях.

Пример использования

```
onAfterClear({ reason, host }) {
  if (reason === "data-empty") {
    host.querySelector('[data-panel-hint]').textContent =
      "Нет данных по фильтру.";
  }
}
```

---

force

Что это

`force` — логический флаг (`boolean`), указывающий, что панель должна обновиться даже если текущая строка не изменилась.

По умолчанию:

```
force = false
```

Где используется

Передаётся в:

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

Доступен как:

```
ctx.force
```

Когда применяется

`force` используется в методах:

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

Зачем нужен `force`

По умолчанию движок может не выполнять повторный bind, если `currentRowId` не изменился.

Если требуется принудительное обновление панели (например):

- изменился формат отображения
- изменился язык
- обновились справочные данные
- изменилась тема
- изменилось состояние вне masterRow

— используется `force: true`.

Пример

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

Поведение по умолчанию

Если:

- `reason` не передан — используется `"refresh"` или `"row-change"` в зависимости от контекста
- `force` не передан — считается `false`

Архитектурный принцип

- `reason` — объясняет *почему* произошёл вызов
- `force` — управляет *должен ли вызов происходить повторно*

Они не обязательны для использования, но дают гибкость для сложных панелей.

Разметка панели (обязательно для автоматической отрисовки и очистки)

Внутри `host` помечайте элементы, которые должны автоматически заполняться/очищаться, так:

- `data-panel="fieldName"` — значение поля из `masterRow[fieldName]`
- `data-panel-hint` — (optional) блок подсказки, показывается при отсутствии currentRow

Пример:

```
<div id="requestDetailsHost" class="uk-text-small abdp-panel">
  <div class="abdp-line"><b>Компания:</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>Имя:</b> <span data-panel="name">—</span></div>

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

  <div data-panel-hint class="uk-text-muted uk-margin-small-top">
    Выберите запись…
  </div>

  <!-- Пример "сложной" части, которую вы чистите/рисуете в onAfter* -->
  <div id="filesList"></div>
</div>
```

Если какой-то элемент не должен очищаться автоматически — добавьте ему атрибут `data-panel-keep`.

9.11 UI-хелперы

UI-хелперы задаются двумя параметрами и управляют разрешением использования внутренних системных диалогов и показом небольшого окна, показывающего выполнение операции компонентом, например, загрузки. Настройки индивидуальны для каждого объекта компонента ABGrid Engine.

```
dialogs: true | false,
loadingOverlay: true | false
```

9.12 Локализация (i18n)

ABGrid Engine использует централизованный набор текстов сообщений (i18n) для UI: ошибки, подтверждения, подписи кнопок и т.п.

- Рекомендации использования:
- вынесите все пользовательские тексты в i18n-словарь приложения и передайте их в конфигурацию грида
- не хардкодьте сообщения в обработчиках — используйте ключи/шаблоны
- все сообщения ABGrid должны быть переопределяемыми

Пример (концептуально):

```
i18n: {
  common: {
    confirmTitle: 'Подтверждение'
  },
  crud: {
    readOnly: 'Режим только просмотр',
    deleteConfirmOne: 'Удалить запись?',
    deleteConfirmMany: 'Удалить выбранные записи?'
  },
  validation: {
    required: 'Поле обязательно'
  }
}
```

Пользователь может полностью переопределить сообщения по своему желанию в своей конфигурации. Полный объект локализации приведен ниже:

```
i18n: {
  // CRUD: тексты ошибок контракта ответа сервера (envelope)
  crud: {
    noOpId: 'Ответ сервера не содержит обязательный код
операции opId',
    noRowId: 'Ответ сервера не содержит обязательный
идентификатор созданной записи id',
    incompleteServerData: 'Неполные данные от сервера',
    serverResponse: 'Ответ самого сервера: ',
    serverRejected: 'Сервер отменил операцию'
  },
  add: {
    caption: 'Добавить',
    hint: 'Новая строка'
```

```
  },
edit: {
  caption: 'Изменить',
  hint: 'Редактирование строки'
},
del: {
  caption: 'Удалить',
  hint: 'Удалить строку (строки)'
},
excel: {
  caption: 'Excel',
  hint: 'Экспорт в Excel'
},
word: {
  caption: 'Word',
  hint: 'Экспорт в Word'
},
pdf: {
  caption: 'Pdf',
  hint: 'Экспорт в Pdf'
},
pager: {
  page: "Стр. {0}",
  from: "из {1}",
  first: "Первая",
  prev: "Предыдущая",
  next: "Следующая",
  last: "Последняя",
  refresh: "Обновить",
  rpp: "Строк на стр.:",
  recordInfo: "{0} из {1} строк",
  emptyTable: "Нет строк для отображения"
},
msgAction: "Действие",
msgBranch: "Открыть ветку",
msgBranchAria: "Открыть связанные строки",
msgLoad: "Загрузка...",
msgSave: "Сохранение...",
msgDelete: "Удаление...",
msgDone: "Выполнено",
msgErrorTitle: "Ошибка",
msgNoData: "Нет данных",
```

```
  confirm: {
    title: "Подтверждение",
    deleteOne: "Удалить запись?",
    deleteMany: "Удалить выбранные записи ({0})?",
    deleteLinkedWarn: "Возможно удаление связанных данных",
    noAskDelete: "Больше не спрашивать"
  },

  errMsg: {
    readOnly: 'Режим только просмотр',
    rollbackOperAtServer: "Операция сервером отменена!",
    noRowsToDelete: "Нет строк для удаления!",
    noSelectedRowsToDelete: "Нет отмеченных строк для
удаления!",
    crudNotConfigured: "CRUD не настроен. Укажите data.url или
подключите model.repository."
  },

  editForm: {
    titleEdit: 'Редактирование записи',
    titleCreate: 'Новая запись',
    btnSave: 'Сохранить',
    btnSaveNext: 'Сохранить и продолжить',
// шаблон сообщения для confirm-полей (например, подтверждение пароля)
// {0} — label исходного поля
    confirmMismatch: 'Значение должно совпадать со значением в
поле «{0}»',
    noChanges: 'Нет изменений'
  },

  autocomplete: {
    notFound: 'Совпадений нет',
    loading: 'Загрузка...',
    error: 'Ошибка'
  },
  buttons: {
    btnClose: "Закрыть",
    btnOK: "OK",
    btnYes: "Да",
    btnDelete: "Удалить",
    btnNo: "Нет",
    btnCancel: "Отмена"
  }
}
```

9.13 Раздел editor

Раздел editor управляет поведением встроенной формы создания/редактирования записи (EditForm).
Важно: режим «только просмотр» задается параметром readOnly в корне options.
В editor нет собственного флага readOnly.
Ключевые параметры:

- requiredByDefault — обязательность полей по умолчанию в редакторе (может быть переопределена на уровне поля в schema).
- excludeFields — список полей, которые редактор никогда не показывает (служебные поля).
- confirmDelete / confirmDeleteMany — показывать подтверждение удаления одной/нескольких записей.
- autocomplete — дефолтные настройки встроенного автокомплита (если он включен на уровне поля).

9.14 Раздел schema

Раздел schema является центральным элементом конфигурации ABGrid Engine.

Через schema описываются:
• поля данных
• колонки грида
• правила отображения
• поведение редактора

9.14.1 Порядок, видимость и ширина колонок

Параметр конфигурации schema.fields содержит описание всех полей строки. Пример:

```
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 }
       }
    }
  }
}
```

Видимость колонок задается параметром schema.fields<fname>.ui.grid.visible. Порядок может быть задан явно через schema.order. Если порядок не задан, используется порядок объявления полей. Заголовок поля задаётся параметром label. Ширина колонок задается только для видимых колонок и рекомендуется задавать ширину каждого столбца в %, см. пример выше. Ширину колонок можно задавать в `px` числом `width:100`. При этом `'px'` писать не надо и даже вредно. Если ширина таблицы выйдет за границы экрана, то появится ползунок горизонтальной прокрутки. Такой ползунок один общий для таблицы и для суммарных строк.
Пример:

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

9.14.2 Переопределение свойств в editor

Для каждого поля можно переопределить дефолтные свойства глобального editor и задать индивидуальные свойства в режимах редактирования и создания записи. Переопределить можно обязательность, видимость, режим readOnly.

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

9.15 Режим отладки конфигурации

ABGrid Engine поддерживает облегчённый dev/debug-режим для раннего обнаружения ошибок в конфигурации.

Конфигурация:

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

Допустим также алиас debug, который обрабатывается так же, как dev.

Параметры: enabled — включает проверки; checkCyrillicKeys — предупреждать, если в ключах конфигурации встречается кириллица; checkUnknownOptions — предупреждать об опциях, которые компонент не понимает.

Назначение режима: быстро ловить опечатки в именах параметров; обнаруживать случайную русскую раскладку в ключах; находить «лишние» опции, которые разработчик ожидает, но движок игнорирует.

Важно: это именно режим диагностики, а не runtime-защита; он не блокирует работу компонента, а только выводит предупреждения в console.warn; разделы schema и i18n намеренно допускают более свободную структуру и поэтому не должны проверяться так же жёстко, как остальные секции конфигурации.

Рекомендуется включать dev.enabled на этапе разработки и выключать в production.

9.16 Раздел view

Раздел view описывает визуальную конфигурацию и поведение UI грида (высоты, тулбар, pager, сортировку, чекбоксы, subGrid, treeГрид и т.п.).
Основные параметры:

- subGrid — вложенная сетка внутри строки: { enabled, linkField, options }.
- treeGrid — режим иерархических строк: { enabled, column, model, pid, isLeaf, level, levels }.
- tableTitle — заголовок таблицы (enabled, caption, hint, toggle).
- toolBar — настройка панели инструментов (кнопки, порядок, кастомные действия).
- pager — постраничная навигация (enabled, rpp).
- sortable / multiSort / sortOrder — сортировка и мультисортировка.
- checkbox — показывать чекбоксы выбора строк.
- height / minHeight / maxHeight — размеры области грида.

Раздел view влияет на UI, но не является механизмом безопасности. Серверная валидация прав остается обязательной.

9.16.1 Выбор строк, размеры, заголовки

```
checkbox: true     // управление отображением чекбокса выбора строк
height: 400        // высота всего контейнера компонента в px
showHeader: true // управление рендером заголовка самой таблицы

// Ограничение размеров в режиме получения дополнительного пространства, px
minHeight: 200
maxHeight: 800

 // Заголовок компонента

tableTitle: {
  enabled: true      // управление разрешением
  caption: "Заголовок таблицы"    // текст заголовка компонента
// всплывающая подсказка при наведении курсора мыши (title)
  hint: "Подсказка",
  toggle: true       // разрешение сворачивания компонента до заголовка
 }
```

9.16.2 Сортировка и мультисортировка

ABGrid Engine поддерживает как сортировку по одному полю, так и мультисортировку (по нескольким полям) на уровне UI и формирования sortOrder.

Настройки находятся в options.view:

- view.sortable: true|false — включить сортировку
- view.multiSort: true|false — разрешить мультисортировку (при удержании клавиши Shift или Ctrl
- view.sortOrder: [] — начальный порядок сортировки (массив {alias, dir})

Пример:

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

Пользователь может задавать сортировку и мультисортировку по полям в соответствии со своим предпочтением. Мультисортировка предусматривет 3 состояния каждого поля (за исключением сортировки по 1 полю) – asc-desc-none. Состояния меняются по кругу при удержании клавиш Shift или Ctrl. Из-за особенностей использования в браузерах клавиши Alt она не задействована под мультисортировку и никак не обрабатывается. Порядок полей при мультисортировке подсвечивается в каждом заголовке поля. При удержании клавиши Shift или Ctrl пользователь может задать сортировку по нескольким полям по своему желанию как по направлению (asc, desc), так и сам порядок по полям. При отпускании клавиши происходит отправка запроса на сервер, в котором порядок сортировки содержится в массиве-параметре sortOrder, сервер обязан вернуть данные в соответствии с данным порядком. Благодаря тому, что используется массив для задания порядка полей пользователем и этот массив постоянно перестраивается под его воздействием, то непосредственно сам порядок полей выбора сохраняется таким, каким его задал пользователь. Сортировка по 1 полю так же доступна и срабатывает при клике по заголовку поля без удержания клавиш. В этом случае у поля нет третьего состояния «none» и оно меняется по кругу asc-desc, запрос на сервер отправляется немедленно по клику по заголовку столбца. Начальное значение сортировки задается конфигурацией и отправляется на сервер при первой же загрузке данных при условии, что автозагрузка разрешена.

9.16.3 Панель инструментов

Панель инструментов позволяет разместить пользовательские кнопки для выполнения каких-либо действий. Конфигурирование производится в объекте toolBar. Правило задания кнопок очень простое - необходимо задать само действие-объект, в котором прописать надпись и подсказку, например, так:

```
toolBar: {
  add: {caption: 'Добавить',  hint: 'Новый пользователь'},
   // актуально только для treeGrid
  addChild: {
     caption: 'Добавить дочернюю',
     hint: 'Новая дочерняя строка'
   },
  edit:{caption:'Изменить',hint:'Редактирование   пользователя'
},
  del:{caption:'Удалить',hint:'Удалить
пользователя(пользователей)' }
}
```

Кнопки необходимо прописывать, существующие в конфигурации по умолчанию автоматически не добавляются на панель инструментов и скорее прописаны там для того, чтобы не забыть как их задавать.

Обработчик нажатия на кнопки один на все кнопки, пользователь может назначить свой обработчик, в котором произвести те или иные действия. Идентификация операции осуществляется передачей параметра, значение которого совпадает с именем объекта конфигурации кнопки, т.е. в соответствии с примером выше это будут «add», «edit», «del».

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

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

Параметры, передаваемые в обработчик:

- action – наименование кнопки(операции)
- ctx – контекст

- grid – инстанс компонента

Пользователький обработчик срабатывает до встроенного, для прерывания работы которого необходимо вернуть в обязательном порядке false, в противном случае (при отсутствии возврата либо возврате true) он продолжит работу. Продукт поставляется со встроенными иконками для отображения действий:

- add – добавление записей
- addChild – добавление дочерней записи в режиме treeGrid
- edit – редактирование записи
- del – удаление записей
- excel – экспорт в Excel
- word – экспорт в Word
- pdf – экспорт в PDF

Встроенный редактор поддерживает создание, редактирование, удаление записей. Обработка экспорта в указанные выше форматы движком не производится.

Обработчик кнопки можно назначить непосредственно в конфигурации, например, так:

```
toolBar: {
    deleteUsedOrExpired: {
    requiresSelections:false, // см. описание ниже
    requiresData:false,       // см. описание ниже
    caption: 'Очистить токены',
    hint:      'Удалить      использованные/истекшие      токены
восстановления',
    async onClick(grid, payload) {
          payload.preventDefault();
          try {
        // ВАЖНО: url свой, не options.data.url
               const res = await grid.http.request({
               url: '/admin/api/password-reset-tokens/cleanup',
          // endpoint
               method: 'POST', // или DELETE
               headers: grid.options?.data?.headers || {},
               dataKey: null,     // если  API без обёртки key
               data: {},          // если нужно тело запроса
               strict: true,     //  ожидаем  envelope  {success,
               message, data}
               unwrapData: true,
               intent: 'toolbar:deleteUsedOrExpired'
           });
```

```
      }
              if (res?.success === true) {
                await grid.msgDone(res.message || 'Готово');
                await       grid.refresh();       //       или
grid.load({resetPage:false})
              } else {
                await   grid.msgError(res?.message   ||   'Ошибка
очистки токенов');
              }
            } catch (e) {
              console.error(e);
              await grid.msgError('Ошибка запроса к серверу');
            }
          }
   }
}
```

Так же моно назначить свой обработчик стрелочной функцией:

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

Если в пользовательском коде происходит обработка стандартных кнопок «add», «addChild», «edit», «del» то необходимо вернуть false для прерывания системного потока обработки нажатия кнопок. Рекомендуется всегда возвращать false после обработки, независимо от типа кнопок.

Важно!

При пустой таблице будет доступна только кнопка «Добавить» при условии, что компонент не является дочерним. Во всех остальных случаях при отсутствии данных кнопки отображаются, но их действия запрещены.

На панель инструментов можно так же добавлять элементы «select» со своими кнопками выполнения действий. Применяется, например, для массовых операций с данными.

Конфигурация задается таким образом (выделено жирным для select:

toolBar: {

bulkGroup: {
type: 'group', // обязательный параметр

```
   class: 'abgrid-bulk',      // CSS-класс

   bulk: {                        // идентификатор группы (id)
    type: 'select',               // тип, обязателен
    caption: 'Операция',
    placeholder: '— выберите —',
    options: [                    // сам select
      { value:'delete', caption:'Удалить' },
      { value:'spam', caption:'Пометить как спам' }
    ]
   },

   applyBulk: {                   // кнопка для select, id кнопки
    caption: 'Применить',
    requiresSelection: true
   }
  },

  edit: { caption:'Изменить' }       // обычная кнопка
}
```

Верхняя обертка (в примере с именем bulkGroup) служит для отрисовки на странице элементов в одном контейнере чтобы при изменении размеров экрана кнопка действия не «убегала» от своего select и не ломалась общая верстка. API группы:

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

В качестве id для select служит ключ, в примере это «bulk», для его кнопки как обычно, ключ самой кнопки, в примере это «applyBulk».

9.16.4 Параметры доступности кнопок панели инструментов

requiresSelection: boolean

Определяет, требуется ли выбранная (текущая) строка для активации кнопки.

- `true` — кнопка активна только при наличии выбранной строки.
- `false` (по умолчанию для пользовательских кнопок) — выбор строки не обязателен.

Типичное применение:

- `edit`, `delete`, `addChild`
- кастомные действия над конкретной записью (клонирование, открыть карточку и т.п.)

requiresData: boolean

Определяет, требуется ли наличие данных в таблице для активации кнопки.

- `true` — кнопка активна только если таблица содержит хотя бы одну строку.
- `false` (по умолчанию для пользовательских кнопок) — наличие данных не обязательно.

Типичное применение:

- экспорт данных
- массовые операции
- действия, имеющие смысл только при наличии записей

Приоритет правил

1. Если `requiresSelection: true` — кнопка будет отключена при отсутствии выбранной строки (даже если данные есть).
2. Если `requiresData: true` — кнопка будет отключена при пустой таблице.
3. Если оба параметра `false` или не заданы — кнопка доступна независимо от состояния грида.

Дефолтные значения параметров:

Кнопка	**requiresSelection**	**requiresData**
add	false	false
edit	true	false
delete	true	false
addChild	true	false
custom (по умолчанию)	false	false

9.16.5 Встроенный пейджер

Копонент имеет свой встроенный пейджер, конфигурирование которого производится объектом pager, имеющий всего 2 параметра-разрешние отображения enabled и массивом rpp[], в котором указывается параметр отображения строк на странице при одном запросе.

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

В этом примере показаны параметры встроенного пейджера по умолчанию, которые можно переопределить своей конфигурацией.

9.16.6 Управление пространством

Движок позволяет управлять рабочим пространством других ABGrid Engine-компонентов на основе конфигурации. Механизм действия заключается в том, что при сворачивании сетки до заголовка (самого компонента, не столбцов таблицы) она рассчитывает своё освобождающееся пространство и распределяет его пропорционально заявкам между другими. При восстановлении исходного размера все потребители восстанавливаются до размеров, заданных в их конфигурации.

Конфигурация задается параметром-массивом view.toggleGrid[].

```
toggleGrids: [
   {
     gridId: 'appointedCompanies', // компонент-заявитель (потребитель)
     enabled: true,
     share: 100,                   // заявка в % от свободного места
       reload: true
   },
   ……
]
```

Каждый объект массива конфигурирует отдельный ABGrid-компонент. Описание параметров конфигурации одного объекта:

- gridId : string, аттрибут «id» корневого div-элемента
- enable : boolean, разрешение\запрещение изменения состояния
- share: number, в %. Заявка каждого элемента массива на объем освобождающегося пространства отдающего компонента
- reload: boolean. Параметр управления перезагрузкой при увеличении пространства для компонента-элемента массива. При true у компонента устанавливается максимальное возможное значение rpp (record per page) и производится автоматическая перезагрузка.

AGBrid Engine автоматически нормализует (производит перерасчет) сумму всех заявок к 100% и затем пропорционально распределяет свободное пространство между ними согласно заявок и устанавливает его. В версии 1.0 компонента механизм работает на одном уровне по принципу «один управляет многими». Иерархического распределения нет. Срабатывает по пиктограмме сворачивания\разворачивания «главного» компонента.

9.16.7 Subgrid

Subgrid — это детальная сетка, отображаемая внутри строки master-grid.
Особенности:
• subgrid создаётся master-grid динамически при раскрытии строки и уничтожается при закрытии
• subgrid всегда связан с конкретной строкой master
• subgrid использует отдельный контейнер внутри строки
• subgrid может получать политику прав от master-grid

По своей сути subgrid представляет собой полноценный компонент ABGrid Engine, управляется master-grid и существует автономно до момента закрытия, при закрытии строки с ним он уничтожается.

Конфигурация задается в объекте subGrid и со стороны разработчика не требуется написания никакого дополнительного кода для работы данного механизма. Все что надо сделать – разрешить сам режим, переопределить при необходимости поле связи в мастер-сетке и определить конфигурацию самого субгрида как ABGrid Engine – компонента.

```
subGrid: {
  enabled: false,  // Разрешение режима
  link:{ masterField: 'id', // поле связи в мастер-сетке
         detailField: 'id'
       }
   options: null   // параметр-конфигурация полноценного ABGrid Engine
}
```

9.16.8 TreeGrid

TreeGrid — режим отображения иерархических данных в одной сетке. Используется для категорий, структур, вложенных сущностей. Версия компонента 1.0.0 поддерживает модель древовидных структур данных Adjacency, модель NestedSet может быть реализована по запросам пользователей. TreeGrid не является разновидностью detail-grid. TreeGrid — это режим отображения иерархических данных в рамках одной сетки. Режим включается через конфигурацию view и не создаёт дополнительных grid-инстансов.

Настройка TreeGrid производится в разделе view, в примере отражены в том числе значения параметров по умолчанию:

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

Основные параметры TreeГрид (v1):

- enabled — разрешает режим TreeGrid.
- model — тип модели древовидной структуры.
- column — имя поля (колонки), в которой рендерится дерево.
- pid — поле идентификатора родительского узла (adjacency list).
- isLeaf — true\false, признак того, что строка есть листок (true) или ветка (false)
- level — поле уровня вложенности, если сервер отправляет (опционально). При отсутствии в ответе сервера компонент вычисляет level автоматически.
- levels — количество уровней вложенности для ответа от сервера за запрос. По умолчанию 1.

Режим TreeGrid является исключительно режимом визуализации. Подготовка иерархии данных находится на стороне сервера или external controller.

9.17 Конфигурация по умолчанию (DEFAULT_OPTIONS)

Ниже приведена полная дефолтная конфигурация ABGrid Engine

```
export const DEFAULT_OPTIONS = {

    // Управление автозагрузкой данных сразу после инициализции,
     // для детальных сеток мастер-сетка запрещает принудительно при их создании
    autoLoad: true,

  // Глобальная политика: режим "только просмотр".
  // Влияет на ВСЕ операции записи (create/update/delete) вне зависимости от того,
  // используется ли встроенная форма редактора или внешняя.
    readOnly: false,

    // Итоги/агрегаты внизу грида (по странице и/или общие).
  // page-итоги считаются на клиенте по текущим rows.
  // total итоги обычно приходят с сервера в data.summary / data.summaryTotal.
    summary: {
        enabled: false,
        page: true,
        total: true,
        pageLabel: 'Σ Page',
        totalLabel: 'Σ Total',
        separator: ' · ' ,
        emptyText: ''
       // кол-во цифр после запятой (глобальная  настройка)
        fractionDigits: 2
    },

  // Политика прав (опционально).
  // Если enabled=true, ABGrid при любом ответе сервера пытается применить
data.policy через setPolicyTree().
 // Сервер является источником истины: если data.policy отсутствует, то ничего не
меняется.
    policy: {
        enabled: false
    },

  // Глобальная политика авторизации/редиректа (опционально).
  // Если redirectOn401=true, то при HTTP 401 ABGrid покажет ошибку (если UI умеет)
  // и через redirectDelayMs выполнит переход на loginUrl (однократно).
    auth: {
        redirectOn401: false,
```

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

// dev/debug режим (минимальные проверки конфига).
// Включайте в разработке, чтобы быстрее ловить опечатки в ключах и ошибочных опциях.
// Проверки:
// - checkCyrillicKeys: предупреждать, если в КЛЮЧАХ конфига встречается кириллица (часто это опечатка)
// - checkUnknownOptions: предупреждать о не поддерживаемых опциях (по известным секциям)
// Примечание: значения строк (label/caption/hint и т.п.) не проверяются — там русские тексты нормальны.

```
    dev: {
        enabled: false,
        checkCyrillicKeys: true,
        checkUnknownOptions: true
    },
// транспорт
    data: {
        url: null,              // url запросов
        method: 'POST',         // HTTP-метод по умолчанию
        headers: {},            // статические заголовки (при каждом запросе)
        filter: null,           // статический фильтр (при каждом запросе)
           // ключ-идентификатор при централизованных запросах на один url
           // если задан, то запрос на сервер оборачивается в объект с этим именем
        dataKey: null,

        // Интерцепторы (request/response/error)
// ABGrid Engine ничего не знает об auth/CSRF/headers и т.д., но имеет механизм,
// предоставляющий пользователю производить модификацию запросов и ответов сервера
// Используйте interceptors для модификации  headers/params/body при каждом запросе.
   interceptors: {
       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)>
   },
```

```
    // Контроллер данных
    // internal: ABGrid сам обрабатывает intents и может работать с сервером
    // external: ABGrid только эмитит intents, решения принимает приложение
        controller: 'internal',
    // transport для internal контроллера. зарезервирован под будущий функционал
        transport: 'server',
    // external intents handler: function(ctx) or object map by action
        intents: null,

    // форматы строк
        inFormat: 'auto', // 'auto'|'aoa'|'object'|'csv'
    // формат исходящих данных (CRUD, пользовательские запросы): 'aoa'|'object'|'csv'
        outFormat: 'object',
        csvDelimiter: ';'
    },
    // CRUD (встроенные операции)
    crud: {
        operParam: 'oper',          // имена параметров запроса
        operCreate: 'create',
        operRead: 'read',
        operUpdate: 'update',
        operDelete: 'delete',
// ограничение количества строк для удалений в режиме batch (0 - нет ограничений)
        deleteBatchSize: 0,
// управление отображением прогресса при batch-удалениях
        deleteProgress: true
    },

    // массив детальных сеток (внешние контейнеры, режим linked)
  // {gridId:'', link:{masterField:'id', detailField:'...'},  options:null}
    detailGrids: [],

    // внешние UI-панели, привязанные к текущей строке (НЕ гриды)
    detailsPanels: [],

    // UI-хелперы
    dialogs: true,
    loadingOverlay: true,

    // i18n (локализация)
    i18n: {
    // CRUD: тексты ошибок контракта ответа сервера (envelope)
        crud: {
```

```
            noOpId: 'Ответ сервера не содержит обязательный
код операции opId',
            noRowId: 'Ответ сервера не содержит обязательный
идентификатор созданной записи id',
            incompleteServerData: 'Неполные данные от
сервера',
            serverResponse: 'Ответ самого сервера: ',
            serverRejected: 'Сервер отменил операцию'
        },
        add: {
            caption: 'Добавить',
            hint: 'Новая строка'
        },
        edit: {
            caption: 'Изменить',
            hint: 'Редактирование строки'
        },
        del: {
            caption: 'Удалить',
            hint: 'Удалить строку (строки)'
        },
        excel: {
            caption: 'Excel',
            hint: 'Экспорт в Excel'
        },
        word: {
            caption: 'Word',
            hint: 'Экспорт в Word'
        },
        pdf: {
            caption: 'Pdf',
            hint: 'Экспорт в Pdf'
        },

        // Toast (уведомления)
        toast: {
            close: 'Закрыть',
            closeAll: 'Закрыть все'
        },

        pager: {
            page: "Стр. {0}",
            from: "из {1}",
            first: "Первая",
```

```
            prev: "Предыдущая",
            next: "Следующая",
            last: "Последняя",
            refresh: "Обновить",
            rpp: "Строк на стр.:",
            recordInfo: "{0} из {1} строк",
            emptyTable: "Нет строк для отображения"
        },
        msgAction: "Действие",
        msgBranch: "Открыть ветку",
        msgBranchAria: "Открыть связанные строки",
        msgLoad: "Загрузка...",
        msgSave: "Сохранение...",
        msgDelete: "Удаление...",
        msgDone: "Выполнено",
        msgErrorTitle: "Ошибка",
        msgNoData: "Нет данных",

        confirm: {
            title: "Подтверждение",
            deleteOne: "Удалить запись?",
            deleteMany: "Удалить выбранные записи ({0})?",
            deleteLinkedWarn: "Возможно удаление связанных
данных",
            noAskDelete: "Больше не спрашивать"
        },

        errMsg: {
            readOnly: 'Режим только просмотр',
            rollbackOperAtServer: "Операция сервером
отменена!",
            noRowsToDelete: "Нет строк для удаления!",
            noSelectedRowsToDelete: "Нет отмеченных строк для
удаления!",
            crudNotConfigured: "CRUD не настроен. Укажите
data.url или подключите model.repository.",
               unauthorized: 'Требуется авторизация'
        },

        editForm: {
            titleEdit: 'Редактирование записи',
            titleCreate: 'Новая запись',
            btnSave: 'Сохранить',
            btnSaveNext: 'Сохранить и продолжить',
```

```
            // шаблон сообщения для confirm-полей (например,
подтверждение пароля)
            // {0} — label исходного поля
            confirmMismatch: 'Значение должно совпадать со
значением в поле «{0}»',
            noChanges: 'Нет изменений'
        },

        autocomplete: {
            notFound: 'Совпадений нет',
            loading: 'Загрузка...',
            error: 'Ошибка'
        },
        buttons: {
            btnClose: "Закрыть",
            btnOK: "OK",
            btnYes: "Да",
            btnDelete: "Удалить",
            btnNo: "Нет",
            btnCancel: "Отмена"
        }
    },
    // view (UI грида как компонента/контейнера)
    view: {

  // Управление отображением заголовка таблицы (THEAD)
  // false: заголовок не рендерится и не участвует в обработке кликов
сортировки/"выбрать все".
        showHeader: true,
          // управление отображением чекбокса выбора строк
        checkbox: true,
        height: null,     // Высота всего контейнера компонента
// Ограничение размеров
        minHeight: 200,
        maxHeight: 800,

        tableTitle: {           //  Заголовок компонента
            enabled: true,     //  управление разрешением отображения
            caption: "",        //  текст заголовка компонента
//  всплывающая подсказка при наведении курсора мыши (title)
            hint: "",
            toggle: true        // управление разрешением
сворачивания\разворачивания компонента
        },
```

```
sortOrder: [],     // массив начальной сортировки
sortable: true,    // управление разрешением  сотрировки
multiSort: true,   // управление разрешением  мультисортировки
sortIcons: {       // иконки отображения направления сортировки
    asc: '▲',
    desc: '▼',
    none: ''
},

toolBar: {}, // панель инструментов для отображения кнопок действий

    // встроенный пейджер
pager: {
    enabled: true,       // управление разрешением отображения
    // массив значений количества строк на странице
    rpp: [10, 20, 30, 50]
},

// массив компонентов для управления их пространством при изменении своего
toggleGrids: [],

// subGrid (встроенный дочерний грид под строкой)
subGrid: {
// Разрешение режима связь master-detail: subgrid.<detailField> = 
master.<masterField>
    enabled: false,
    link: { masterField: 'id', detailField: 'id' },
// параметр-конфигурация полноценного ABGrid Engine
    options: null
},

// tree (UI: иерархические строки)
treeGrid: {
    enabled: false,
// В какой колонке отображать дерево (основная/единственная колонка дерева).
// Указывается как alias/имя поля из schema.
    column: '',
// Модель хранения дерева на сервере (для выбора TreeEngine).
// Сейчас реализована adjacency-list (pid). nestedset зарезервировано под будущую 
// реализацию.
    model: 'adjacency',   // модель древовидной структуры данных
    pid: 'pid',           // имя поля родителя
```

```
   // имя поля признака того, что строка есть "лист" (не ветка, не имеет родителя)
            isLeaf: 'isLeaf',
  // уровень вложенности. если серввер не отдает, то вычисляется автоматически
            level: 'level',
  // запрос количества уровней вложенности данных от сервера за запрос
            levels: 1
        },
    },

    // editor (встроенный редактор записи)
    editor: {
        requiredByDefault: false,
        // исключить служебные/editor-only поля из UI редактора (страховка)
        excludeFields: ['actions', 'extData'],
        // подтверждения для удаления
        confirmDelete: true,
        confirmDeleteMany: true,

        autocomplete: {
            url: null,
// dataKey по умолчанию, если options.data.dataKey пуст.
// Ключ идентификации запросов при централизованной обработке на стороне сервера
            dataKey: null,
            queryParam: 'q',      // параметр, содержащий строку поиска
            fieldParam: 'field',  // идентификатор поля запроса поиска
            minChars: 3,          // минимальное количество символов для
начала поиска
            debounceMs: 250,      // защитная пауза в мс
            limit: 30,        // ограничение количества ответов от серера
            strict: true      // управление ответом сервера(см. ниже)
        }
    },

   // schema (правила). ВАЖНО: первичный ключ ВСЕГДА зарезервирован как поле 'id'
    schema: null,

    // дефолты внутреннего состояния
    page: 1,
    totalRecords: 0
};
```

О параметре strict в autocomplete.

Параметр strict в autocomplete определяет, должен ли сервер строго соблюдать формат ответа ABGrid (success, data, dataKey) или допускается “мягкий” режим, в котором компонент пытается извлечь список вариантов из любого разумного формата ответа.
В строгом режиме любое нарушение контракта считается ошибкой запроса, в мягком — приводит лишь к пустому списку вариантов без генерации ошибки.

10. Политика разрешений (policy) подробно

ABGrid Engine поддерживает модель разрешений в виде дерева policy (policyTree), получаемого с сервера или устанавливаемого программно. Компонент, получивший политику прав распространяет ее на свои детальные компоненты и на свой компонент subgrid при его наличии. Распространение политики прав происходит каскадно вниз, т.е. детальные сетки применяют политику прав и прокидывают дальше для своих детальных и субгрид компонентов.

Формат (пример):

```
data: {
  ...,
  policy: {
     // для материнского компонента
     allow: { create: true, update: false, delete: false },
     details: {
        // политика прав для детальной сетки price
          price: { read: true, write: false },
          …
     },
      // для своей подсетки
      subgrid:{allow:{create:false,update:true,delete:false} }
      }
}
```

Важно: опция policy.enabled управляет только автоматическим применением data.policy из ответов сервера. Ручной вызов setPolicyTree() работает независимо от policy.enabled.

11. Editor и schema подробно

Editor отвечает за создание и редактирование записей. Поведение редактора полностью описывается через schema.

11.1 Общая идея schema

В ABGrid Engine поведение редактора описывается через единую конфигурацию schema.
schema — это «контракт» между UI-редактором и данными: какие поля существуют, как они отображаются, какие значения допускаются и какие правила действуют при создании и редактировании.
Ключевой принцип :
• дефолты задаются на верхнем уровне schema (общие для всех режимов\полей)
• переопределения задаются по режимам create/edit (только то, что отличается)

11.2 Глобальные настройки Editor

Editor — это подсистема UI для создания/редактирования записи. Общие параметры редактора задаются в options.editor.
Основные параметры:

- requiredByDefault (boolean) — глобальная политика обязательности полей. Если true — поля считаются required по умолчанию, если false — нет. На уровне schema это можно переопределять для конкретного поля и для режимов create/edit.
- visibleByDefault (boolean) — глобальная политика видимости полей в редакторе по умолчанию (если поле не переопределяет visible).
- readOnlyByDefault (boolean) — глобальная политика readonly для полей редактора по умолчанию (если поле не переопределяет readOnly).
- confirmDelete / confirmDeleteMany — подтверждение удаления одной записи или пачки.
- autocomplete — глобальные параметры автокомплита (enabled, debounceMs, limit и т.п.), которые могут быть локально переопределены.

Внутренний редактор открывает автоматически создаваемую форму редактирования по кнопке «Добавить» или «Редктировать», форма на редактирование записи открывается так же по двойному клику по строке.

11.3 Структура schema

В общем виде schema имеет следующую форму:

```
schema: {
  fields: {
    <fieldName>: {
       // дефолты поля (общие для create/edit)
       ...
       // переопределения только для режима create
       create: { ... },
       // переопределения только для режима edit
       edit:   { ... }
     }
  }
}
```

Примечания:
• fieldName — ключ поля (обычно совпадает с именем свойства в объекте строки данных).
• Внутри поля допускаются только “простые” дефолты; всё, что зависит от режима — в create/edit.
• create/edit переопределяют только заданные свойства, остальное берётся из дефолтов поля и глобальных editor.*ByDefault.

11.4 Поля и их свойства

Ниже приведён рекомендуемый минимальный набор свойств поля . Конкретный набор зависит от типа редактора и UI.

- label — человекочитаемое название поля (может быть i18n-ключом).
- type — Тип данных/виджет редактора (text, number, date, select и т.п.).
- required — Обязательность. Если не задано — применяется editor.requiredByDefault.
- visible — Видимость поля в форме. Если не задано — применяется editor.visibleByDefault.
- readOnly — Только чтение (true/false). Если не задано — применяется editor.readOnlyByDefault.
- validators — Набор валидаторов (см. раздел 11.6).
- autocomplete — Локальные настройки автокомплита для поля (если поле поддерживает).
- ui — UI-настройки поля (ширина, классы, layout-параметры).

11.5 Переопределения в режимах create/edit

Для большинства проектов правила в create и edit различаются. В ABGrid Engine это решается явно через вложенные секции create/edit внутри описания поля.

Пример: поле обязательно при создании, но не редактируется после:
```
schema: {
      fields: {
             code: {
                   label: "Код",
                   create: { required: true},
                   edit: { readOnly: true }
             }
        }
}
```

11.6 Валидация в редакторе

Валидация описывается на уровне поля через validators (и/или required). Рекомендуется держать правила рядом с описанием поля, а не размазывать по событиям UI.
Пример:

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

Важно: ABGrid Engine валидирует форму на клиенте для UX, но серверные проверки обязательны в любом случае. Поддерживаемые типы:
- string
- int
- number
- float

- email
- phone
- date
- enum
- boolean

Важно для понимания:
Параметры для валидатора:
type:”email”,
required: true,
minLen: 6,
maxLen: 120,
regex: null

11.6.1 Числовые ограничения

min : number

Минимально допустимое значение.

max : number

Максимально допустимое значение.

messageMin : string

Сообщение при нарушении min.

messageMax : string

Сообщение при нарушении max.

ABGrid Engine использует **строго фиксированный набор параметров message***. Произвольные имена параметров сообщений не поддерживаются. Правило написания параметра message : необходимо дописать сам параметр в стиле camelCase , например,
messageRequired: 'Email обязателен',
messageEmail: 'Введите корректный email',
messageMinLen: 'Email слишком короткий',
messageMaxLen: 'Email слишком длинный

- Проверки выполняются **только для видимых и редактируемых полей**
- Если поле **не required** и значение пустое — остальные проверки не выполняются
- Все сообщения message*:
    - имеют **фиксированные имена**
    - применяются **по типу проверки**, а не по имени поля
- Если message* не задан — используется стандартное сообщение движка

11.6.2 Проверка по регулярному выражению

regex : RegExp | string | null

Регулярное выражение для проверки значения.

- Может быть:
    - объектом RegExp
    - строкой (будет преобразована в RegExp)
    - null — проверка отключена

messagePattern : string

Полный список допустимых сообщений:

- messageType – сообщение, если введенное значение не соответствует ожидаемому типу
- messageMinLen, messageMaxLen – сообщения о нарушении длины строки
- messagePattern - сообщение при несоответствии regex
- messageMin, messageMax – сообщение при нарушении ограничений числа
- messageEmail – сообщение при некорректном email
- messagePhone - сообщение при некорректном номере телефона
- messageDate – сообщение при некорректной дате
- messageEnum, messageEnumEmpty – сообщение если значение не входит в список enum или вообще пустое значение

В большинстве случаев для валидации поля достаточно декларативных параметров (required, min/max, minLen/maxLen, regex, email, enum и т.п.). Для нестандартных сценариев ABGrid Engine поддерживает пользовательские валидаторы поля — массив функций, которые вызываются после всех встроенных проверок.

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

Правила работы:

- validators — массив функций.
- Функция должна вернуть:
  - true / null / undefined — значение корректно;
  - string — текст ошибки;
  - { message, code } — объект ошибки.
- Валидаторы выполняются только для видимых и редактируемых полей.
- Пользовательские валидаторы не заменяют встроенные, а **дополняют их**.

Межполевые (строчные) валидаторы являются внутренним механизмом ABGrid Engine и не относятся к публичному API.

11.6.3 Пользовательская валидация

При необходимости разработчик может написать собственную функцию проверки перед сохранением записи. Механизм schema.validators.row **дополняет** встроенную валидацию и выполняется **после** валидации отдельных полей, перед отправкой на сервер.

Например, имеются 2 поля, причем ситуация такова, что они или оба могут быть пустыми, но если заполнено какое-то одно, то и второе должно так же иметь значение. Тогда кастомная валидация может быть такой:

```
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 'Поля A и B должны быть заполнены вместе';
      }
      return true;
    }
  ]
}
```

11.7 Загрузка файлов из формы редактирования.

ABGrid Engine поддерживает отправку файлов через multipart/form-data в create/update сценариях, если форма редактирования содержит файловые поля.

Фактический multipart-контракт v1: запрос отправляется как FormData; в части meta передаётся JSON-метаданные операции; бинарные файлы передаются отдельными частями с именами вида file_<fieldName>.

Схема примерно такая:

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

Важно: при multipart-запросе компонент не должен принудительно задавать Content-Type: application/json; boundary формируется браузером автоматически; внутренние служебные поля ABGrid из meta удаляются до отправки.

Особенность v1: в multipart-режиме CRUD-полезная нагрузка для одной записи отправляется как одиночный объект row, а не как стандартный массив rows. Это известная особенность текущей версии v1. Серверный DTO должен ожидать именно row в meta для одиночной операции загрузки файлов.

Практическое замечание для серверной стороны: удобно принимать meta как @RequestPart("meta") DTO/Json и файлы как отдельные @RequestPart("file_<alias>") части.

Пример для 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;
        }
    }
```

Пример контроллера:

```
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. прочитать данные строки из row
            // 2. сохранить файл(ы)
            // 3. создать запись в БД
            // 4. вернуть id созданной записи
```

```
            Long createdId = 101L;

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

            if (avatarFile != null && !avatarFile.isEmpty()) {
                // сохранить второй файл
            }

            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())
{
                // заменить/сохранить файл
            }

            if (avatarFile != null && !avatarFile.isEmpty()) {
                // заменить/сохранить файл
            }

            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 Практический пример schema

Ниже пример небольшого набора полей, демонстрирующий дефолты и переопределения по режимам:

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

В этом примере:

- editor.requiredByDefault=false — по умолчанию поля не обязательны
- name.required=true — обязательное поле
- id в edit readonly, а в create скрыто
- createdAt заполняется сервером и скрыто при создании

12. Протокол обмена с сервером

12.1 dataKey: обертка запроса

Если параметр options.data.dataKey задан, то запросы на сервер оборачиваются в объект с этим ключом. Пример: { "payload": { ... } }. Ключ может используеться на серверной стороне для идентификации запросов из разных источников (не обязательно ABGrid Engine) при централизованных запросах на один url.

12.2 Загрузка данных (load)

Для load() используется строгий режим: сервер должен вернуть envelope вида { success:true, message?, data:{ rows, recordCount?, page?, rpp? } }.

Минимальный пример ответа:

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

Поля data:

- rows — массив строк.
- recordCount — общее количество строк (если не задано, используется rows.length).
- page / rpp — опционально: сервер может вернуть фактические значения, они синхронизируются с состоянием грида.
- summary – итоговая информация по странице и по выборке в целом.(Сумма, среднее значение, количество)

12.3 CRUD-операции и Result

При работе с сервером все CRUD-операции происходят с использованием только одного метода, по умолчанию POST и обязаны возвращать объект Result единого формата (для встроенного CrudEngine и для model.repository).

Result — единый контракт между UI, grid и сервером. Даже при локальных данных рекомендуется возвращать Result-объект.

Result:

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

- success — результат выполнения операции сервером (также принимается alias commit=true | false). Рекомендуется использовать success.
- message — текстовое сообщение (для UI/toast и диагностики).
- opId — идентификатор операции в браузере. Обязательный параметр для возврата сервером для подтверждения операции. Используется для обновления, создания записей в целях обновления состояния View на стороне клиента(браузера). Изначально отправляется клиентом на сервер для однозначной идентификации операции.
- data — полезная нагрузка (например, созданная/обновленная строка, метаданные).
- raw — исходный ответ сервера (для серверных операций).

Важно: ABGrid Engine может осуществлять первичную валидацию данных, но не заменяет серверную проверку прав и валидацию. Сервер остается единственным источником истины.

При использовании внутреннего редактора компонент при сохранении данных отправляет на сервер только diff – т.е. только те поля, значения которых были изменены пользователем.

В режиме создания новой записи на сервер отправляется внутренний идентиффикатор `__clientId` и идентификатор операции `opId`. Сервер при успешной операции вставки записи обязан вернуть этот opId, присвоенный идентификатор строки в параметре «id» (по умолчанию). Возвращать __clientId в версии компонента 1.0.0 не обязательно. Данный параметр введен под будущие версии для batch create. Но уже в версии 1.0.0 с той же целью на сервер новые строки отправляются в массиве rows.

Пример запроса на сервер при создании записи:

```
gridUsers: {
   format:'object',
   oper:'create',
```

```
    opId:' mks82ecz-1-3gbf8',
    rows:[
      {  __clientId: "c_mks7afwn_xbhov4at",
         name: "c_mks7afwn_xbhov4at"
      }
    ]
}
```

При редактировании записи на сервер __clientId, естественно, не формируется и не отправляется, для идентификации строки на стороне сервер отправляются измененные поля и их значения, а так же идентификатор строки.

Пример запроса на сервер при редактировании записи:

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

При удалении записей на сервер отправляется массив id удаляемых строк в параметре-массиве rowIds, даже при удалении всего одной записи. Сделано так намеренно для унификации обработки операции на стороне сервера и для batch delete.

Пример запроса на сервер при удалении записи\записей:

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

При возврате сервером изменных или вставленных строк движок компонента обновляет их во внутреннем репозитории, в самой сетке и в форме редактирования без перезагрузки всей страницы.

13. Публичный API

ABGrid Engine предоставляет стабильный публичный API для управления компонентом извне. Все методы предназначены для использования во внешнем коде (контроллеры, формы, страницы).

13.1 Жизненный цикл

• destroy() — уничтожает грид и освобождает ресурсы
• load(opts) — перезагружает данные (с сохранением или сбросом страницы)
• refresh() — перерисовывает UI без загрузки данных

13.2 Работа с данными

• load(opts) – загрузить данные с опциями
• getRows() - получить все строки
• getRowById(id) – получить объект строки по id
• getCurrentRow() - получить объект текущей строки
• setCurrentRow(id) – устанавливает текущей строку со значеним идентификатора id
• getCheckedRowIds() – вернуть массив строк, отмеченных чекбоксами.

13.3 CRUD API

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

Все методы CRUD возвращают объект Result.

13.4 Работа с фильтрами и сортировкой

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

13.5 Навигация и UI

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

13.6 События и обработчики

Грид поддерживает регистрацию обработчиков:
• on(event, handler)
• off(event, handler)

Основные события:
• data:loading
• data:loaded
• row:select
• row:deselect
• crud:success
• crud:error

События строк: row:click, row:dblclick, row:current, row:check.

События загрузки: load:start, data:loaded, load:success, load:error, load:end.

События subgrid: subgrid:opened, subgrid:closed.

События сортировки и toolbar: sort:changed, toolbar:change.

События intents: intent, а также само action-имя как событие (например, sort:apply), если оно эмитится через dispatch-намерения.

13.7 Публичный API (подробно)

Ниже приведён перечень публичных методов экземпляра грида (GridCore / ABGrid), которые предназначены для использования из внешнего кода приложения. Таблица собрана по исходникам архива и отражает фактические сигнатуры, параметры и возвращаемые значения.

Примечания:
• Методы, возвращающие Promise, являются асинхронными (их можно await).
• Типы приведены как JavaScript/DOM типы (string/number/boolean/object/Array/Function/HTMLInputElement/AbortSignal и т.п.).
• Если параметр допускает несколько форматов (например объект или serverRow-массив), это указано в описании.

Метод	Параметры (тип, описание)	Возвращает
on(name, fn)	name: string — Имя события. fn: (payload:any)=>any — Обработчик события. Если вернёт false и payload поддерживает preventDefault(), будет отменено действие по умолчанию.	Function — функция отписки (() => void).
off(name, fn)	name: string — Имя события. fn: Function — Тот же обработчик, который ранее был подписан.	void — ничего не возвращает.
msg(text, title = '', variant = 'info')	text: any — Текст сообщения (будет приведён к строке).	Promise<void> — резолвится после закрытия диалога.

	title: string — Заголовок (опционально). variant: string — Вариант/тип сообщения (например 'info'/'success'/'error').	
confirm(text, title = '')	text: any — Текст вопроса (будет приведён к строке). title: string — Заголовок (опционально).	Promise<'yes'|'no'|'cancel'> — результат выбора в диалоге подтверждения.
async request({ url = null, method = null, data = null, headers = null, params = null, body = null, signal = null, dataKey = null, strict = false, unwrapData = true, intent = 'user:request', interceptors = null } = {})	args: object — Параметры запроса (см. поля ниже). args.url: string|null — URL. Если не задан — может использоваться дефолтный из HttpClient/настроек. args.method: string|null — HTTP-метод (например 'GET','POST'). args.data: object|null — JSON payload (если используется). args.headers: object|null — HTTP заголовки. args.params: object|null — Query-параметры, будут добавлены в URL. args.body: any|null — Сырой body (если	Promise<any> — strict: возвращает объект data; non-strict: возвращает «развёрнутый» payload/результат.

	нужно отправить не-JSON). args.signal: AbortSignal|null — AbortController.signal для отмены запроса. args.dataKey: string|null — Ключ для unwrap данных (переопределяет dataKey по умолчанию). args.strict: boolean — Если true — ожидается контракт {success/commit,message,data} и возвращается data. args.unwrapData: boolean — Если true — при не-strict может извлекаться dataKey. args.intent: string|null — Строковый идентификатор намерения/контекста запроса (для логов/интерцепторов). args.interceptors: object|null — Интерцепторы (before/after/error и т.п.), будут смерджены с дефолтными.	
async load(opts = {})	opts: object — Опции загрузки.	Promise<void> — выполняет загрузку данных и

	opts.resetPage: boolean — Если true — сбрасывает страницу на 1 перед загрузкой (полезно для деталей/субгридов). opts.preserveTree / opts.preserveTreeState : boolean — Если true — не сбрасывает состояние дерева при перезагрузке (treeGrid).	обновляет состояние/рендер (результат через события load:*).
refresh()	—	Promise<void> — то же что load({resetPage:false}).
setReadOnly(readOnly)	readOnly: boolean — Режим readOnly: блокирует write-действия (create/update/delete).	this — возвращает экземпляр грида (для чейнинга).
getReadOnly()	—	boolean — текущий флаг readOnly (учитывает options.readOnly).
setData(data)	data: object — Данные для локальной установки без сетевого запроса. Формат: {rows:Array, page?:number, rpp?:number, recordCount?:number}.	void — устанавливает данные и перерисовывает, если нужно.
getData()	—	object — снимок данных/пейджинга. Если pager отключён: {rows}.

		Иначе: {rpp,page,rows,recordCount}.
setServerData(rows, total)	rows: Array — Строки в server-format (array) или external-format (object) — будут нормализованы. total: number — Общее количество записей (recordCount).	void — локально устанавливает server data (без запроса).
resetRowFormat()	—	void — сбрасывает форматирование/приведение serverRow (используется при изменении схемы/индексов).
setCurrentRow(rowId, { emit = true } = {})	rowId: string|number — Идентификатор строки (значение PK). options: object — Опции. options.emit: boolean — Если true — эмитит событие выбора текущей строки.	void — устанавливает текущую строку.
clearCurrentRow({ emit = true } = {})	options: object — Опции. options.emit: boolean — Если true — эмитит событие сброса текущей строки.	void — очищает текущую строку.
setFilter(filterObjOrFn)	filterObjOrFn: object|Function — Либо объект	this — возвращает экземпляр грида (для чейнинга).

	фильтра, либо функция (grid)=>object для вычисления фильтра на лету. Пользовательская функция для установки фильтра должна вернуть объект фильтра.	
setPage(page, { load = true } = {})	page: number — Номер страницы (1..totalPages). options: object — Опции. options.load: boolean — Если true — автоматически вызывает grid.load().	void.
setRpp(rpp, { load = true } = {})	rpp: number — Rows per page. options: object — Опции. options.load: boolean — Если true — автоматически вызывает grid.load().	void (также сбрасывает page=1).
getTotalPages()	—	number — количество страниц (ceil(total/rpp), минимум 1).
getVisibleColumnCount()	—	number — количество видимых колонок.
getCellValueByAlias(rowId, alias)	rowId: string|number — Идентификатор строки (PK).	any|null — значение ячейки или null, если строка/колонка не найдена.

	alias: string — Алиас (имя) поля/колонки.	
getRowObjectById(rowId)	rowId: string|number — Идентификатор строки (PK).	object|null — объект строки в external-format или null.
buildRequestData(overrides = {})	overrides: object — Частичные переопределения для результирующего объекта запроса.	object — requestData для запроса (oper,page,rpp,sortOrder,filter, treeLevels?, ... + overrides).
buildRequestPayload(overrides = {})	overrides: object — То же что buildRequestData().	object — request payload (в текущей версии совпадает с buildRequestData).
render()	—	void — выполняет перерисовку грида (рендер DOM).
renderTreeCell({ value, column, rowObject })	args: object — Аргументы рендера. args.value: any — Значение ячейки. args.column: object — Описание колонки (column config). args.rowObject: object — Объект строки (external-format).	string — HTML/текст ячейки для дерева (если treeGrid выключен — возвращает value как строку/значение).
bindAutocomplete(inputEl, options = {})	inputEl: HTMLInputElement — Инпут, к которому подключается autocomplete. options: object — Опции autocomplete (url, debounceMs, limit, minChars, mapping и т.д.).	object — handle { destroy, open, close, clear, setValue, getValue }.

deleteRowById(rowId)	rowId: string|number — Идентификатор строки (PK).	boolean — true если строка была реально удалена локально.
deleteRowsByIds(rowIds)	rowIds: Array<string|number> — Массив идентификаторов строк (PK).	number — количество реально удалённых строк.
deleteCheckedRows()	—	Array<string> — массив rowId реально удалённых строк (по состоянию checked).
updateRowById(rowId, rowObjectOrServerRow)	rowId: string|number — Идентификатор строки (PK). rowObjectOrServerRow: object|Array — Либо объект строки (external-format), либо serverRow (array).	boolean — true если строка найдена и обновлена локально.
createRow(rowObjectOrServerRow, opts = {})	rowObjectOrServerRow: object|Array — Либо объект строки (external-format), либо serverRow (array). opts: object — Опции вставки. opts.position: 'top'|'bottom'|'afterCurrent' — Куда добавить строку (по умолчанию — bottom).	string|null — rowId добавленной строки или null, если ничего не добавлено.

createRows(rows, opts = {})	rows: Array<object|Array> — Массив строк (external-format) или serverRows (array). opts: object — Опции вставки. opts.position: 'top'|'bottom'|'afterCurrent' — Куда добавить строки.	Array<string> — массив rowId добавленных строк.
onToolBar(handler)	action,ctx,grid	action <string> - ключ(имя) действия(кнопки), ctx - контекст, grid –инстанс компонента
destroy()	—	void — освобождает ресурсы: субгриды, детали, обработчики DOM/событий и т.д.
getDetailGrid(containerId)	containerId – строка контейнера без # детальной сетки или number - индекс массива с конфигурацией детальных сеток detailGrids[]	grid-инстанас детальной сетки
getCheckedRowIds()		Возвращает массив отмеченных строк или пустой массив, если их нет
handleUnauthorized(response)	response	Глобальная обработка ответа сервера со статусом 401

14. Интерцепторы (request/response/error)

Раздел описывает механизм интерцепторов ABGrid Engine: перехват запросов и ответов внутреннего HTTP-клиента для реализации сквозной логики (например, автоматическое применение policy).

Интерцепторы в ABGrid Engine — это механизм перехвата и обработки запросов к серверу и ответов от сервера на уровне движка объекта ABGrid Engine. Они позволяют централизованно реализовывать сквозную логику, не привязываясь к конкретным операциям (load, create, update, delete). Интерцепторы задаются массивом (или объектом, если интерцептор каждого типа всего один). При необходимости пользователь может написать множество интерцепторов одного типа с разными задачами и все они будут обработаны в рамках одного запроса\ответа в порядке очередности, прописанной при задании их в массиве.
Зачем нужны интерцепторы.
Интерцепторы используются для:
• автоматической обработки политики прав (policy)
• централизованного анализа ответов сервера
• внедрения общих правил поведения для всех сеток
• уменьшения дублирования кода в CRUD-операциях

Интерцепторы работают прозрачно для пользователя компонента и не требуют явного вызова в конфигурации каждой операции. Конфигурирование интерцепторов осуществляется в параметре data и в общем случае выглядит так:

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

где fn*-пользовательские функции, срабатывающие в порядке их написания

Интерфейс интерцепторов:

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

14.1 Интерцепторы запросов (request)

Интерцептор запросов вызывается перед отправкой HTTP-запроса на сервер.

В текущей архитектуре ABGrid Engine интерцепторы запросов:
• не изменяют логику прав доступа
• не добавляют служебные параметры, связанные с policy

Это сделано намеренно: сервер сам решает когда и какие права вернуть клиенту. ABGrid Engine не сигнализирует серверу о необходимости прислать политику.

Пример практического применения.
Задача: На форме расположен глобальный фильтр, например, выбор организации, её тип и др. Необходимо при каждом запросе передавать на сервер значения этого глобального фильтра (в том числе при инициализации страницы и при автозагрузке компонента-сетки при этом).
Решение:
Функция сбора данных глобального фильтра:
```
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};
 }
```

Конфигурация компонента:
```
data: {
      interceptors: {
        request: buildUsersFilter,
      },
   }
```

Еще пример использования (передаем на сервер состояние чекбокса):

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

14.2 Интерцепторы ответов (response)

Интерцептор ответов вызывается для КАЖДОГО ответа сервера, полученного через внутренний HTTP-клиент грида. Основная задача интерцептора ответов — анализ стандартного envelope ответа сервера и применение глобальной логики.

Параметр payload в интерцепторы данного вида передаются по значению и могут быть подменены в коде пользователя.

```
response(ctx, payload) {
     ...
     return payload;        // ✓ можно заменить
}
```

Автоматическое применение политики прав

Если в конфигурации грида включена опция `policy:{ enabled: true },` то интерцептор ответов выполняет следующую логику:

1. Проверяет наличие объекта data.policy в ответе сервера
2. Если policy присутствует — вызывает grid.setPolicyTree(data.policy)
3. Политика сохраняется целиком во внутреннем состоянии грида
4. Политика автоматически применяется к:
   - мастер-сетке
   - детальным сеткам
   - субгридам

Если `data.policy` отсутствует, никаких изменений прав не происходит.

Ручная установка политики

Метод `setPolicyTree(policyTree)` является публичным API и может вызываться вручную из пользовательского кода.

Важно:
• `setPolicyTree()` работает независимо от опции policy.enabled
• `policy.enabled` управляет ТОЛЬКО автоматическим применением политики из ответов сервера
Почему policy.enabled может быть отключён.
Опция policy.enabled может быть отключена в следующих случаях:
• политика прав приходит из другого API (например, /me или JWT)
• используется мок-сервер или оффлайн-режим
• политика прав слишком объёмная и не должна передаваться в каждом ответе
• интеграция требует полного контроля со стороны приложения

При `policy.enabled = false` грид полностью игнорирует `data.policy` из ответов сервера.
Связь интерцепторов с readOnly
Важно: интерцепторы не управляют режимом readOnly.
Режим readOnly:
• является локальным режимом работы грида
• управляется через конфигурацию или методы `setReadOnly()/getReadOnly()`
• имеет приоритет над `policy`

Даже если `policy` разрешает CRUD-операции, `readOnly=true` запрещает любые изменения данных.
Итак:
• сервер остаётся источником истины
• клиент хранит состояние прав самостоятельно
• политика прав применяется единообразно и централизованно

14.3 Интерцепторы ошибок(error)

Интерцепторы данного типа вызываются **в случае ошибки выполнения операции**, после попытки выполнения через контроллер (встроенный или внешний), но **до** стандартной реакции грида (сообщения, rollback, завершения операции). Пользователь может обрабатывать ошибки запроса, установив один и несколько интерцепторов с типом «error». В обработчик передаются 2 параметра –контекст ctx и объект ошибки error. Параметр ctx передается по ссылке и его можно анализировать, логировать, расширять. Параметр error жестко не стандартизирован, что дает большую гибкость для пользователя.

Порядок выполнения:

1. controller (internal/external)
2. error-interцепторы
3. стандартная реакция грида (toast/msg/rollback)

14.4 Интерцепторы error. Рекомендации

Обязательно

- есть сервер
- есть права доступа
- есть бизнес-ошибки
- есть разные типы пользователей

Очень полезно

- централизованная обработка ошибок
- единый UX
- логирование
- аналитика

Можно не использовать

- если данные локальные
- если всё «просто»
- если хватает дефолтных сообщений

Чего не стоит делать в error-интерцепторе

❌ Пытаться повторно выполнять CRUD
❌ Менять данные грида вручную
❌ Делать UI-действия на «авось»
❌ Полагаться на конкретный формат error без проверок

Примеры использования

- логирование

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

- Кастомная обработка HTTP-кодов

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

  if (err?.status === 403) {
    grid.msgError('Недостаточно прав');
  }
}
```

- Подмена\нормализация ошибок сервера

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

- Глобальное UX-поведение

```
error(ctx, err) {
  if (ctx.op === 'load') {
    grid.ui.toast.show('Не удалось загрузить данные', 'error');
  }
}
```

- Подавление стандартного поведения

```
error(ctx, err) {
  return false; // сказать гриду: "я обработал ошибку сам"
}
```

14.5 Итог

Интерцепторы в ABGrid Engine обеспечивают чистую и масштабируемую архитектуру, являясь единственной точкой вмешательства пользователя в запросы и ответы сервера. Это позволяет использовать ABGrid Engine как в простых проектах, так и в сложных enterprise-интеграциях.

15. Режим Master / Detail режим подробней

ABGrid Engine поддерживает сложные иерархические сценарии отображения данных:

master–detail, subgrid (детальные данные в строке), treeGrid, detailsPanels. Все режимы могут сочетаться между собой. Никаких ограничений на сложность иерархии не существует, компонент поддерживает весь необходимый функционал автоматически.

Master-grid управляет детальными сетками. При смене текущей строки master:
• извлекает значение masterField из текущей строки
• устанавливает фильтр в детальной сетке с параметром detailField со значением поля masterField
• инициирует загрузку данных детальной сетки

Детальная сетка не подписывается на события мастера самостоятельно. Инициатором загрузки всегда является master-grid. Детальная сетка поступает аналогичным образом со своими детальными сетками (если заданы), процесс отрабатывает каскадно и останавливается при условии того, что у детальных сеток отсутствуют свои детальные сетки. Процесс полностью автоматический, задается исключительно конфигурацией и не требует написания какого-либо дополнительного кода для его управления. Мастер-сетка сама создает детальные сетки из параметров конфигурации, однако div-элементы уже должны быть на странице. Не запрещается создавать детальные сетки до создания мастер-сетки, но и не рекомендуется. Мастер-компонент при создании детальных проверяет их существование и не создает новые при их наличии.

15.1 Сочетание режимов

ABGrid Engine допускает комбинации режимов:
• treeGrid + detailGrids
• master/detail + subgrid
• treeGrid + subgrid

Логика загрузки данных остаётся неизменной:
• master инициирует загрузку зависимых сеток
• каждый grid управляет только своими прямыми дочерними компонентами

16. Режим ReadOnly

Режим ReadOnly предназначен для перевода объекта ABGrid Engine в состояние «только просмотр». В этом режиме пользователь может просматривать данные, осуществлять навигацию по страницам, сортировать и фильтровать записи, но любые операции изменения данных запрещены.

16.1 Назначение

ReadOnly в ABGrid Engine является глобальной политикой компонента. Он не относится к отдельному редактору или форме, а описывает политику операций изменения данных в целом.
Ключевая идея:
• данные можно читать
• данные нельзя создавать, изменять или удалять

16.2 Как включить ReadOnly

Режим ReadOnly задаётся в корне конфигурации грида:

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

Это единый источник истины для всего грида, включая:
• встроенный editor
• toolbar
• публичные CRUD-методы
• внешний контроллер

16.3 Что разрешено в режиме ReadOnly

В режиме ReadOnly разрешены следующие действия:

- загрузка и обновление данных (load, reload)
- навигация по страницам
- сортировка и фильтрация
- выбор строк (current / checked)
- открытие форм в режиме просмотра

16.4 Что запрещено в режиме ReadOnly

В режиме ReadOnly запрещены любые операции изменения данных:

- create (создание новых записей)
- update (редактирование существующих записей)
- delete / deleteMany (удаление одной или нескольких записей)

При попытке выполнения запрещённой операции:
• запрос на сервер не отправляется
• операция завершается локально
• возвращается объект Result с признаком ошибки

16.5 ReadOnly и UI

Режим ReadOnly не меняет структуру UI автоматически.
Кнопки toolbar могут оставаться видимыми, но действия будут заблокированы.

Такой подход позволяет:
• использовать единый layout для разных ролей
• управлять видимостью кнопок отдельно от политики данных

16.6 ReadOnly и Editor

Встроенный editor подчиняется режиму readOnly:
• формы могут открываться для просмотра
• кнопки сохранения не выполняют действия
• поля могут быть переведены в readonly/disabled состояние

16.7 ReadOnly и внешний контроллер

Все публичные CRUD-методы компонента учитывают режим readOnly. Это означает, что внешний контроллер не может обойти ограничения, даже если вызывает create/update/delete напрямую.

16.8 Рекомендации использования

Рекомендуемые сценарии применения ReadOnly:

- режим просмотра справочников
- ограниченный доступ для пользователей без прав редактирования
- временная блокировка изменений (maintenance mode)
- использование объекта компонента как view-компонента

17. UI-intents

UI-intents – это то, как модуль компонента View взаимодействет с приложением.
Intent — это декларативное уведомление о пользовательском намерении.
ABGrid не решает, *что делать*, он сообщает, *что пользователь хотел сделать*.

Важно:

- intents генерируются всегда
- режим controller влияет только на обработку, не на генерацию

Общая схема:

UI действие
↓
intent { action, payload }
↓
controller (internal или external)
↓
(опционально) server
↓
обновление data / refresh

17.1 Список стандартных intents

Это **очень важно**. Пример:

- 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`

Важно:

Список intents является частью публичного API компонента.

Пример 1. Внешний контроллер

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

ABGrid не загружает данные самостоятельно, а уведомляет приложение о намерении пользователя.

Пример 2. Чистый 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 Чего intents не делают

Важно:

- intent ≠ HTTP запрос
- intent ≠ CRUD операция
- intent ≠ бизнес-логика

18. Глобальные статические helpers

ABGrid предоставляет несколько статических helper-методов и свойств.

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

Помощники даты/времени:

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

ABGrid.now()   // ISO datetime,

ABGrid.timestamp()   // Unix timestamp в миллисекундах.

Статический autocomplete API:

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

Назначение: быстро получить версию подключённой сборки; использовать единые helper-функции дат в конфигурации и формах; подключать autocomplete без создания встроенного editor.

Пример из админки сайта компонента.

В редакторе устанавливаем в поле date текущие значения

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

19. Типовые сценарии использования

В этом разделе приведены практические сценарии использования ABGrid Engine.
Примеры ориентированы на реальные прикладные задачи и демонстрируют рекомендуемые архитектурные подходы.

19.1 Master + Detail (классический справочник)

Сценарий:
• master-грид отображает список сущностей (например, заказы)
• detail-грид отображает связанные данные (позиции заказа)
• master управляет загрузкой 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' }
         }
        }]
      });
```

При смене текущей строки в ordersGrid:
• извлекается значение поля id
• в detail-grid устанавливается фильтр
• detail-grid перезагружается

19.2 Master + внешняя форма редактирования

Сценарий:
• грид используется только для выбора записи
• редактирование выполняется во внешней форме
• внешний код управляет 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 Использование external controller

Сценарий:
• ABGrid не выполняет запросы напрямую
• вся логика загрузки и сохранения данных находится во внешнем коде

```
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 Грид в режиме ReadOnly

Сценарий:
• грид используется как компонент просмотра
• изменения данных запрещены

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

19.5 TreeGrid (иерархический справочник)

Сценарий:
• одна сетка отображает иерархические данные (категории, структуры, подразделения)
• используется режим view.treeGrid для визуализации дерева
• источник данных может быть как плоским (parentId), так и заранее подготовленным сервером

Пример (adjacency list / parentId):

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

Примечания:
• TreeGrid — это режим отображения одной таблицы, а не detail-grid.
• Загрузка children может выполняться:
  – одним запросом (все узлы сразу)
  – лениво (по раскрытию узла), если сервер поддерживает отдельный endpoint.
Конкретный режим зависит от реализации data.controller и вашего API.
TreeGrid можно комбинировать с detailGrids:
• master (treeGrid) выбирает узел
• detail-grid показывает связанные записи выбранного узла

19.6 Использование системных диалогов в пользовательском коде

ABGrid Engine предоставляет унифицированные системные диалоги, которые могут вызываться из пользовательского кода.
Они используются для:

- отображения сообщений пользователю;
- запроса подтверждения действий;
- унификации UI (единый стиль, i18n, поведение);
- работы в асинхронных сценариях (через Promise).

Системные диалоги доступны через методы экземпляра грида.

19.6.1 Диалог сообщений grid.msg()

Назначение

Метод `msg()` используется для отображения информационных, предупреждающих и ошибочных сообщений пользователю.

Диалог **блокирует выполнение пользовательского сценария** до закрытия и возвращает `Promise`.

Сигнатура

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

Параметры

- text — текст сообщения (любой тип, будет приведён к строке)
- title *(необязательный)* — заголовок диалога
- variant *(необязательный)* — тип сообщения
  (например: "info", "success", "warning", "error")

Пример 1. Простое информационное сообщение

grid.msg('Данные успешно сохранены');

Пример 2. Сообщение с заголовком

```
grid.msg(
    'Изменения вступят в силу после перезагрузки страницы',
    'Внимание'
);
```

Пример 3. Сообщение об ошибке

```
grid.msg(
    'Ошибка при сохранении данных',
    'Ошибка',
    'error'
);
```

Пример 4. Использование с await

```
await grid.msg(
    'Операция завершена',
    'Готово',
    'success'
);
```

// код выполнится только после закрытия диалога

```
console.log('Диалог закрыт');
```

19.6.2 Диалог подтверждения grid.confirm()

Назначение

Метод `confirm()` используется для запроса подтверждения действия у пользователя (удаление, массовые операции, необратимые изменения и т.п.).

Метод **всегда асинхронный** и возвращает результат выбора пользователя.

Сигнатура

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

Возвращаемые значения

- 'yes' — пользователь подтвердил действие
- 'no' — пользователь отказался
- 'cancel' — пользователь закрыл диалог или нажал «Отмена»

Пример 1. Простое подтверждение действия

```
const result = await grid.confirm('Удалить выбранную
запись?');

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

Пример 2. Подтверждение с заголовком

```
const answer = await grid.confirm(
    'Все выбранные записи будут удалены без возможности
восстановления.',
    'Подтверждение удаления'
);

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

// продолжение операции

Пример 3. Использование в обработчике кнопки тулбара

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

    const res = await grid.confirm(
        'Вы уверены, что хотите удалить записи?',
        'Удаление'
    );

    if (res !== 'yes') {
        // отменяем действие по умолчанию
        payload.preventDefault();
        return false;
    }
});
```

Пример 4. Подтверждение перед пользовательским запросом

```
async function deleteWithConfirm(rowId) {
    const res = await grid.confirm(
        'Удалить запись?',
        'Подтверждение'
    );

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

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

    grid.msg('Запись удалена', 'Готово', 'success');
}
```

19.6.3 Диалог подтверждения grid.confirmEx()

Назначение

Метод `confirmEx()` используется для запроса подтверждения действия у пользователя (удаление, массовые операции, необратимые изменения и т.п.). Отличается от confirm() расширенными настройками.

Позволяет:

- задать стиль диалога (`variant`)
- полностью настроить кнопки
- управлять фокусом
- добавить чекбокс (например: “Не спрашивать больше”)
- получить структурированный результат

Метод **всегда асинхронный** и возвращает результат выбора пользователя в виде объекта.

Сигнатура

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

Параметр	Тип	Описание
`message`	`string`	Текст сообщения
`title`	`string`	Заголовок диалога
`options`	`object`	Дополнительные параметры

Параметры options

Параметр	Тип	Описание
`variant`	'default'	'danger'
`buttons`	`Array`	Массив кнопок
`focus`	'yes'	'cancel'`
`checkbox`	`object`	Настройка чекбокса

Параметр **variant** управляет визуальным стилем окна подтверждения. Поддерживаются следующие значения:

Значение	**Назначение**
`'default'`	Обычный нейтральный диалог
`'danger'`	Разрушающее действие (удаление, очистка, сброс)
`'warning'`	Потенциально опасное действие
`'info'`	Информационное подтверждение

Формат кнопки:

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

Формат checkbox:

```
{
  text: 'Не спрашивать больше',
  checked: false,
  position: 'buttons-left', // optional
  onChange: (checked) => {}
}
```

Возвращаемое значение – объект:

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

Минимальный вызов

Пример 1:

```
await grid.confirmEx('Удалить запись?');
```

Пример 2 (с указанием варианта):

```
await grid.confirmEx('Удалить запись?', 'Подтверждение', {
variant: 'danger' });
```

Поведение по умолчанию

Если в параметре options не указаны buttons, focus или checkbox, компонент автоматически применяет стандартную конфигурацию.

Кнопки по умолчанию

- Создаются 2 кнопки:

  Кнопка подтверждения — value: 'yes'
  Кнопка отмены — value: 'cancel'

Тексты кнопок берутся из options.i18n.buttons (btnYes, btnCancel). Если i18n не настроен, используются встроенные значения.

Классы для кнопок:

- **`primary`** — это *роль/акцент* (может менять цвет в зависимости от variant)
- **`abgrid-btn-*`** — это *семантический класс кнопки* (danger/primary/neutral)
- Для destructive диалогов:
  - Confirm = `abgrid-btn-danger`
  - Cancel = `abgrid-btn-primary` (без `primary`)

Пример 3. Простое подтверждение действия

```
const r = await grid.confirmEx(
  'Удалить 10 записей?',
  'Подтверждение',
  {
    variant: 'danger',
    buttons: [
      { label: 'Удалить', value: 'yes', className: 'abgrid-
btn-danger' },
      { label: 'Отмена', value: 'cancel' }
    ],
    focus: 'cancel'
  }
);
if (r?.value === 'yes') {
  // действие
}
```

Пример 4. Пример с чекбоксом

```
const r = await grid.confirmEx(
  'Удалить все записи?',
  'Подтверждение',
  {
    variant: 'danger',
    checkbox: {
      text: 'Не спрашивать больше',
      checked: false
    }
  }
);

if (r?.value === 'yes') {
  if (r.checked) {
    // сохранить настройку "не спрашивать"
  }
}
```

Если встроенный UI-диалог недоступен, confirmEx() автоматически использует grid.confirm().

19.6.4 Рекомендации по использованию

- ❗ **Всегда используйте await** или .then() при работе с msg() и confirm(), confirmEx()
- ❗ Не используйте window.alert() и window.confirm() — системные диалоги:
    - поддерживают i18n;
    - единообразны по стилю;
    - корректно работают с асинхронными сценариями грида
- ✅ Используйте confirm(), confirmEx() **до выполнения действий**, а не внутри CRUD-логики
- ✅ В обработчиках событий можно отменять действие через payload.preventDefault()

19.7 Уведомления (toast)

ABGrid Engine поддерживает неблокирующие уведомления (toast-сообщения), предназначенные для информирования пользователя о ходе и результате операций без прерывания пользовательского сценария**.**

Toast-уведомления используются, когда:

- не требуется подтверждение действия;
- нельзя или не нужно блокировать интерфейс;
- важно показать статус операции (успех, ошибка, предупреждение, информация).

Уведомления отображаются поверх интерфейса и автоматически скрываются через заданный интервал времени.

19.7.1 Метод grid.toast()

Назначение

Метод `toast()` отображает краткое уведомление пользователю и **не блокирует выполнение кода**.

Сигнатура

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

Параметры

- text — текст уведомления
  (любой тип, будет приведён к строке)
- variant *(необязательный)* — тип уведомления
  Возможные значения (рекомендуемые):
  - "info"
  - "success"
  - "error"
- options *(необязательный)* — объект настроек отображения:
  - timeout: number — время показа в миллисекундах
    (по умолчанию используется глобальное значение)
  - closeable: boolean — отображать кнопку закрытия
  - position: string — позиция на экране
    (например: "top-right", "bottom-left" и т.п.)

19.7.2 Примеры использования

Пример 1. Информационное уведомление

```
grid.toast('Данные загружены');
```

Пример 2. Уведомление об успешной операции

```
grid.toast(
    'Запись успешно сохранена',
    'success'
);
```

Пример 3. Уведомление об ошибке

```
grid.toast(
    'Ошибка при выполнении операции',
    'error'
);
```

Пример 4. Уведомление с кастомными опциями

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

19.7.3 Использование в асинхронных сценариях

Toast-уведомления удобно использовать в цепочках асинхронных операций, где блокировка UI нежелательна.

```
try {
    await grid.request({
        url: '/api/save',
        method: 'POST',
        data: formData,
        strict: true
    });
    grid.toast('Данные сохранены', 'success');

} catch (e) {
    grid.toast('Ошибка сохранения', 'error');
}
```

19.7.4 Использование совместно с событиями грида

```
grid.on('load:success', () => {
    grid.toast('Данные успешно загружены', 'info');
});

grid.on('load:error', () => {
    grid.toast('Ошибка загрузки данных', 'error');
});
```

19.8 Когда использовать toast, а когда msg / confirm

Сценарий	Рекомендуемый инструмент
Требуется подтверждение действия	confirm()
Нужно заблокировать сценарий	msg()
Информирование без блокировки	toast()
Ошибка фоновой операции	toast('...', 'error')
Критическая ошибка	msg('...', 'Ошибка', 'error')

19.8.1 Диалоги и уведомления: UX-рекомендации

- ✅ Используйте toast() для **статусных и фоновых сообщений**
- ❗ Не используйте toast() для действий, требующих осознанного подтверждения
- ✅ Для ошибок CRUD-операций часто лучше сочетание:
  - toast() — для краткого уведомления
  - msg() — для подробного описания (по необходимости)
- ❗ Не злоупотребляйте количеством уведомлений — они не должны «заспамливать» пользователя

• confirm() — только перед действиями, изменяющими данные
• msg() — для блокирующих и критических сообщений
• toast() — для фоновых и статусных уведомлений
• error-интерцепторы — для централизованного UX

Не используйте toast для подтверждений и не блокируйте UI без необходимости.

19.8.2 Итог

Toast-уведомления в ABGrid Engine:

- не блокируют UI;
- легко встраиваются в асинхронный код;
- поддерживают единый стиль и i18n;
- идеально дополняют системные диалоги (msg, confirm).

19.9 Autocomplete для пользовательских input

ABGrid Engine позволяет использовать подсистему «autocomplete» не только во встроенном редакторе, но и подключать её к произвольным пользовательским input-элементам вне грида. Это особенно полезно при использовании внешних форм и кастомных UI, когда разработчик хочет повторно использовать механику поиска и выбора значений. По умолчанию ABGrid Autocomplete ожидает стандартный формат ответа сервера: items: [{ id, value }]. При использовании autocomplete для пользовательских input возможно указание маппинга полей ответа (valueField, textField), если сервер возвращает данные в произвольном формате.

- autocomplete в ABGrid Engine не жёстко привязан к внутреннему редактору
- любой input может быть зарегистрирован как источник autocomplete
- используется единый механизм запросов, debounce и обработки ответа

Пример подключения autocomplete к пользовательскому input:

```
<input type="text" id="companyInput" placeholder="Введите
название компании">
const input = document.getElementById('companyInput');

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

или

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

В этом примере:
• пользователь вводит текст в обычный input
• ABGrid Engine выполняет запрос autocomplete
• при выборе значения input заполняется текстовым значением
• идентификатор может быть сохранён разработчиком отдельно (data-атрибут, hidden input и т.п.)

Autocomplete, подключённый к пользовательскому input:
• не зависит от schema
• не требует наличия editor
• может использоваться совместно с внешним контроллером

20. Работа с внутренним HTTP-клиентом

ABGrid Engine позволяет отправлять произвольные операции на сервер через встроенный HTTP-клиент.

Это используется для:

- массовых операций (bulk)
- кастомных действий тулбара
- серверных команд (экспорт, смена статуса и т.д.)

20.1 Отправка запроса через grid.data.request()

Рекомендуемый способ — использовать внутренний механизм DataEngine:

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

if (!ids.length) {
  grid.toast('Нет выбранных строк', 'error');
  return false;
}

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

const response = await grid.data.request({
  requestData: {
    [operParam]: 'bulkDeleteSpam', // имя операции
    ids                            // массив id строк
  },

  unwrapData:false,      // переопределяем, см. ниже
  strict: false,
  intent: 'user:bulkDeleteSpam'
});

grid.toast ('Операция выполнена', 'success');
```

- берёт `url` из `options.data.url`
- использует внутренний HttpClient
- автоматически применяет:
  - headers
  - dataKey
  - unwrapData
- проходит через interceptors
- поддерживает loading/strict режим

Параметр unwrapData можно переопределить как false и тогда response будет содержать не только data, но и success и message.

20.2 Использование buildRequestPayload()

Если операция должна учитывать:

- текущий фильтр
- сортировку
- пагинацию
- treeGrid уровни

можно добавить:

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

20.3 Альтернативный способ — grid.request()

Если требуется:

- другой URL
- другой HTTP метод
- отдельный endpoint

можно использовать:

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

20.4 Параметр strict

Параметр strict управляет тем, насколько жёстко проверяется ответ сервера.

Используется для обычного `load()`.

Тогда:

- ожидается, что сервер вернёт:

        - `success`
        - `rows`
        - `total`
        - возможно `message`

- если структура не соответствует — будет считаться ошибкой
- DataEngine применяет стандартную логику:

    - обновляет rows
    - обновляет total
    - запускает перерисовку
    - показывает сообщения

Это “строгий контракт загрузки данных”.

Если `strict: false` (кастомные операции)

Используется для:

- bulk операций
- export
- смены статуса
- серверных команд
- любых нестандартных вызовов

В этом режиме:

- ответ сервера НЕ обязан содержать `rows`
- НЕ проверяется наличие `total`
- DataEngine НЕ пытается перерисовать таблицу
- возвращается “как есть” payload

Что будет, если strict оставить true для bulk?

DataEngine может:

- ожидать rows
- попытаться обновить данные
- среагировать как на обычный load
- показать “ошибку структуры ответа”

То есть поведение станет некорректным. Если очень коротко, то strict – это стандартная загрузка данных или нет.

Сводная таблица для установки данного параметра:

Сценарий	**strict**
load / reload	true
bulk операции	false
export	false
произвольная команда	false
сервер возвращает новые rows	true

21. Работа в режиме View

В режиме View ABGrid Engine выступает исключительно как визуальный компонент, генерирующий intents, но не принимающий решений о загрузке и сохранении данных.

В этом режиме ABGrid Engine выступает **только представлением**: компонент **ничего не решает про загрузку/CRUD**, а лишь:

- рендерит данные,
- хранит UI-состояние (страница, rpp, сортировка, фильтр),
- **эмитит "намерения" (intent)** о действиях пользователя.

Все решения (что грузить, как сортировать, что делать при кликах тулбара) остаются в вашем **Controller/Presenter**.

21.1 Ключевые настройки режима View

Чтобы превратить грид в "чистую View", в конфиге нужно:

1. **Отключить автозагрузку**

```
autoLoad: false
```

2. Переключить контроллер намерений в **external**

```
data: {
  controller: 'external',
  intents: ... // ваш обработчик
}
```

3. (Рекомендуется) Включить глобальный режим "только чтение", чтобы любые попытки записи были запрещены политикой грида:

```
readOnly: true
```

Важно: в external-режиме ABGrid **не делает сетевых запросов сам по себе**, пока вы не вызываете grid.load()/grid.refresh()/grid.request() вручную. Поэтому для View-режима обычно достаточно autoLoad:false и не дергать методы загрузки.

21.2 Как отдавать данные гриду из Controller

Контроллер в вашем приложении получает данные любым способом (API, стор, кэш, вычисления) и **обновляет View** через публичные методы:

- grid.setData(data) — установить данные напрямую (локально).
- grid.setServerData(rows, total) — установить строки + total (удобно для серверной пагинации).

Пример (MVC):

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

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

  data: {
    controller: 'external',

  // Единая точка намерений от View
  intents: ({ action, payload, grid }) => {
      switch (action) {

    // пагинация
        case 'pager:page': {
          const page = Number(payload.page);
            // обновим UI-состояние (без загрузки!)
          grid.setPage(page, { load: false });
            // дальше решает Controller:
          controllerLoadPage(page);
          break;
        }

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

    // сортировка
        case 'sort:preview': {
     // preview — только обновить индикаторы сортировки (без загрузки)
     if (payload.sortOrder !== undefined) {
            grid.state.sortOrder = payload.sortOrder;
          }
          break;
        }
```

```
        case 'sort:apply': {
  // apply — Controller решает как перезагрузить/пересчитать данные
          controllerApplySort(payload.sortOrder);
          break;
        }

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

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

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

 // В режиме "чистая View" эти действия обычно игнорируются,
 // либо вы делаете свою реакцию (например, открыть внешний диалог).
        case 'toolbar:add':
        case 'toolbar:edit':
        case 'toolbar:delete':
        default:
          break;
      }
    }
  },

 // ...остальные настройки колонок/рендера/UI
});


// Controller: первичная отрисовка
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 Список intents, которые эмитит View (и которые вы перехватываете)

В текущей реализации предусмотрены следующие action:

Toolbar

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

Pager

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

Sort

- `sort:preview` (только превью/индикаторы)
- `sort:apply` (подтверждение сортировки)

Filter

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

21.4 Практические рекомендации для чистого View

- Если вам нужен **строгий View-only**, ставьте `readOnly: true` и в external-контроллере **не обрабатывайте** `toolbar:add/edit/delete` (или показывайте свой toast "только просмотр").
- Для серверной пагинации всегда используйте:
  - grid.setServerData(rows, total) + grid.setPage(page, {load:false})
- Не вызывайте `grid.refresh()`/`grid.load()` в этом режиме, если не хотите сетевой активности от грида (с сетью должен работать ваш Controller).

22. ABGrid Engine — Events, Public API и Patterns использования

22.1 Система событий ABGrid (Event System)

ABGrid использует встроенную систему событий, которая позволяет реагировать на действия пользователя и изменения состояния грида.

Подписка на событие выполняется через метод:

```
grid.on(eventName, handler)
```

Пример:

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

22.2 Основные события

row:current — изменение текущей строки
data:loaded — данные загружены
data:error — ошибка загрузки данных
crud:success — успешная операция CRUD
crud:error — ошибка CRUD операции
grid:ready — грид полностью инициализирован

22.3 Public API Grid

Основные методы публичного API грида:

22.3.1 Методы работы с данными

grid.reload() — перезагрузка данных

grid.getRowData(rowId) — получение данных строки

grid.getCurrentRowId() — получить текущую строку

grid.getCheckedRowIds() — получить выбранные строки

22.3.2 Методы панели инструментов

grid.toolbar.getValue(id) — получить значение элемента
grid.toolbar.setValue(id, value) — установить значение элемента

22.3.3 Методы работы с деталями

grid.details.showForRow(rowId) — показать панели для строки
grid.details.clear() — очистить панели

22.4 Patterns использования 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

Используется для отображения иерархических данных.
Структура:
id
pid
name

22.6 Admin panel pattern

Типичная структура:

- грид со списком записей
- toolbar с действиями
- встроенный редактор записей
- master-detail связи

23. ABGrid Engine — пример реализации: Master → Detail → Detail

23.1 Общая архитектура интерфейса

Этот пример демонстрирует типичную структуру административной панели, где используется несколько связанных гридов.

Структура:

Users (master)
  ↓
Licenses (detail)
  ↓
Product Versions (detail-detail)

Выбор строки в родительском гриде автоматически обновляет дочерний грид.

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 Связывание гридов (каскад обновления)

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

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

23.6 Каскадная очистка деталей

При смене строки важно очищать нижние уровни, чтобы не отображались устаревшие данные.

Пример:

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

23.7 Результат

Такой подход позволяет построить мощные административные интерфейсы:

- управление пользователями
- управление лицензиями
- доступные версии продукта
- вложенные зависимости данных

Важно:
ABGrid автоматически обрабатывает выбор строки, очистку детальных сеток и загрузку данных, что значительно упрощает реализацию сложных интерфейсов.

24. Интеграция с React

24.1 Обзор

ABGrid Engine предоставляет официальный адаптер для работы с React-приложениями.

React-обвязка реализована как **тонкий wrapper**, который:

- монтирует ABGrid в DOM
- управляет жизненным циклом (init / destroy)
- позволяет обновлять конфигурацию
- предоставляет доступ к экземпляру грида

Важно:
ABGrid Engine остаётся **самостоятельным stateful-движком**, а React используется только как слой интеграции.

24.2 Установка

```
npm install abgrid-engine
```

24.3 Импорт

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

24.4 Базовое использование

```
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 Получение экземпляра грида

```
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 Обновление данных

Если передаётся `data`, компонент автоматически обновляет данные грида:

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

24.7 Жизненный цикл

React-адаптер автоматически:

- создаёт grid при mount
- уничтожает (destroy) при unmount
- пересоздаёт grid при изменении config

24.8 События

События можно обрабатывать через props:

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

(Поддерживаемые события соответствуют событиям ABGrid Engine)

24.9 Важные замечания

- Не используйте React state для управления каждой ячейкой — ABGrid уже управляет состоянием
- Используйте ref для вызова методов грида
- Избегайте частого пересоздания config (желательно мемоизировать)

25. Интеграция с Vue

25.1 Обзор

ABGrid Engine предоставляет официальный адаптер для Vue-приложений.

Vue-обвязка:

- монтирует grid в компонент
- управляет жизненным циклом
- отслеживает изменения конфигурации
- предоставляет доступ к экземпляру грида

25.2 Установка

```
npm install abgrid-engine
```

25.3 Импорт

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

25.4 Базовое использование

```
<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 Получение экземпляра грида

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

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

25.6 Обновление данных

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

25.7 Жизненный цикл

Vue-адаптер автоматически:

- создаёт grid при `mounted`
- уничтожает при `beforeUnmount`
- пересоздаёт при изменении `config`

25.8 События

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

25.9 Важные замечания

- ABGrid управляет своим состоянием, Vue — только оболочка
- не нужно пытаться полностью «реактивить» grid
- используйте события и API экземпляра

26 Общие рекомендации

26.1 Архитектурный принцип

React и Vue адаптеры являются:

тонкими обёртками над ABGrid Engine

Они:

- не дублируют бизнес-логику
- не заменяют DataEngine
- не вмешиваются в внутреннее состояние

26.2 Когда использовать адаптеры

Используйте React/Vue адаптеры, если:

- проект уже построен на React или Vue
- требуется быстрая интеграция без ручного DOM-кода

26.3 Когда использовать vanilla ABGrid

Используйте напрямую ABGrid Engine, если:

- нужен полный контроль
- нет зависимости от фреймворков
- требуется максимальная производительность

27. Цветовая палитра

ABGrid Engine поставляется с 7 профессиональными цветовыми темами. Пользователи могут на их основе разработать и подключать свои. Все цветовое оформление расположено в папке themes. Там же находятся глобальные CSS-файлы для всей html-страницы, гармонично сочетающиеся с цветовым оформлением компонента.

28. Комплект поставки

Поставка включает в себя все необходимые файлы и не требует никаких внешних библиотек. ABGrid Engine распространяется в нескольких форматах:

- ESM — для современных проектов и сборщиков,
- UMD — для универсального подключения,
- IIFE (abgrid.min.js) — для прямого подключения через <script> без сборщиков.

В корневой папке так же находятся файлы логотипа, файл иконок CSS.
В папке themes 7 профессиональных тем `abgrid-theme-*.css` и 7 гармоничных тем для страниц в целом `page-skin-*.css`

Ниже перечислены структура и файлы поставки.

1) Файлы компонента:

```
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) Руководство разработчика (этот документ)