# Wasaby Templates

## Реактивные свойства

**Функционал реактивности на свойства** для прикладного разработчика означает, что изменение свойств контрола будет вести к его перерисовке. Свойство контрола становится реактивным, когда используется в шаблоне. Если значение свойства изменяется, Wasaby автоматически запрашивает его перерисовку в шаблоне.

Для свойств с типом Object перерисовка происходит, когда выполнено одно из следующих условий:

* свойство меняется по ссылке
* является экземпляром наследника класса модели Types/entity:Model и модель изменяется.
* является экземпляром наследника класса рекорда Types/entity:Record и рекорд изменяется.
* является экземпляром наследника класса реактивного объекта Types/entity:applied.ReactiveObject и объект изменяется.

Для свойств с типом Array к автоматической перерисовке приводят изменения, вызванные методами push, pop, shift, unshift, splice, sort и reverse или смена ссылки. Если удалять из массива через ~~delete~~ или менять значение через индекс – перерисовки не будет.

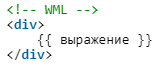
Однако, если массив передается опцией в дочерний контрол, то при изменении массива нативными методами дочерний контрол перерисовываться не будет, т.к. нужна новая ссылка в этом случае. Для решения заявленной проблемы необходимо создать копию массива.

## Синтаксис шаблонов

### Mustache

Шаблонизатор Wasaby распознаёт конструкции, созданные с помощью {{}}.

Выражение этих конструкций можно составлять из переменных, находящихся в области видимости шаблона, а также из синтаксических конструкций языка JavaScript.

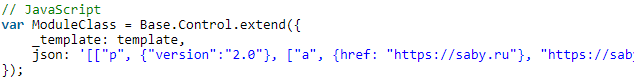


При обработке шаблона вычисляется выражение конструкции, его значение экранируется и, как следствие, выводится на странице как простой текст (не HTML)

### Вывод JsonML

Значение конструкции можно вывести как HTML. Для этого необходимо отрендерить Controls/decorator:Markup с опцией value, которая принимает данные формата JsonML (перед вставкой выполняет санитизацию небезопасных тегов):





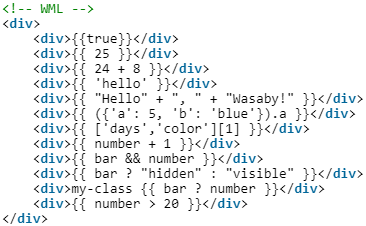
### Спецсимволы HTML

Для отображения спецсимволов html рекомендуется либо вставлять в шаблон сам символ, либо его числовой код. В следующем примере в обоих случаях в вёрстку страницы выводится символ рубля:

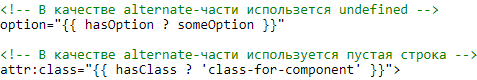
будет преобразовано в: 

### Использование JavaScript в шаблоне

Шаблон поддерживают работу с выражениями языка JavaScript: арифметические операции, операторы сравнения и условные операторы, использование переменных из области видимости шаблона:



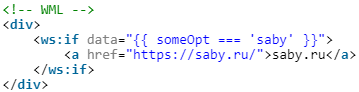
В шаблонах Wasaby Framework тернарный оператор может использовать без alternate-части:



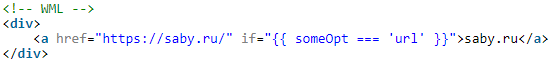
### Условия

**Директива ws:if**

Директива <ws:if> позволяет добавлять html-разметку при выполнении условия, описанного в атрибуте data. Значением атрибута является конструкция шаблонизатора. Когда атрибут принимает значение true, в DOM вставляется содержимое блока <ws:if>:



Короткая нотация:



### Директива ws:else

Директиву <ws:if> можно дополнить блоками <ws:else>. Для всех блоков, кроме последнего, допускается создание условия в атрибуте data:



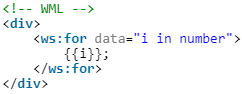
Между блоками <ws:if> и <ws:else> запрещается встраивание разметки!

### Циклы

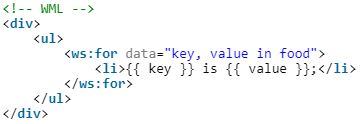
**Директива ws:for**

В атрибуте data допускается организовывать итерации по целочисленному диапазону, объектам, массивам и экземплярам классов Types/entity:Model или Types/entity:Record. Для каждого типа данных используется собственная форма записи.

* Итерация по целочисленному диапазону или элементам массива:



* Итерация по свойствам объекта или элементам массива:



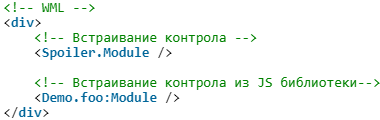
Директива создаёт новую область видимости переменных. Переменные видны только внутри оператора, но не за его пределами.

### Встраивание контрола или внешнего шаблона

**По имени**

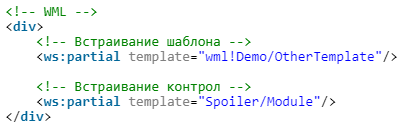
Встроенный таким образом контрол будет автоматически загружен Wasaby, поэтому нет необходимости указывать его в массиве зависимостей модуля.

«/» - заменяют на символ точки



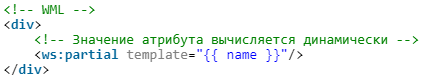
### Директива ws:partial

Используется для добавления в шаблон контрола или другого шаблона. В атрибут template передают имя контрола или шаблона:



Так же нет необходимости указывать добавленный таким образом контрол в массиве зависимостей модуля.

Значение атрибута можно вычислять динамически с помощью синтаксических конструкций. Однако в этом случае не происходит автоматической загрузки зависимости:

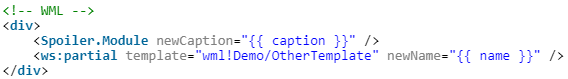


### Передача опций

Добавленный контрол/шаблон создаёт изолированную область видимости, в которой недоступны переменные родительского шаблона. Но эти переменные можно передать в опциях.

В шаблоне опции доступны в переменной \_options, а в JS-модуле — в переменной this.\_options.

Пример: в области видимости родительского шаблона существует переменная caption и name. С помощью опций newCaption и newName в мы передали значения переменных:



**Синтаксис XML для опций**

Общая форма:



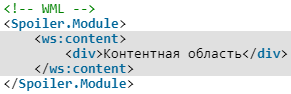
Пример:

<https://codesandbox.io/embed/configuration-options-template-syntax-x5bz4?fontsize=14&hidenavigation=1&theme=dark>

**Контентная опция**

Предназначена для шаблона, который можно задать по месту при встраивании контрола. Если название контентной опции в контроле не задано, то по умолчанию используется название "content".

Пример: Встраиваение шаблона в контентную опцию контрола Spoiler.Module:



Пример 2:



Шаблон контрола ListView:



Контентные опции используются в компонентах для вставки заданного шаблона через конструкцию ws:partial

### Передача атрибутов

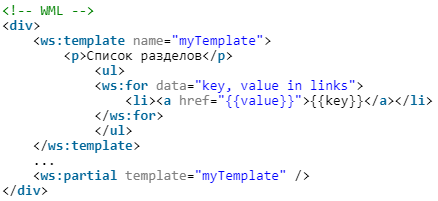
Для встроенного контрола/шаблона используется для обозначения атрибутов префикс attr. Описанные таким образом атрибуты будут добавлены на корневой DOM-элемент:



### Директива <ws:template>. Встроенный шаблон

С помощью этой директивы вы можете объявить встроенный шаблон - фрагмент вёрстки, предназначенный для локального использования.

* Шаблон объявляют до момента его использования
* В атрибуте name задают имя шаблона
* Для добавления встроенного шаблона используют директиву <ws:partial>

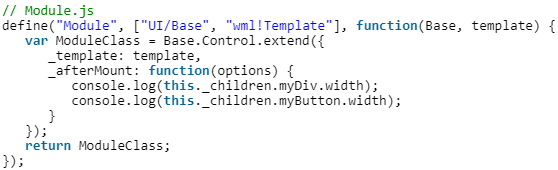


В таком шаблоне опции доступны по прямому названию аттрибута, без options

## Доступ к элементам вёрстки и дочерним контролам

В Wasaby изменение состояния дочерних html-элементов и контролов следует выполнять через обновление свойств родительского контрола (через опции). Но если потребовалось манипулировать дочерними элементами напрямую, используйте объект this.\_children, содержащий дочерние элементы (html-элементы, контролы), которым задан атрибут name:





Обращение к дочерним контролам возможно на всех фазах кроме хука \_beforeMount.

Запрещено обращаться к дочерним контролам дочерних контролов, поскольку это нарушает принцип инкапсуляции.

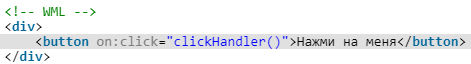
## Работа с событиями

Для подписки на событие используют директиву **on**. Можно подписаться на события элементов шаблона:

* DOM-элемент;
* контрол;
* шаблон, встроенный директивой ws:partial.

Директива on имеет следующую форму записи: on:<событие>="<обработчик>(<параметры>)"

### Подписка на нативное событие DOM-элемента:



JS:



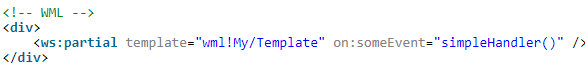
### Подписка на событие контрола:



JS:

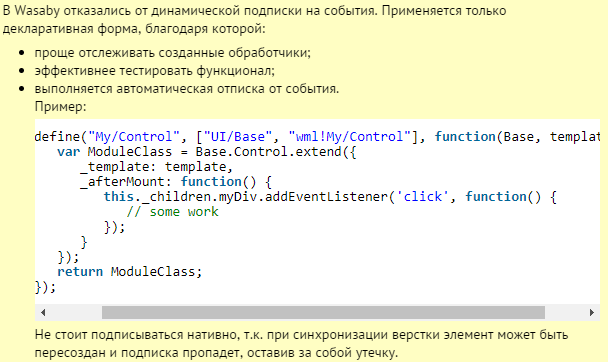


### Подписка на событие шаблона:



JS:





### Передача параметров события из шаблона



TS:



**е** – это дескриптер события (экземпляр класса Vdom/Vdom:SyntheticEvent), он всегда приходит первым аргументом. Он содержит свойства:

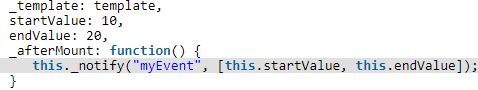
* *nativeEvent* — содержимое этого свойства зависит от типа события. Для нативного события содержит поля интерфейса Event. Принимает null, если событие публикуется контролом.
* *result* — значение, возвращенное из предыдущего обработчика, который выполнен на то же событие.
* *stopped* — признак, что не происходит дальнейшее всплытие события по цепочке.
* *tar*g*et* — DOM-элемент, в котором произошло событие.
* *type* — имя события.

### Публикация пользовательского события

Для публикации события в контроле вызывают метод \_notify(). Первый аргумент метода — обязательный, он принимает имя события.

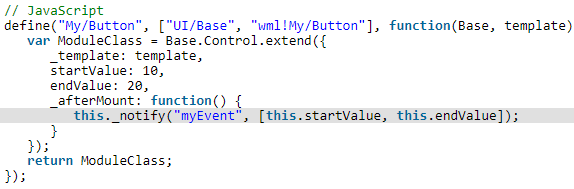


Во втором аргументе метода \_notify() можно передать массив параметров. Они будут доступны в аргументах обработчика события.

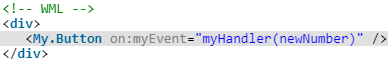


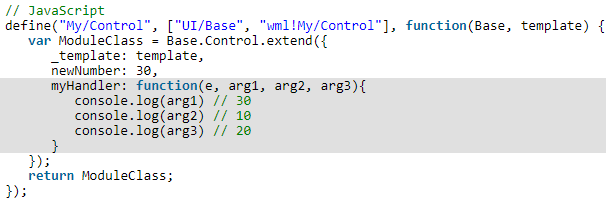
### Приоритет обработки параметров события

В аргументах обработчика события сначала доступны параметры из шаблона, а далее — параметры из модуля, переданные в момент публикации.



Control.wml:

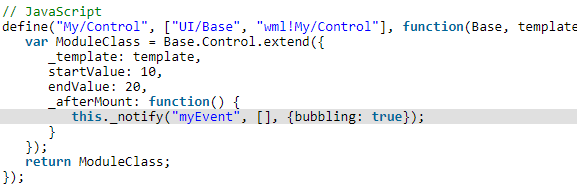




### Всплываемость

При публикации события его можно сделать всплываемым. Тогда оно будет передаваться из контрола (источника события) вверх по всей цепочке родительских **элементов шаблона**. Данное поведение можно использовать для перехвата события и дальнейшей обработки.

Для этого следует в методе \_notify() передать третьим аргументом значение {bubbling: true}.



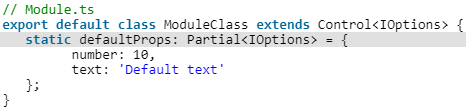
## Свойства и опции

**Опции** — это объект, в котором хранятся параметры, переданные при инициализации контрола. В шаблоне опции доступны в переменной \_options, а в модуле — в переменной this.\_options.

Опции доступны только на чтение.

**Начальные значения опций**

Для указания начального значения опций определите в контроле статическое поле defaultProps. Данное поле устанавливает значение опции по-умолчанию, если эта опция не задана:



## Синхронизация опций

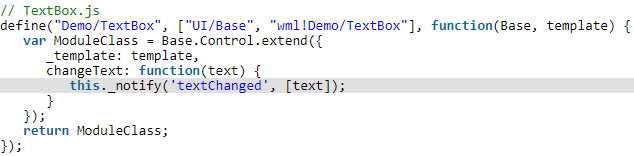
### bind

Wasaby по умолчанию синхронизирует изменения опций в одном направлении — из родительского контрола к дочерними. Для создания двусторонней синхронизации используют директиву bind.



То есть у контрола **TextBox** есть опция **text**, и при ее изменении будет соответственно меняться свойство **myText** у родительского компонента **Module.**

При этом из **TextBox**, должно быть опубликовано событие на изменение значения свойства. Имя публикуемого события должно соответствовать формату **<свойство дочернего контрола>Changed:**



Полная форма записи директивы такова:

bind:<свойство дочернего контрола>="свойство родительского контрола"

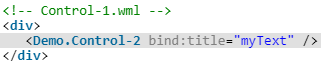
### Зачем нужно событие <свойство дочернего контрола>Changed?

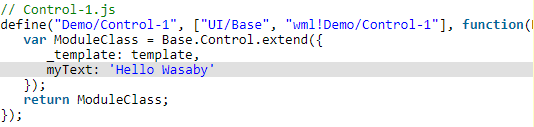
Событие "<свойство дочернего контрола>Changed" может быть предназначено для bind, который объявлен в контроле, который располагается выше родительского контрола. В этом случае нужно выполнить одно из следующих условий:

1. Отправлять событие с параметром bubbling: true
2. Проксировать событие у родительского контрола. Примечание: для проксирования не нужно объявлять bind у родительского контрола. Событие достаточно поймать, а затем — опубликовать такое же. Ниже показан пример как это реализуется для трёх контролов:

* Control 1:

У контрола Control 1 есть свойство myText, которое он передает в опцию title контрола Control 2 и биндит на изменение.





* Control 2:

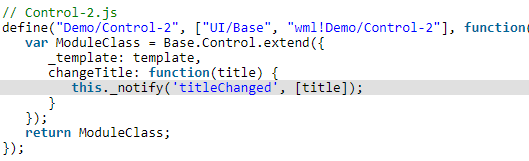
Контрол2 прокидывает в опцию title Control 3 свою опцию title и подписывается на события on:titleChanged Контрола 3, которое будет срабатывать при изменении в нем опции title

<!-- Control-2.wml -->

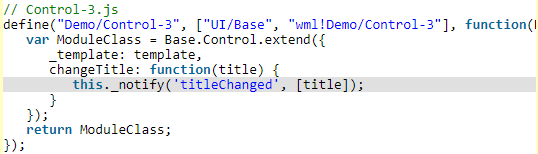
**<div>**

**<Demo.Control-3 title="{{ \_options.title }}" on:titleChanged="changeTitle()" />**

**</div>**



* Control 3:



## Асинхронное построение контролов

Про recivedState

## Работа с фокусами

Система фокусов в Wasaby позволяет пользоваться нативной системой фокусов (нативными событиями фокусов, нативным методом HTML-элемента focus), а также расширяет и дополняет её.

### Определение фокусируемости элемента

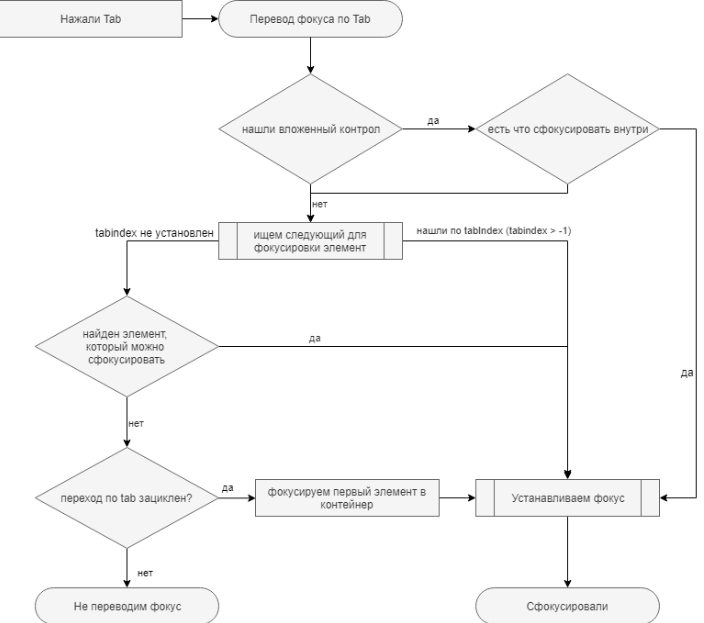
Элементы при обходе по Tab фокусируются согласно стандарту HTML. Как и в нативном js, фокусироваться по умолчанию будут элементы, реагирующие на нажатия клавиш (у них tabindex=0). Например:

* <button> — выполнение действия кнопки по Enter.
* <input> — ввод с клавиатуры.
* <checkbox> — выставление флажка по пробелу.
* <a> — переход по ссылке по Enter.

Также будут сфокусированы элементы с настроенным табиндексом, поскольку tabindex >= 0:



### Принцип обхода по Tab



То есть в обходе по Tab содержимого шаблона участвуют как элементы, так и вставленные в шаблон контролы. Когда в процессе обхода попадается контрол, шаблон этого контрола рекурсивно обходится по такому же принципу. Когда обход контрола завершен, продолжается обход текущего шаблона.

### Настройка табиндексов в шаблонах

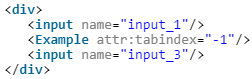
Порядок обхода с учетом табиндексов в пределах шаблона соответствует стандарту HTML, то есть от меньшего табиндекса к большему.

В основном tabindex нужно указывать, чтобы задать порядок обхода (tabindex >= 0), либо чтобы исключить элемент/контрол из обхода (tabindex == -1):

Шаблон контрола Example:



Шаблон контрола Container:



Обход по Tab в данном примере: input\_1, input\_3.



### Изолирование табиндексов внутри контроллов

Контекстная зависимость табиндексов позволяет задавать табиндексы в пределах шаблона не боясь, что их значения пересекаются со значениями других шаблонов:

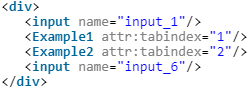
Шаблон контрола Example1:



Шаблон контрола Example2:



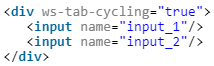
Шаблон контрола Container:



Example1 и Example2 имеют изолированную настройку табиндексов в своих шаблонах. При обходе по Tab сначала будут пройдены все элементы шаблона Example1, а затем все элементы шаблона Example2.

### Зацикливание обхода по Tab

Чтобы добиться зацикливания обхода по Tab в пределах элемента, достаточно воспользоваться атрибутом ws-tab-cycling:



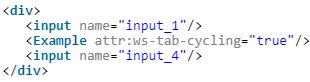
Обход по Tab: input\_1, input\_2, input\_1, input\_2, input\_1, input\_2 и т.д.

Таким же образом можно настроить контрол по месту использования:

Шаблон контрола Example:



Шаблон контрола Container:



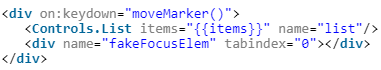
Обход по Tab: input\_1, input\_2, input\_3, input\_2, input\_3, input\_2, input\_3 и т.д.

### Настройка контролов, реагирующих на нажатия клавиш

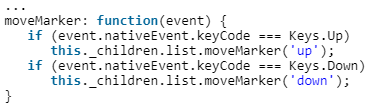
Если в шаблоне контрола нечего фокусировать, обход по Tab проигнорирует контрол. Необходимо обеспечить, чтобы внутри контрола что-то могло сфокусироваться, чтобы контролом можно было управлять с клавиатуры.

Пример:

Шаблон контрола Controls.ImprovedList — списка, между строками которого можно перемещаться клавишами стрелок вверх и вниз:



Контроллер Controls.ImprovedList:



При обходе по Tab в контроле Controls.ImprovedList будет найден и сфокусирован элемент fakeFocusElem. Нажатия клавиш начнут перехватываться обработчиком moveMarker.

### Запуск активации контрола

Физический родитель контрола

**Физический родитель контрола** - определятся использованием одного контрола в другом. Пример:

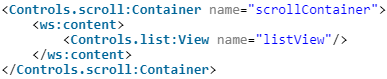
Шаблон контрола Example:



Физическим родителем контрола Add будет контрол Example.

Пример 2: объявление в одном контроле, а использование в другом:

Шаблон контрола Example:



Шаблон контрола Controls/scroll:Container:



Controls.list:View объявляется в контроле Example, но используется в Controls.scroll:Container.

В таком случае:

* Физическим родителем Controls/list:View будет Controls/scroll:Container (т.к. физический родитель определяется по использованию).
* Физическим родителем Controls/scroll:Container будет Example.

Опенер контрола

**Определение понятия "опенер контрола" (opener) -** Контрол\_1 является опенером по отношению к Контролу\_2, если для Контрола\_2 опция opener равняется Контролу\_1.

Опция opener задает логическую связь между открываемым дочерним контролом и родительским контролом, который является причиной открытия дочернего. Без этой связи невозможно доподлинно определить интересующую нас логическую связь между областями.

Таким образом, опенеры корректируют структуру контролов так, что она перестает учитывать только DOM-дерево, но также начинает учитывать логическую связь между контролами.

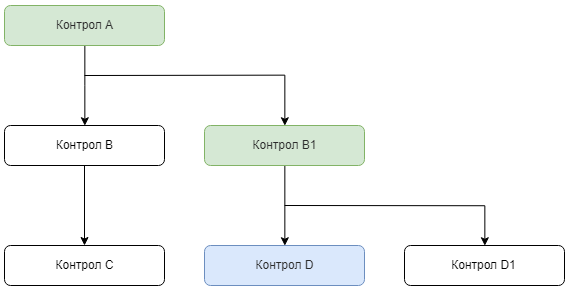


"Активность" контрола

Контрол активен, если выполняется одно из условий:

1. Один из элементов в его шаблоне — в фокусе.
2. Он является физическим родителем контрола, который активен и у которого нет опенера.
3. Он является опенером контрола, который активен.

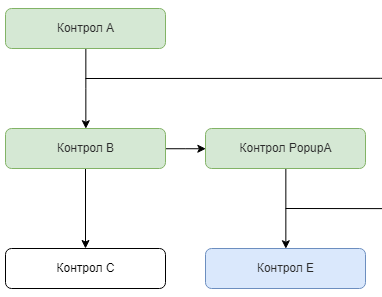
Пример 1:



Активными будут считаться "Контрол А", "Контрол B1" и "Контрол D", т.к. выполняются 1 и 2 условия активности:

* "Контрол D" активен т.к. в его шаблон установлен фокус. (условие 1)
* "Контрол B1" активен т.к. является физическим родителем для "Контрол D". (условие 2)
* "Контрол А" активен т.к. является физическим родителем для "Контрол B1". (условие 2)

Пример 2:



* "Контрол E" активен т.к. в его шаблон установлен фокус. (условие 1)
* "Контрол Popup" активен т.к. является физическим родителем для "Контрол E". (условие 2)
* "Контрол B" активен т.к. является опенером для "Контрол Popup". (условие 3)
* "Контрол А" активен т.к. является физическим родителем для "Контрол B". (условие 2)

Метод wasaby-контрола activate для запуска активации

Метод activate предоставляет высокоуровневый способ найти и сфокусировать элемент внутри выбранного контрола. Метод является частью API базового контрола UI/Base:Control.



API метода activate: control.activate(config), где где config — необязательный аргумент. Это объект, который может содержать параметры:

* enableScreenKeyboard — в случае значения true, на мобильных устройствах разрешается фокусировка полей ввода и соответствующее отображение экранной клавиатуры. в случае значения false, на мобильных устройствах запрещается фокусировка полей ввода (фокусироваться будет контейнер). По умолчанию значение равно false.
* enableScrollToElement — в случае значения true, если фокусируемый элемент находится за пределами видимой области, после фокусировки происходит подскролл к элементу, чтобы он стал видимым. в случае значения false, подскролл к элементу отключен. По умолчанию значение равно false.

Так как enableScreenKeyboard по умолчанию false, при автофокусировке на мобильных устройствах поля ввода фокусироваться не будут и экранная клавиатура не покажется. Подскролла к фокусируемому элементу так же не будет, так что при перезагрузке страницы состояние скролла страницы сохранится.

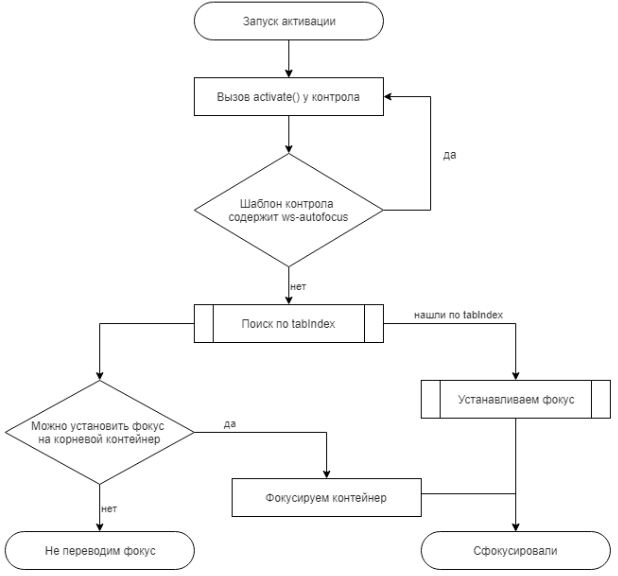
Автофокусировка отобразившейся области

В рамках системы фокусов метод wasaby-контрола activate зовется только для запуска автофокусировки.

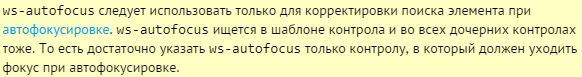
**Авто**фокусировка запускается:

* При загрузке страницы — вызов activate для корневого контрола страницы.
* При переходе на страницу в SPA-режиме — вызов activate для корневого контрола страницы.
* При открытии попапа — вызов activate для попапа.

Алгоритм активации



1. Вызов **activate()** у контрола.
2. Рекурсивно ищем директиву **ws-autofocus** в шаблоне контрола.
3. Если нашли — вызываем **activate()** у найденного контрола.
4. Если не нашли — ищем элемент с учетом контекстной зависимости tabindex.
   * Нашли элемент — фокусируем (в таком случае считаем активацию успешной).
5. Не нашли — пытаемся фокусировать корневой элемент области, для который вызвали activate().
   * Корневой элемент можно сфокусировать — фокусируем (в таком случае считаем активацию успешной).
   * Корневой элемент нельзя сфокусировать — фокус остается на месте (в таком случае считаем активацию не успешной).



Пример в конце статьи: <https://wi.sbis.ru/doc/platform/developmentapl/interface-development/ui-library/focus/activate-control/>

### События активности

Чтобы можно было отреагировать на изменение активности контрола, предоставляется два события:

* activated (контрол активирован)
* deactivated (контрол деактивирован)

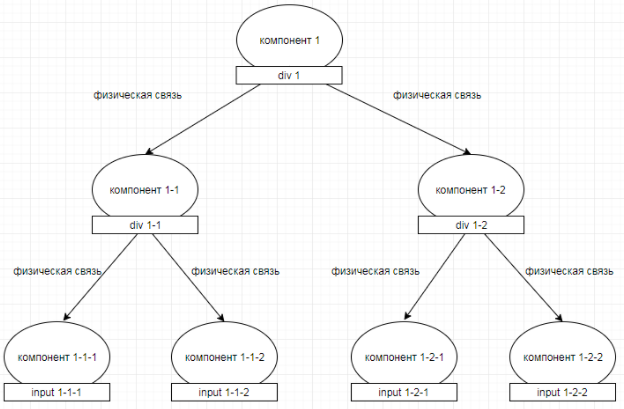
Подробнее о событиях: <https://wi.sbis.ru/doc/platform/developmentapl/interface-development/ui-library/focus/events-activity/#activated-event>



Вычисление активности контролов вслед за изменением фокуса

Пересчет состояний активности базируется на положении фокус. Изменение фокуса влечет за собой анализ состояний активности контролов и нотификацию событий активности. При этом на контролах, которые не теряли активность, не будет вызвано событий активности.

Пример:



При переводе фокуса с 1-1-1 на 1-2-1 сначала сработают нативные события, а потом события активности.

* Событие потери фокуса focusout сработает на элементах:

*input 1-1-1, div 1-1, div 1*

* Событие получения фокуса focusin сработает на элементах:

*input 1-2-1, div 1-2, div 1*

* Событие deactivated сработает:

*на контроле 1-1-1, контроле 1-1*

* Событие activated сработает:

*на контроле 1-2-1, контроле 1-2*

### Особенности фокусировки по клику (ws-no-focus)

Клик по области фокусирует область. Таким образом можно фокусировать области кликом и начинать управление с клавиатуры по месту клика.

Атрибут ws-no-focus не дает клику перевести фокус на элемент, оставляя фокус там, где он находится. При этом обработчик клика выполняется.

Так как элемент при клике не будет сфокусирован, соответственно события activated и deactivated не будут вызваны. Управление с клавиатуры будет осуществляться в контексте элемента, который сфокусирован.

### Восстановление фокуса

Потеря фокуса происходит, если сфокусированный элемент неожиданно скрывается или удаляется из DOM. В таком случае фокус перемещается в body и возможность управления с клавиатуры теряется. Чтобы предотвратить это, реализовано восстановление фокуса.

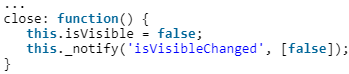
По умолчанию фокус восстанавливается ближайшему родителю, способному принять фокус. Поиск контрола осуществляется среди **активных** контролов, то есть с учетом опенеров. У найденного контрола фокусируется корневой элемент.

Пример:

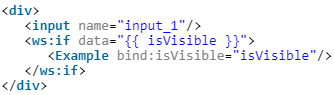
Шаблон контрола Example:



Контроллер Example:



Шаблон контрола Container:



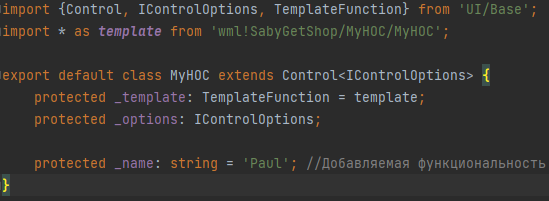
Если фокус находится на input\_2 и пользователь кликает на кнопку закрытия, контрол Example пропадает. Механизм восстановления фокуса переводит фокуса на корневой элемент контрола Container — <div>. При этом события activated и deactivated не срабатывают. Таким образом контекст фокуса сохранился и можно продолжать управление с клавиатуры.

Если такой способ восстановления не устраивает — программист, чей контрол своими действиями повлек потерю фокуса, может восстановить фокус самостоятельно туда, куда ему нужно. Для этого он может воспользоваться методом wasaby-контрола activate или нативным методом HTML-элемента focus.

## Практические notes

### Пример реализации HOC

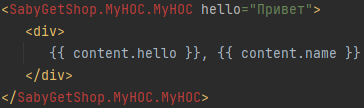
* MyHOC.ts:



* MyHOC.wml:



* Использование MyHOC:



В HOC передали в качестве опции строку «Привет». HOC в свою очередь передает все свои опции в тот контент, который оборачивает, а так же добавляет в опции контента еще и свою переменную.

Результат:



Итого: в оборачиваемый компонент прозрачно переданы все опции + добавленная функциональность. В partial может быть отрендерен передаваемый компонент из контентной опции плюс своя верстка с помощью директивы <ws:template>.

# TypeScript

## TypeScript basics

<http://typescript-lang.ru/docs/>

## Жизненный цикл

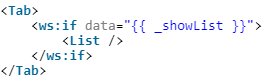
Есть три основных фазы жизненного цикла: создание, обновление и уничтожение

### Фаза «Создание»

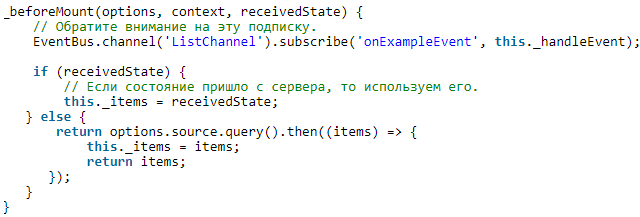
|  |  |  |
| --- | --- | --- |
| **Сигнатура хука** | **Предназначение** | **Момент вызова** |
| \_beforeMount(options, context, receivedState) | Служит для инициализации состояния по полученным опциям.  Избегайте побочных эффектов в этом методе, потому что они могут привести к утечкам памяти. Подробнее читайте в статье: [Опасность побочных эффектов в \_beforeMount](https://wi.sbis.ru/doc/platform/developmentapl/interface-development/ui-library/control/#danger-of-before-mount-side-effects).  Если из этого хука вернуть Promise, то построение этого контрола отложится до его разрешения. Результат Promise, выполненного на сервере, придёт третьим аргументом при построении на клиенте. Более подробно смотрите в [примере](https://wi.sbis.ru/doc/platform/developmentapl/interface-development/ui-library/control/#async-build). | Вызывается до построения вёрстки контрола. Соответственно, здесь ещё нет DOM и детей.  **Единственный хук, который вызывается и на сервере, и на клиенте**. |
| \_componentDidMount(options) | Служит для манипуляций с DOM **перед тем, как пользователь увидит кадр.** Например, для корректирования положения скролла.  **Хук вызывается синхронно**, поэтому может вызвать проблемы с производительность. Поэтому предпочтительнее использовать \_afterMount(), чтобы не откладывать отрисовку кадра | Вызывается синхронно после того, как изменения были **впервые** применены к DOM **до того, как браузер отрисовал кадр.** |
| \_afterMount(options) | **Служит для инициализации состояния, которое зависит от DOM, а также подписок на события.**  Не стоит без необходимости изменять реактивные свойства в данном методе, т.к. это приведёт к перерисовкам. | **Вызывается после построения вёрстки и отрисовки кадра браузером.**  Вызывается один раз после построения на клиенте. |

**Опасность побочных эффектов в beforeMount:**

Пример: Есть шаблон:



\_beforeMount у List выглядит вот так:



За то время, пока List ходит за данными (ждет разрешения промиса), состояние приложения изменяется, и Tab умирает. В то же время, у List \_beforeUnmount не вызовется, потому что не успел вмонтироваться. В следствии, в \_beforeUnmount не произойдет отписка.

*Чтобы избежать этой проблемы, все подписки нужно делать в \_afterMount()*

**Загрузка данных дважды: на сервере и на клиенте**

Если пришел recivedState, то промис возвращать не надо!

### Фаза "Обновление"

|  |  |  |
| --- | --- | --- |
| **Сигнатура хука** | **Предназначение** | **Момент вызова** |
| \_beforeUpdate(newOptions) | **Служит для изменения состояния в ответ на изменение опций**. Старые опции доступны на this.\_options, новые приходят первым аргументом.  **Не забывайте сравнивать старые и новые опции перед изменением состояния.** Более подробно о том, к каким проблемам это приведёт, читайте в статье: [Контрол безусловно меняет состояние в \_beforeUpdate](https://wi.sbis.ru/doc/platform/application-debugging/js/debugging-common-errors/change-state/). | Вызывается перед тем, как контрол обновится с новыми опциями. |
| \_afterRender() | **Служит для манипуляций с DOM перед тем, как пользователь увидит кадр.** Например, для корректирования положения скролла.  Использование данного хука обычно приводит к проблемам с производительностью, например, к [forced reflow](https://wi.sbis.ru/doc/platform/application-debugging/js/debugging-common-errors/forced-reflow/). Поэтому для большинства операций предпочтительнее использовать \_afterUpdate(), чтобы не откладывать отрисовку кадра. | Вызывается синхронно после того, как изменения были применены к DOM до того, как браузер отрисовал кадр. |
| \_afterUpdate(oldOptions) | Служит для изменения состояния после обновления. Здесь можно взаимодействовать с DOM, например, измерять размеры элементов.  **Чтобы избежать зацикливания синхронизаций, меняйте состояние только при наличии изменений.** | Вызывается после обновления контрола и, в отличие от \_afterRender(), уже после того, как кадр был отрисован.  Этот метод вызывается **только** после обновлений. Если вам нужно что-то сделать после построения, то используйте \_afterMount(). |

### Фаза "Уничтожение"

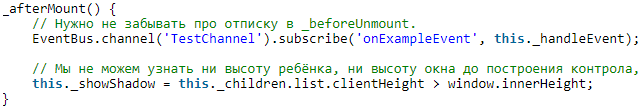
|  |  |  |
| --- | --- | --- |
| **Сигнатура хука** | **Предназначение** | **Момент вызова** |
| \_beforeUnmount() | **Служит для очистки состояния контрола, отписок от событий и т.д.** После вызова этого метода любое взаимодействие с контролом (обращения к свойствам, вызовы методов и т.д.) является ошибкой.  В этом методе не нужно зачищать поля контрола, т.к. Wasaby делает это автоматически. | Вызывается перед уничтожением контрола. |

### Примеры использования хуков

* **\_beforeMount**

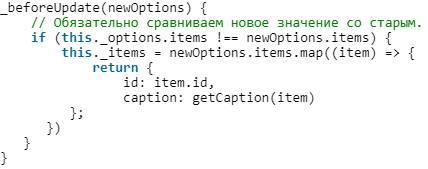
Понятно

* **\_afterMount**



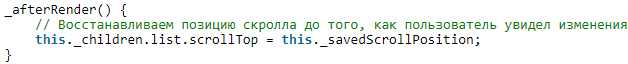
В этом методе DOM уже вмонтирован и показан и мы можем на основании высоты элементов высчитывать значение тени.

* **\_beforeUpdate**

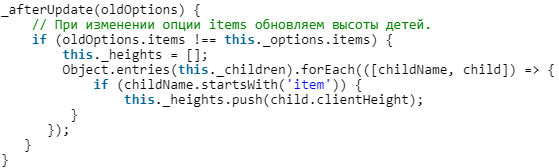


Если пришли новые значения опций – то на их основании перерасчитаем реактивные свойства

* **\_afterRender**



* **\_afterUpdate**



* **\_beforeUnmount**



## Типы, структуры данных и коллекции

Первичный ключ - <https://otvet.mail.ru/question/210740205>

### MVC

Приложение в веб-фреймворке Wasaby строится по концепции MVC.

В срезе визуальных компонентов (контролов) принимается, что:

* моделью является или абстрактная модель (прим. класс Model), или коллекция (прим. класс RecordSet). Для привязки значений используется механизм binding-ов;
* представлением являются контролы;
* контроллерами являются прикладные модули.

Схема работы приложения:

* Контроллер манипулирует моделью, внося изменения в ее состояние:
  + в случае абстрактной модели — меняет значения ее полей;
  + в случае коллекции — добавляет/удаляет/обновляет элементы коллекции;
* представление получает от модели уведомления об изменении состояния и актуализирует внешний вид согласно изменениям.

Работа с данными в контролах коллекций

Для обеспечения возможности использовать веб-фреймворк с любым back-end'ом, выделен интерфейс источника данных.

Поддержка различных форматов кодирования данных осуществляется через набор адаптеров, которые могут внедряться в модели и источники данных.

Чтобы подключить новый back-end, нужно лишь реализовать источник данных, работающий по его протоколу и с его форматом данных.

Источники данных используются контролами для решения следующих задач:

* получение списка записей (выборки) для отображения;
* отправка измененных записей обратно в источник (синхронизация операций изменения, добавления, удаления).

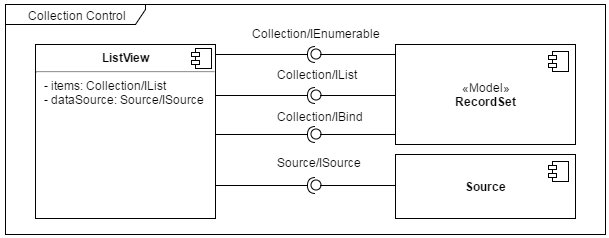
Работа с данными является важным аспектом жизненного цикла компонента. Пользователю доступны два сценария работы:

* Данные будут внедрены в контрол явно (в виде экземпляра коллекции прим. RecordSet);
* Данные будут получены самим контролом через специализированный объект — источник данных.

Возможны и смешанные сценарии: можно установить отображаемую коллекцию явно, а все изменения в ней синхронизировать с источником данных.

Для реализации этих сценариев контрол агрегирует два объекта:

* items — коллекция для отображения;
* dataSource — источник данных.



### Базовые сущности. Model, Record

Types/entity:Record

Запись — обертка над данными, которые можно представить в виде строки таблицы (упорядоченный набор именованных полей и их значений).

Возможности:

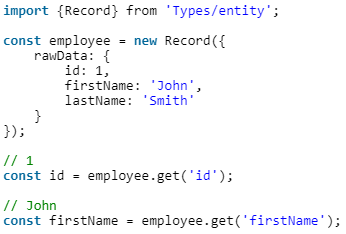
* использование сырых данных в разных форматах;
* чтение/запись значения поля по имени;
* проверка существования поля;
* последовательный перебор полей;
* манипуляции с форматом полей;
* сериализация/десериализация;
* поддержка шаблона проектирования "Издатель-подписчик".

*По умолчанию используется адаптер для данных в формате JSON*. Если вы используете данные в каком-либо другом формате (СБИС-JSON, XML), то нужно подключить соответствующий адаптер.

Про адаптеры: <https://wi.sbis.ru/doc/platform/developmentapl/interface-development/data-sources/abstraction-layers/>

Назначение адаптера - предоставить общий интерфейс для работы различными форматами данных.

Пример: rawData – «сырые данные», пришедшие с БЛ (тут JSON):



Пример с сырыми данными СБИС-JSON:



Types/entity:Model

Модели обеспечивают доступ к данным и поведению объектов предметной области (сущностям). Такими сущностями могут быть, например, товары, пользователи, документы — и другие, которые моделируются в приложении.

Возможности:

* обладает всеми возможностями Record.

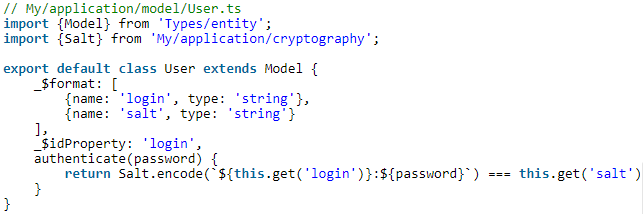
Дополнительные возможности:

* определение собственных свойств сущности (пример: поле "Фамилия и инициалы" сущности "Пользователь" рассчитывается по полям "Фамилия", "Имя", "Отчество");
* наличие уникального идентификатора сущности среди ей подобных.

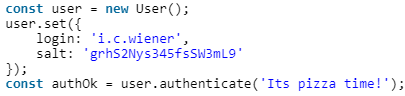
Поведенческие аспекты каждой сущности реализуются ее прикладным модулем путем наследования от абстрактной/промежуточной модели.

Прикладные модели могут внедряться в порождающие их объекты, такие как источники данных или рекордсеты. Это позволяет использовать свои модели в контролах.

Пример: создадим модель пользователя:



И создадим экземпляр модели пользователя и вызовем его метод:



### Коллекции. ICollection

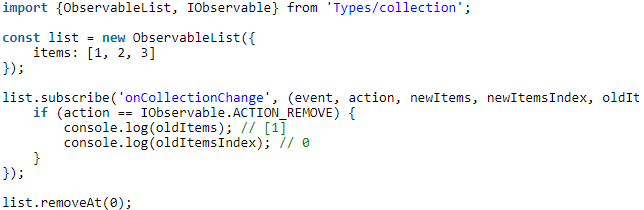
Полное описание всех интерфейсов и их реализаций: <https://wi.sbis.ru/doc/platform/developmentapl/interface-development/models-collections-types/icollection/>

Интерфейс Types/collection:IList и его реализация Types/collection:ObservableList

Возможности интерфейса:

* чтение/запись элемента коллекции по индексу;
* добавление/удаление элементов коллекции по индексу;
* поиск индекса элемента;
* очистка списка, замена содержимого из другой коллекции;
* получение количества элементов.

Types/collection:ObservableList – список, в котором можно отслеживать изменения:



Types/collection:RecordSet

Рекордсет — список записей, имеющих общий формат полей.

Возможности:

* Рекордсет обладает всеми возможностями ObservableList.

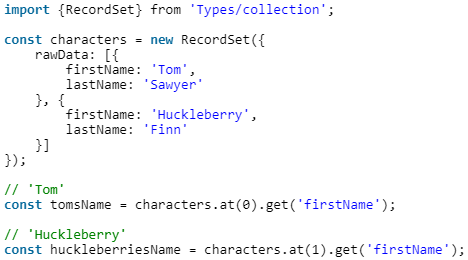
Дополнительные возможности:

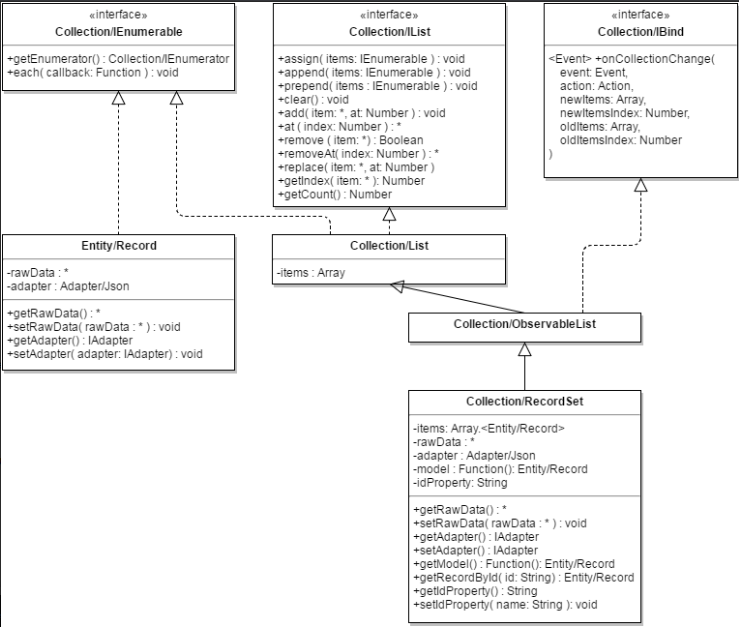
* использование сырых данных в разных форматах;
* генерация записей по сырым данным;
* внедрение собственных конструкторов записей;
* манипуляции с форматом полей;
* сериализация/десериализация;

По умолчанию используется адаптер для данных в формате JSON.

Пример:

Рекордсет, в котором в качестве сырых данных используется JSON:





## Источники данных

### Интерфейс источников данных

Источник данных — это объект с CRUD + архитектурой, предоставляющий доступ к типовым операциям (*create, read, update, delete,* ...), применяемым к объекту предметной области (сущности).

Задачи источников:

* обеспечение получения данных с широкого спектра хранилищ;
* порождение сущностей Model и DataSet, содержащих данные из хранилища.

Интерфейс источника данных определяет набор операций, выполняемых с данными. Реализован в интерфейсах Types/source:ICrud и Types/source:ICrudPlus (merge, copy и move).

Список операций:

* create — создать модель (при этом она не сохраняется в хранилище);
* read — прочитать модель по первичному ключу;
* update — обновить модель;
* destroy — удалить модель по первичному ключу;
* merge — объединить две модели;
* copy — скопировать модель;
* query — получить список моделей по запросу;
* call — вызвать метод (только для RPC источников).

Термины

* **провайдер** — объект, предоставляющий доступ к хранилищу данных определенного вида. Зачастую это реализация стандартной технологии и/или протокола (например, HTTP, RESTful, JSON-RPC, XML-RPC, SOAP);
* **хранилище данных** — программно-аппаратный комплекс, предоставляющий API для удаленного получения данных. Должен работать по протоколу, используемому провайдером;
* **endpoint** (конечная точка) — настройка, обеспечивающая доступ клиента к функциональным возможностям источника данных. Минимальными настройками конечной точки являются адрес подключения и имя (или адрес) сущности. Если протокол взаимодействия требует дополнительных настроек (например, идентификации клиента при подключении), то спецификация конечной точки расширяется;
* **binding** (привязка) — настройка, определяющая соответствие методов CRUD+ контракту. Определяет, как именно конкретный источник реализует каждый метод CRUD+.