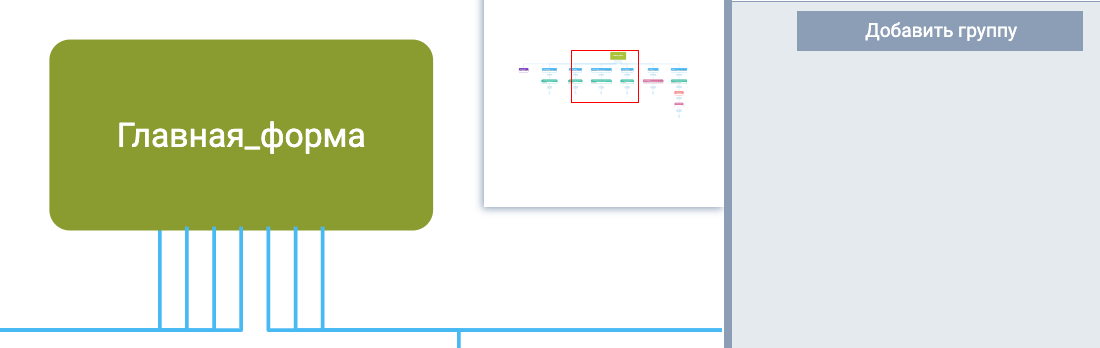
**Блоки диаграммы**

В этом уроке будут разобраны:

|  |  |  |
| --- | --- | --- |
| * блок Формы * блок Группа событий | * блок Событие * блок Элемент | * блок Сценарий формы * блок Сценарий сервера |

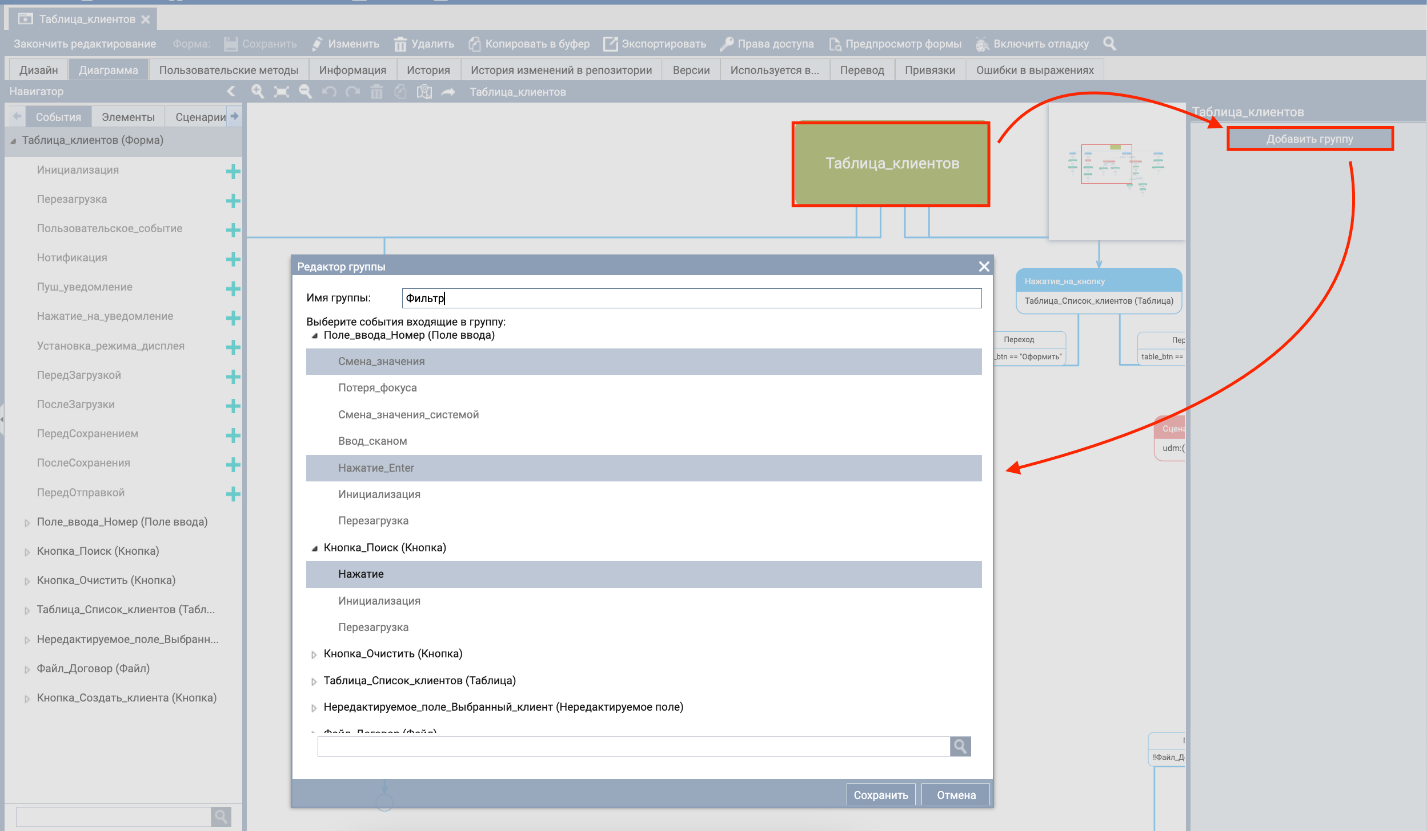
**Блок Формы**



Дефолтный зеленый блок на диаграмме. Его нельзя удалить или отредактировать. Из него берут начало все ветки диаграммы.

**Блок Группа событий**

Для того, чтобы визуально диаграмма читалась легче, рекомендуется объединять логически одинаковые события в группу событий. Группа событий — это специальный блок в диаграмме, объединяющий в себе несколько веток — обработчиков событий. Группу событий можно добавить к Блоку формы

Объединение нескольких событий в Группу событий

Сравните две диаграммы (изображения можно увеличить, кликнув по ним):

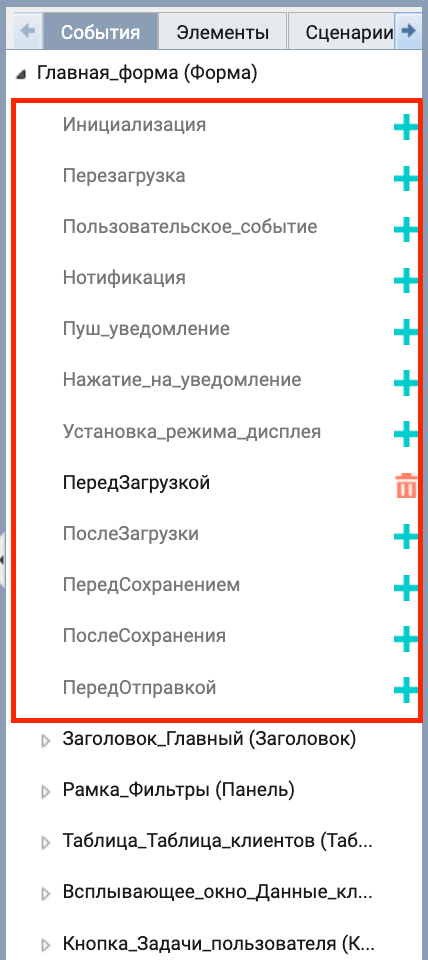
|  |  |
| --- | --- |
| диаграмма без Групп событий | диаграмма с Группами событий |

**Блок Событие**

Все события представлены на первой вкладке События в редакторе Диаграммы.

Событие — это обычно синий блок, с которого начинается каждая ветка диаграммы. То есть это триггер, с которого начинает выполняться настроенная логика действий в Case Platform. Блок каждого события можно выбрать только один раз на форме. Условно события можно разделить на два типа — события формы и события каждого виджета.

**События формы**



У каждой формы есть стандартный набор Событий (они выделены красной рамочкой на изображении слева).

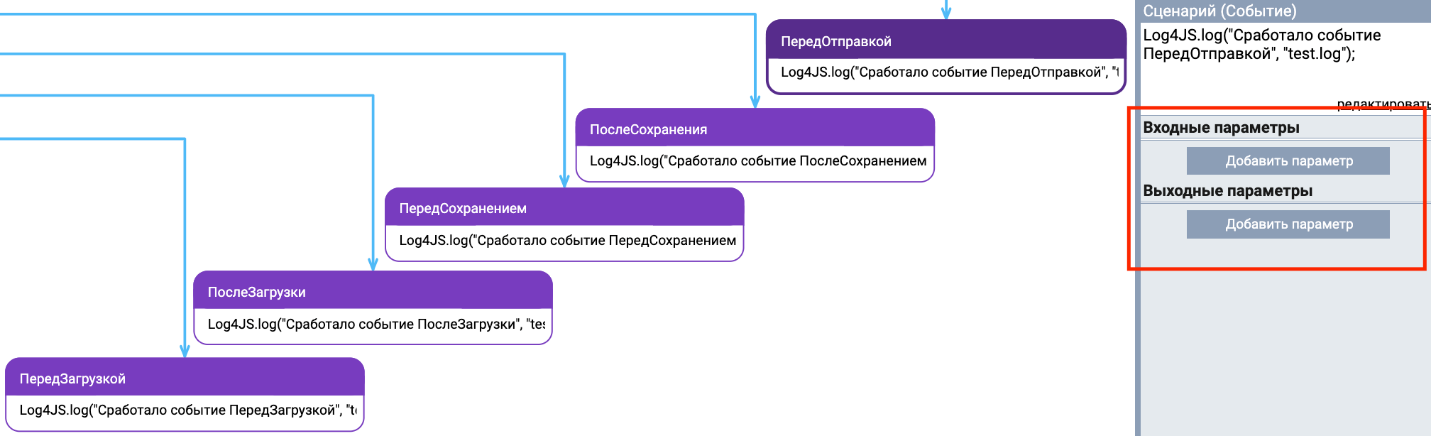
Каждое событие немного отличается друг от друга. Например, блоки ПередЗагрузкой или ПослеСохранения (фиолетовые) позволяют написать **UDMS** код без возможности продолжить ветку события другими элементами. А блок Инициализация (зеленый) позволяет выстроить нужную логику из нескольких блоков и даже нескольких переходов.

❗️Важно понимать порядок исполнения этих событий формы при первом ее открытии:

      1️⃣ — ПередЗагрузкой  
      2️⃣ — ПослеЗагрузки  
      3️⃣ — Инициализация

и порядок исполнения событий при ее закрытии (например, нажатие на кнопку, у которой выставлены свойства *Сохранить\_форму = Да* и *Закрыть\_форму = Применить\_и\_закрыть):*

      1️⃣ — ПередСохранением  
      2️⃣ — ПередОтправкой  
      3️⃣ — ПослеСохранения



Для любого фиолетового блока можно добавить входные/выходные параметры. **Параметры сценария** — это UDM атрибуты, доступные только в рамках этого блока. Во входные атрибуты можно передавать значения перед запуском блока, а выходные передавать наружу при завершении работы скрипта.

Об этих и других событиях формы подробнее описано в [документации](https://doc.caseplatform.tech/#0.0.0:0:0:user_guide/form_builder/form_diagram/form_diagram_events/form_diagram_events_list.html:0:en).

**События виджетов**

У каждого виджета свой набор событий, они определяют разные действия пользователя на форме. Например, нажатие на кнопку, выбор строки в таблице или потеря фокуса на поле ввода. У каждого виджета есть два дефолтных события — Инициализация и Перезагрузка. Остальной набор событий зависит от конкретного виджета.

|  |  |
| --- | --- |
|  | На полотне диаграмм эти события синего цвета.  Для каждого из них есть возможность продолжить  ветку с необходимой логикой. Например, после  Нажатия на кнопку, Сохраняем Рамку с полями  фильтрации и Перезагружаем таблицу  для отображения отфильтрованных данных. |

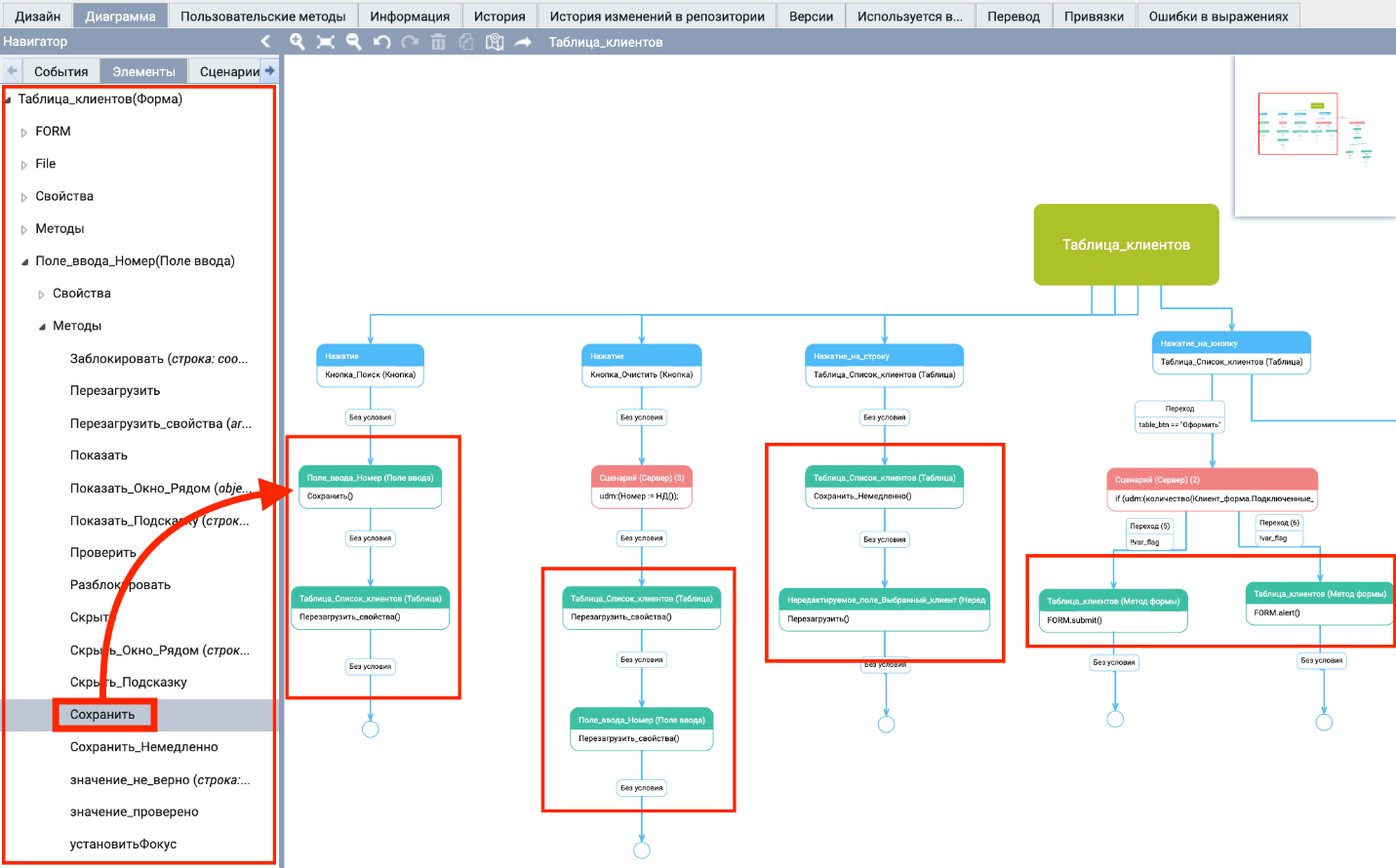
❗️Обратите внимание, что под синей плашкой указано **Системное имя виджета**. Именно для того, чтобы диаграмма была читаемой и понятной с первого взгляда, рекомендуется менять системные имена. При этом необходимо сохранить его первоначальное название и дополнить его названием объекта на интерфейсе 1 к 1. Например, системное имя поля ввода стандартно имеет имя — «Поле\_ввода\_1», если это поле ввода для фамилии, то измените его на «Поле\_ввода\_введите\_фамилию» или «Поле\_ввода\_фамилия» при надписи на интерфейсе “Введите фамилию" или “Фамилия", соответственно.

**Блок Элемент**

Элемент вызывает метод или присваивает значение свойству или стилю, он всегда зеленого цвета. Блоков каждого свойства и метода можно выбрать неограниченное количество раз. Для добавления такого блока на диаграмму нужно

* схватить блок в навигаторе и перетащить на нужную стрелку ([Drag'n'Drop](https://learn.javascript.ru/mouse-drag-and-drop" \t "_blank)), или
* выделить нужную стрелку на диаграмме, затем даблкликом мыши выбрать в навигаторе нужный элемент.

Условно элементы также можно разделить на два типа — элементы формы и элементы каждого виджета.



При выборе блока в меню редактирования появится возможность задавать параметры, указывать значения свойств в **JS**-редакторе и управлять переходами. У каждого блока разные параметры, есть блоки и без параметров. Ниже несколько примеров:

|  |  |
| --- | --- |
|  | Таблица\_Список\_клиентов.Перезагрузить\_свойства(*array: свойства*)  Данный метод виджета Таблица перезагружает указанные в скобках свойства, например, ["Источник\_данных"]. Если свойств несколько, то они указываются через запятую в кавычках. |
|  | FORM.alert(str: *заголовок, str: текст сообщения, class: тип сообщения, bool: модальность*)  Это метод формы, который показывает сообщение на форме с указанными параметрами. В данном случае задан заголовок и текст сообщения. |
|  | Файл\_Договор.Видимость(*bool*)  Этот элемент позволяет изменять свойство виджета. В данном случае виджет Файл станет видимым. Для того, чтобы задать значение свойства, необходимо либо нажать даблкликом на блок, либо справа *Редактировать.* |
|  | Файл\_Договор.Сохранить()  Этот метод формы сохраняет виджет, при этом у него нет входных параметров. В данном случае дополнительно приведен пример добавления дополнительного перехода из этого блока. |

❗️Обратите внимание, что в зеленой плашке блока указано **Системное имя виджета**. Именно для того, чтобы диаграмма была читаемой и понятной с первого взгляда, рекомендуется менять системные имена. При этом необходимо сохранить его первоначальное название и дополнить его названием объекта на интерфейсе 1 к 1. Например, системное имя поля ввода стандартно имеет имя – «Поле\_ввода\_1», если это поле ввода для фамилии, то измените его на «Поле\_ввода\_введите\_фамилию» или «Поле\_ввода\_фамилия» при надписи на интерфейсе “Введите фамилию" или “Фамилия", соответственно.

**Блок Сценарий формы**

Это розовый блок кода. Здесь поддерживается синтаксис JavaScript. В данном блоке можно обращаться к методам и свойствам виджетов и формы (то есть к блокам Элементы), а также использовать управляющие конструкции синтаксиса **JS**.

❗️Сценарий формы исполняется на клиенте, поэтому здесь обычный JS и синтаксические возможности ограничиваются только версией браузера, на котором работает пользователь.

Пример использования блока Cценарий формы:



Ниже представлена одна и та же логика одного события. Слева использованы блоки Элементов — ветка получилась длинная, плоховоспринимаемая и трудночитаемая. А справа использован блок Сценарий формы с теми же методами и свойствами виджетов — гораздо удобнее!

|  |  |
| --- | --- |
|  |  |

При использовании сценариев формы, взамен зеленым блокам Элементов, есть особенности, о них мы поговорим в следующих уроках. А также разберем кейсы работы с атрибутами формы и переменными диаграммы в таком блоке.

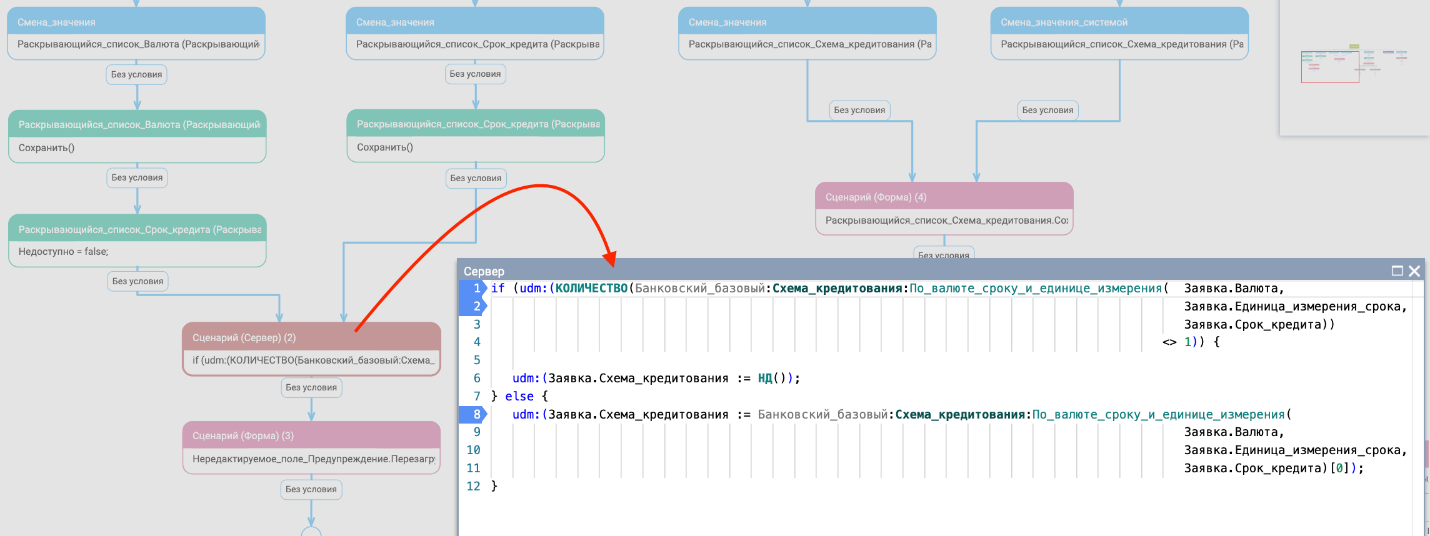
❗️Не забывайте давать говорящие названия сценариям формы вместо дефолтных “Сценарий (Форма) 7”, это значительно упрощает чтение диаграммы формы.

**Блок Сценарий сервера**

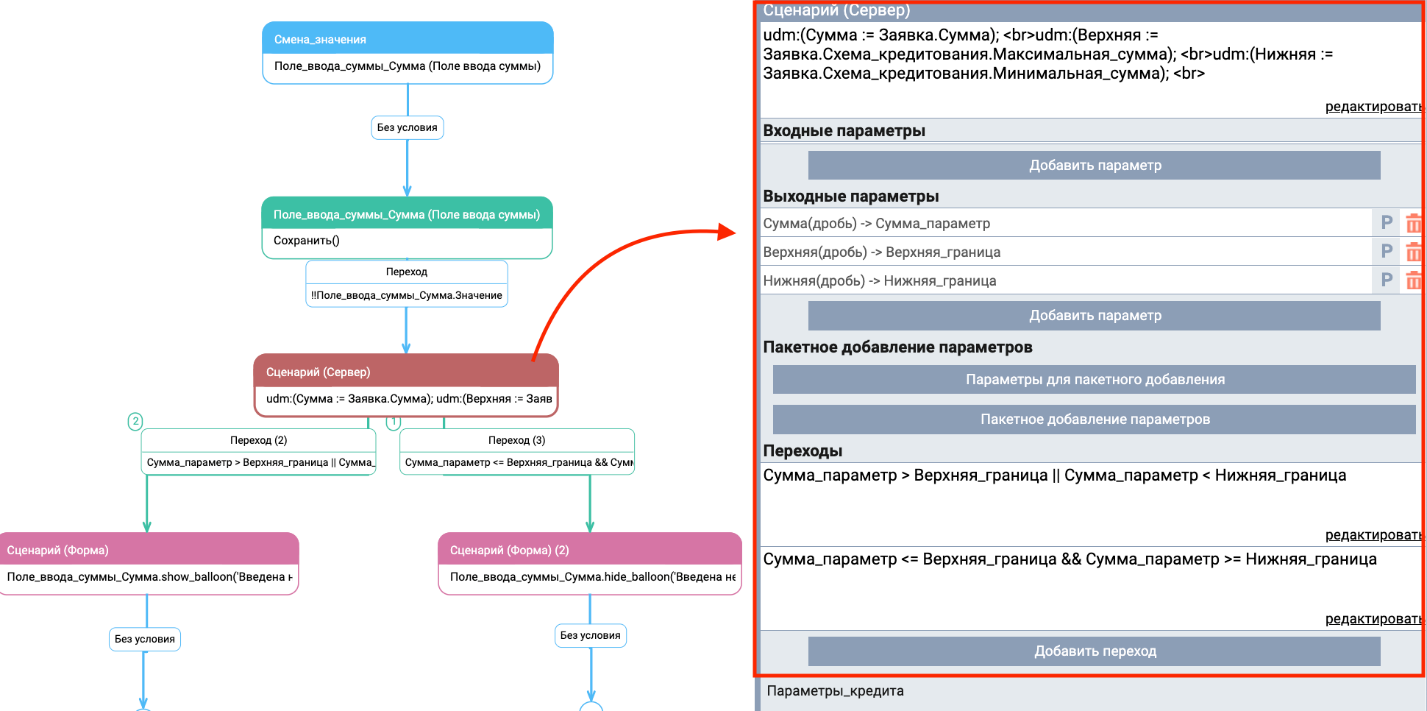
Это красный блок кода. Здесь поддерживается синтаксис **UDMS**. Написанный в таком блоке код, исполняется на сервере.

❗️На сервере используется кастомный JS, который основан на спецификации EcmaScript5 (ES5), поэтому мы не можем применять фичи новых спецификаций ES: объявление переменных через let, стрелочные функции, обратные кавычки, деструкторы и т.д.

Пример использования Сценария сервера:



Справа в редакторе можно отредактировать скрипт, добавить входные/выходные параметры и дополнительные переходы. **Параметры Сценария сервера** — это UDM атрибуты, доступные только в рамках этого блока. Во входные атрибуты можно передавать значения перед запуском Сценария сервера, а выходные передавать наружу при завершении работы скрипта.



*Блок Сценария сервера аналогичен блоку Скрипта в процессе. Поэтому можно не выполнять в процессе то, что можно выполнить на форме.*

❗️Не забывайте давать говорящие названия сценариям сервера вместо дефолтных “Сценарий (Сервер) 7”, это значительно упрощает чтение диаграммы формы.

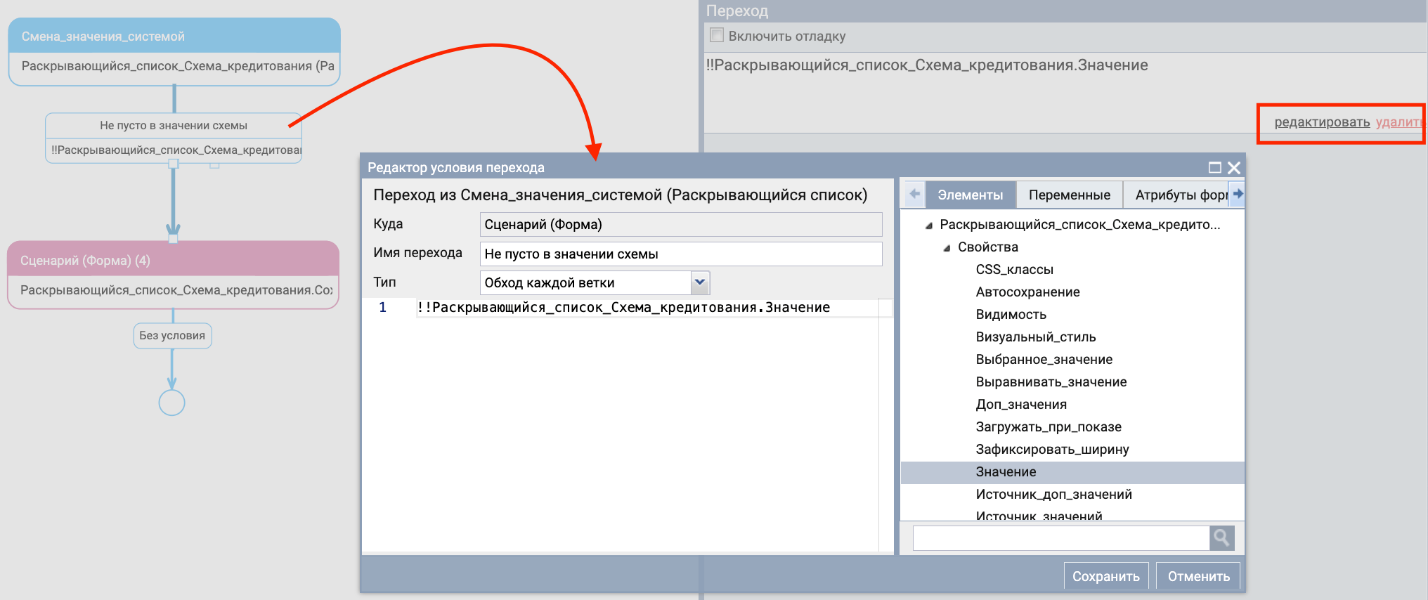
✔️ Мы познакомились с основными блоками и областью их применения. В следующих уроках мы рассмотрим некоторые интересные кейсы и особенности работы с этими блоками.

**Переходы между блоками**

Все блоки диаграммы должны быть соединены переходами. Важно понимать, что можно соединять ветки из разных событий в один блок, а также, наоборот, добавлять ветвление там, где это необходимо. Основное правило — диаграмма должна быть читабельной.

примеры настройки переходов между блоками

Переходы могут быть **безусловными** — дефолтно переход именно такой, он синего цвета с надписью *Без условия*. Также для любого перехода можно добавить **условие перехода** — это должно быть выражение возвращающее true или false.

редактор перехода

Редактор перехода открывается по даблклику или справа кнопкой *Редактировать*. Здесь используется **JS** синтаксис (аналогичный редактор в Сценарии формы). Для удобства можно задать Имя перехода и Тип.

Тип определяет порядок проверки условий.   
**Обход каждой ветки** (выбран по дефолту) — будем проверять все условия по порядку, сколько условий на переходах выполнены, столько веток и отработает система. Стрелки синие.  
**Обход одной из веток —** если не задан порядок обхода, то будем проверять все условия в порядке их создания, пока не наткнемся на первое выполненное. Если порядок обхода задан, то будем идти по заданному порядку, пока не наткнемся на первое выполненное условие. Стрелки зеленые, если задан порядок, то появляется номер очереди в зеленом кружке рядом с условием.

|  |  |
| --- | --- |
| выбор Типа перехода    очередь обхода переходов | условные переходы с заданной очередью |

Микс разных веток, исходящих из одного блока, допускается, но **на практике почти не используется**. Ради интереса, посмотрим на порядок проверки веток в таком кейсе:

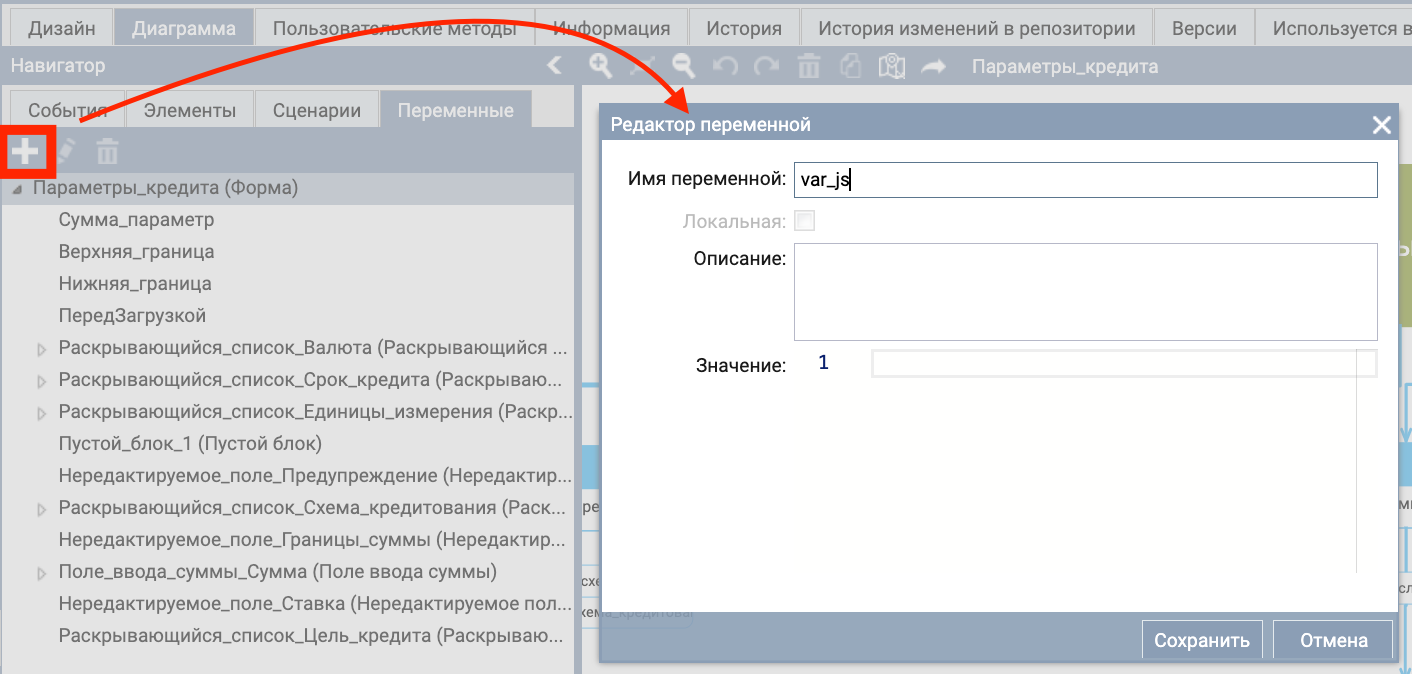
|  |  |
| --- | --- |
|  | 1. Обязательно будет проверяться синяя ветка. Вне зависимости есть на этой ветке условие или нет. 2. Проверится зеленая ветка без порядка очереди. Если условие на ней выполняется, то дальше проверок не будет. 3. Если в п2 зеленая ветка не прошла дальше по условию, то пойдем к ветке с очередью 1. Если условие на ней выполняется, то дальше проверок не будет. 4. Если в п3 зеленая ветка 1 не прошла дальше по условию, то пойдем к ветке с очередью 2. |

❗️ При использовании нескольких переходов в диаграммах рекомендуется задавать им приоритет в порядке от наиболее вероятного к наименее вероятному. При данном подходе в обработчике будет выстроен короткий цикл вычислений, что положительно влияет на оперативность работы формы.

❗️Не забывайте давать говорящие имена Переходам вместо дефолтных “Переход (4)”, это значительно упрощает чтение диаграммы формы.

**Переменные формы**

Переменные на форме создаются в соответствующей вкладке Переменные.

редактор переменной

Для создания необходимо выбрать всю форму (первая строка в списке), в этом случае переменная будет **глобальной**, или конкретное настроенное событие на форме, в этом случае переменная будет **локальной**. Далее нажать ➕ и в открывшемся редакторе указать параметры переменной. В поле *Значение* можно указать значение переменной по дефолту.

Важно понимать, что это JS переменные, доступные только в рамках диаграммы. *Глобальная* переменная доступна в рамках всей формы, а *локальная* — только в рамках конкретного события.

**Для чего нужны переменные формы?**

❗️**Переменные хранятся на клиенте, а атрибуты форм хранятся на сервере. Каждое обращение к атрибутам форм в блоках Элементы (зеленые), Сценарии форм (розовые) и переходах — это обращение к серверу. В последствии это может замедлить работу клиента, так как он ждет рассчитанного значения.**

Если мы часто обращаемся в диаграмме к атрибуту формы, то его лучше выгрузить в выходную переменную блока (например, ПередЗагрузкой или Сценария сервера), а далее записать в JS переменную диаграммы. Также переменной можно заменить использование атрибута формы, который был специально создан для проверки чего-нибудь на диаграмме этой формы.

Рассмотрим пример:

|  |
| --- |
| **Задача:** На форме есть таблица с клиентами. Мы выбираем конкретного клиента в таблице и нажимаем кнопку *Оформить услугу*, для оформления одной или нескольких услуг этому клиенту. Выбранный клиент сохраняется в атрибут формы **Клиент\_форма**. В это время на диаграмме происходит проверка — сколько услуг подключено клиенту:   * если подключены все имеющиеся в списке услуги, то показываем *Предупреждение* (alert), о том что нет доступных к подключению услуг. * если у клиента подключены не все предоставляемые услуги, то запускаем процесс *Оформление услуги*. |

**Решение 1 (плохое). Использование атрибута формы в переходе между блоками.**

1️⃣ На форме создадим атрибут Флаг (не входной, bool)

2️⃣ В сценарии сервера, где происходит проверка на количество подключенных услуг, напишем следующий скрипт:

if (udm:(количество(Клиент\_форма.Подключенные\_услуги.Тип\_услуги) != количество(Услуги:Тип\_услуги:ВСЕ()))) {

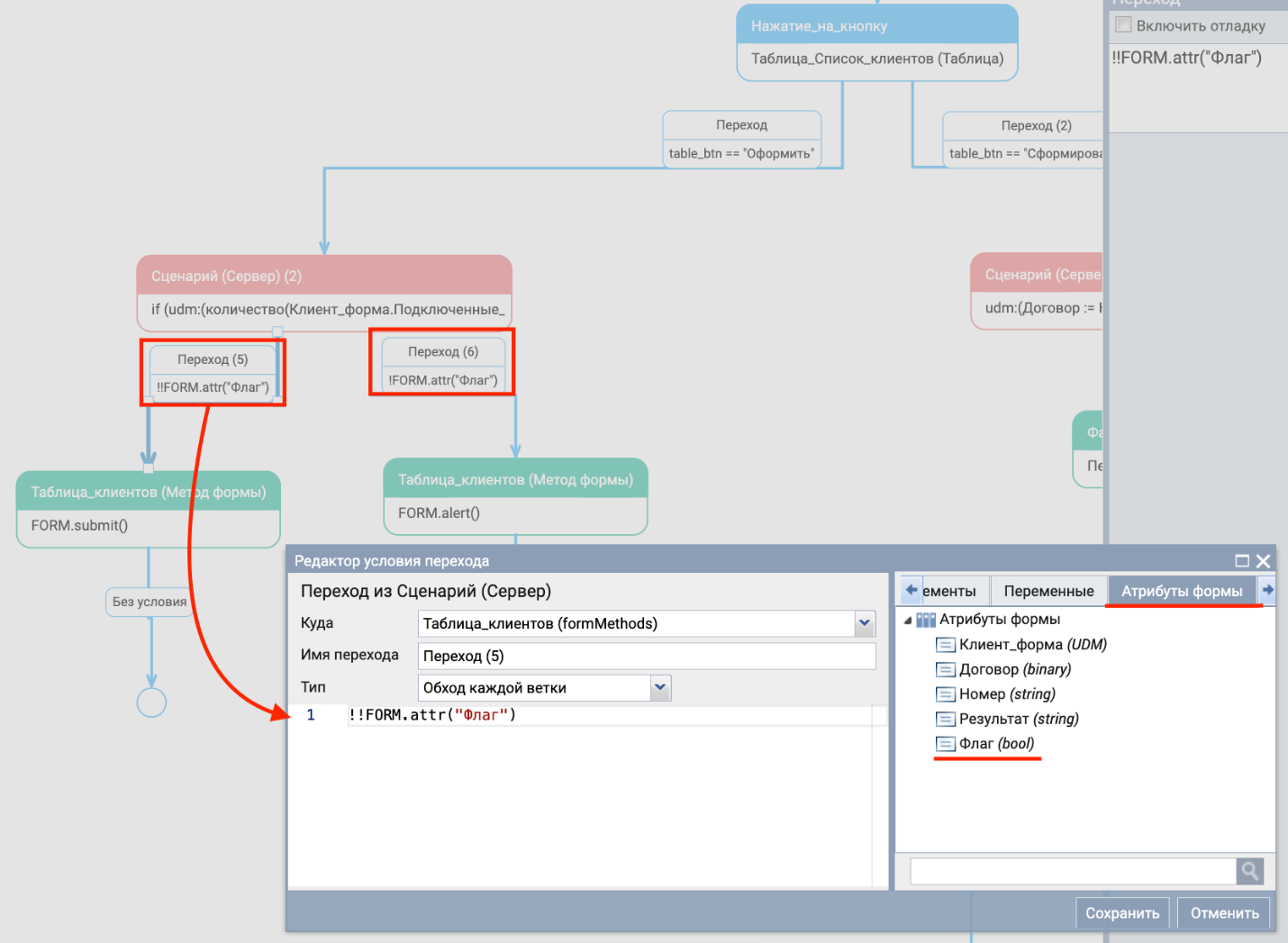
udm:(Флаг := true);

} else {

udm:(Флаг := false);

}

3️⃣  Далее на переходах прописываем условия, используя атрибут формы Флаг



❗️В редакторах Сценария формы, Элементов и переходах можно обращаться к атрибутам формы. При этом используется конструкция FORM.attr("udm\_атрибут"). Важно, что это “верхнеуровневое” обращение к атрибуту, то есть конструкция FORM.attr("Клиент\_форма.Фамилия") является некорректной.

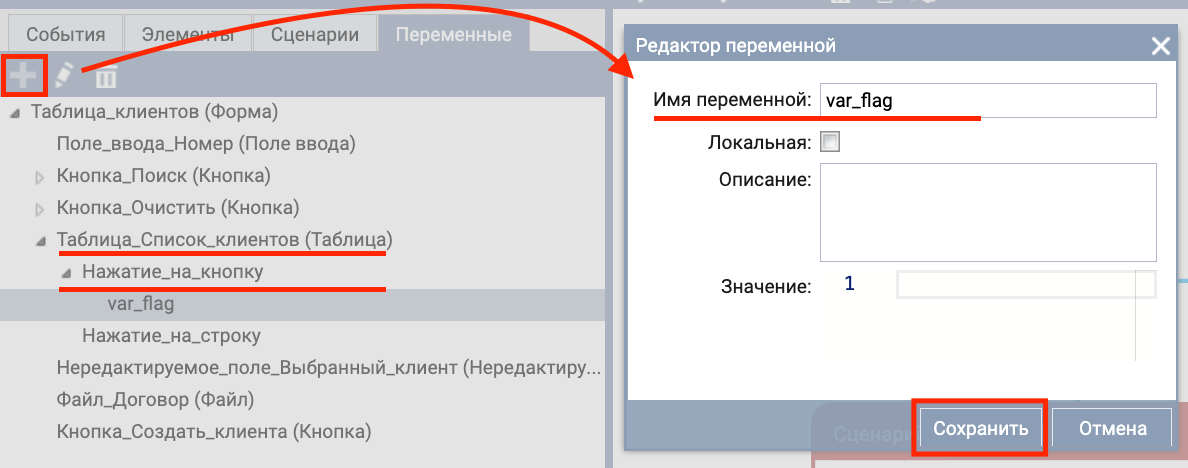
4️⃣ А теперь посмотрим на количество запросов от клиента к серверу при нажатии на кнопку *Оформить услугу*:



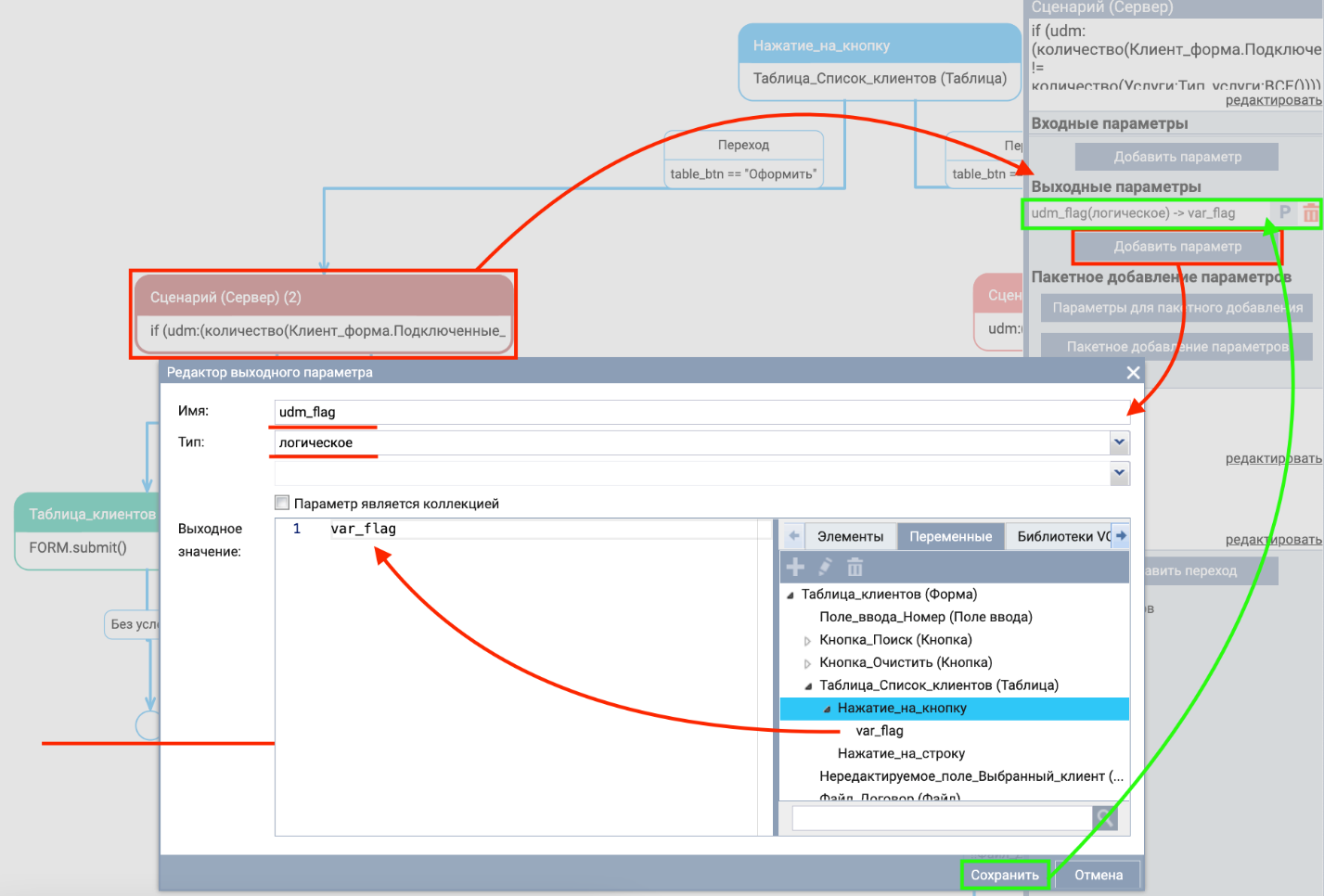
Две записи *udmmessagequeue* говорят об отправке двух запросов на сервер, то есть это расчет выражения FORM.attr("Флаг") и расчет блока сценария сервера.

**Решение 2 (хорошее). Использование JS переменной в переходе между блоками.**

1️⃣ На вкладке Переменные создаем JS переменную var\_flag для События Нажатие\_на\_кнопку (таблица). После создания она появится в списке соответствующего события.



2️⃣ В сценарии сервера, где происходит проверка на количество подключенных услуг, добавляем Выходной параметр udm\_flag (тип — логическое значение). В редакторе параметра указываем выходным значением нашу JS переменную — var\_flag.



После добавления должна появиться запись вида udm\_flag(логическое) → var\_flag, это значит, что бы не присвоилось в Сценарии сервера переменной udm\_flag, при выходе из сценария это значение попадет в переменную var\_flag.

3️⃣ В сценарии сервера напишем следующий скрипт:

if (udm:(количество(Клиент\_форма.Подключенные\_услуги.Тип\_услуги) != количество(Услуги:Тип\_услуги:ВСЕ()))) {

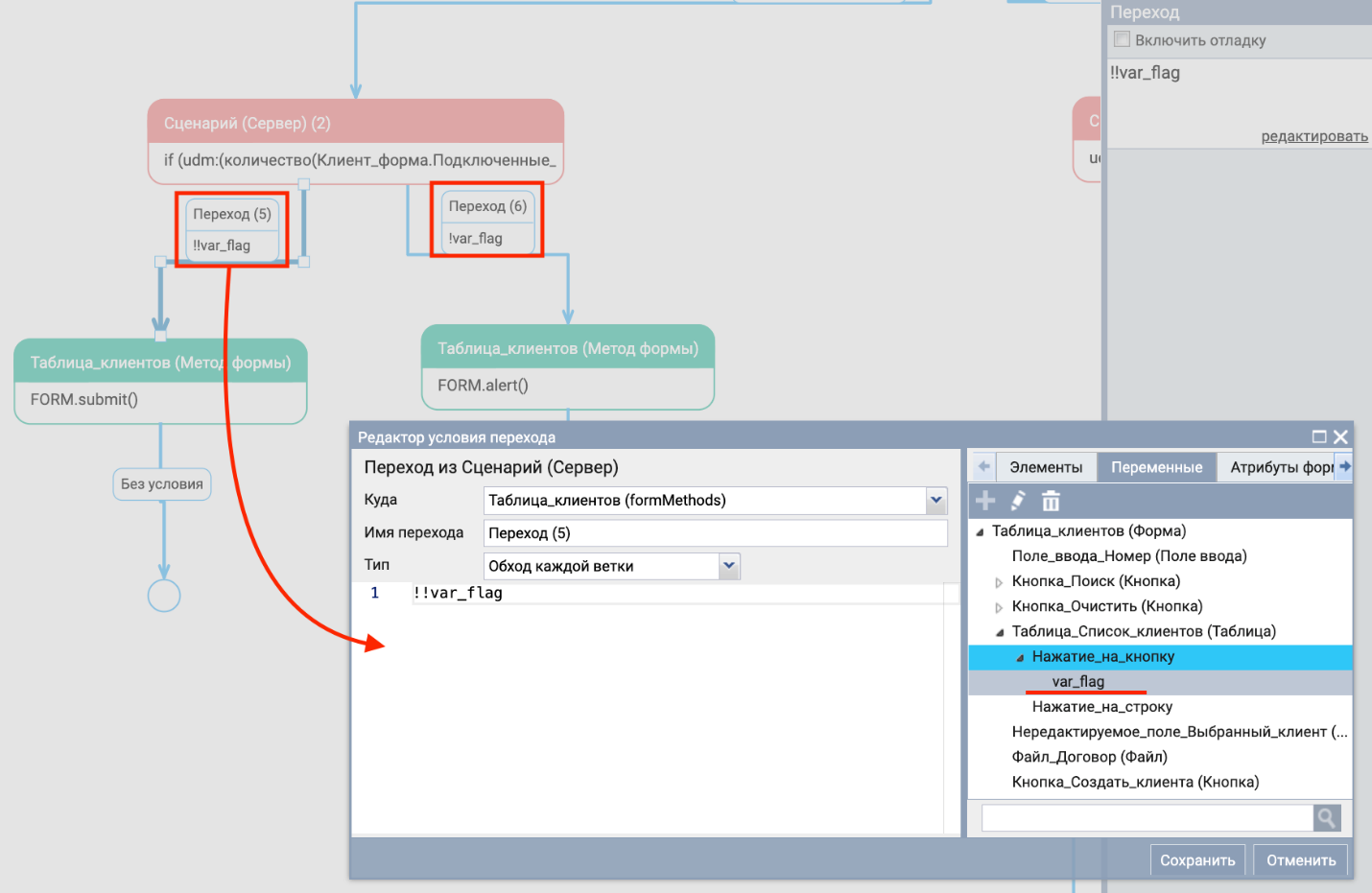
udm:(udm\_flag := true);

} else {

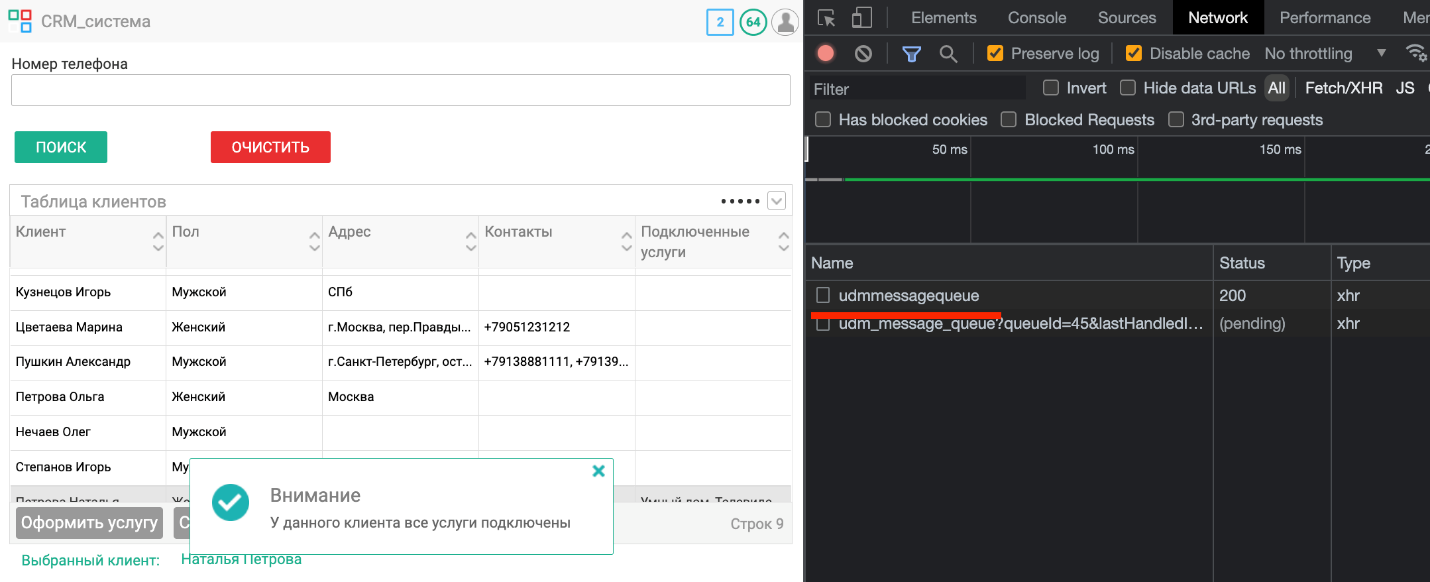
udm:(udm\_flag := false);

}

4️⃣  Далее на переходах прописываем условия, используя JS переменную var\_flag



5️⃣ А теперь посмотрим на количество запросов от клиента к серверу при нажатии на кнопку *Оформить услугу*:

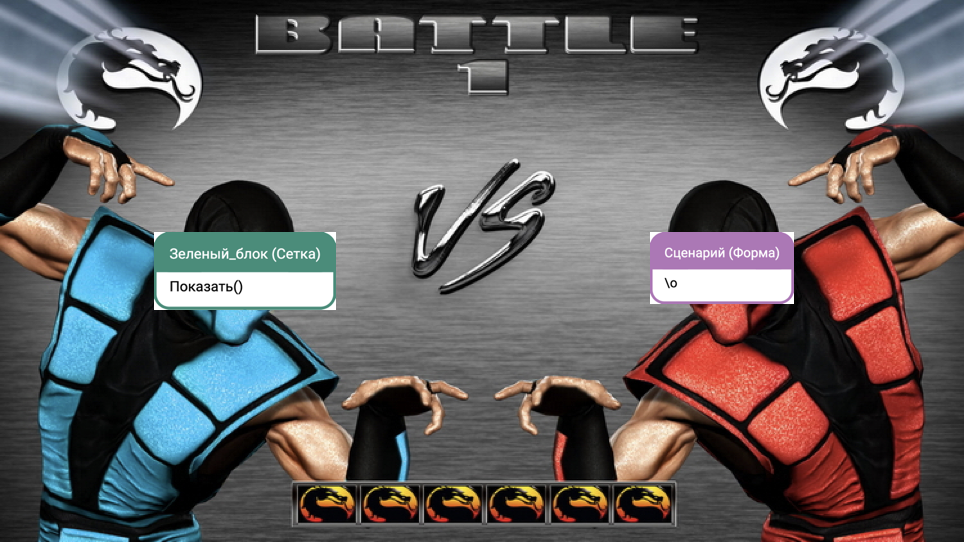


Одна запись *udmmessagequeue* говорит об отправке всего одного запроса на сервер, то есть это расчет блока сценария сервера.

**☝🏼 ВЫВОД:** этот пример показывает, что для проверок в диаграмме оптимальнее использовать JS переменную вместо атрибута формы. Конструкция FORM.attr("udm\_атрибут") отправляет на сервер самостоятельный отдельный запрос на расчет указанного атрибута, поэтому, по возможности, стараемся минимизировать такие конструкции на диаграмме.

**Зеленые блоки vs Сценарий формы**

В прошлых уроках мы рассмотрели случай, когда цепочку зеленых блоков можно заменить на Сценарий формы, но не все так просто. Давайте разбираться подробнее.

картинка для поднятия духа, потому что дальше будет сложно

#### ****Асинхронные и синхронные методы элементов****

В первую очередь, заострим внимание на том, что не все методы работают одинаково. Если говорить на простом языке, то, есть **синхронные** методы, которые выполняются последовательно друг за другом. А есть **асинхронные** методы, когда код продолжит работу без ожидания результата этого метода. Подробнее об этом можно почитать, например, в [статье](https://stasonmars.ru/javascript/polnoe-ponimanie-syncronnogo-i-asyncronnogo-javascript-s-async-await/).

❗️**Асинхронными методами в Case Platform являются — *Перезагрузить(), Перезагрузить\_свойства(), Отрисовать(), Очистить(), Сохранить()* и *Сохранить\_Немедленно()*.**

❗️**Также асинхронным методом будет *Показать()*, если его используем для *Всплывающего окна* (например, для Поля ввода этот метод — синхронный).**

Почему это важно и на что это влияет? Рассмотрим пример:

|  |
| --- |
| **Задача:** На Главной форме есть таблица с клиентами. Мы выбираем конкретного клиента в таблице даблкликом (*on\_rowDblClick*), выбранный клиент сохраняется в атрибут формы **Клиент\_форма**, далее показываем всплывающее окно с данными выбранного клиента. Особенность кейса состоит еще в том, что на всплывающем окне используется **компонента** формы Данные клиента, поэтому накладываются дополнительные условия: через диаграмму сделать всплывающее окно нередактируемым, а кнопки Назад и Завершить сделать невидимыми.  Главная форма, таблица клиентов  Главная форма, всплывающее окно с компонентой |

Необходимо реализовать на диаграмме следующую логику:

|  |
| --- |
| **Таблица**.Двойное\_нажатие\_на\_строку →     **Таблица**.Сохранить\_немедленно →        **Всплывающее\_окно**.Не\_редактируется = true →           **Всплывающее\_окно**.Перезагрузить →              **Всплывающее\_окно**.Показать →                 **Всплывающее\_окно**.**Компонента**.Кнопка.Видимость = false (для обеих кнопок) |

#### ****Решение 1. Через Зеленые блоки****

Это можно сделать с помощью последовательно расположенных блоков Элементов (зеленые блоки).



❗ **Важная заметка**: изменение свойств кнопок на компоненте должно следовать **после** вызова метода Показать(), так как в противном случае всплывающее окно еще не отрендерилось и система не знает о существовании “внутренностей” всплывающего окна. Если поменять порядок вызова Элементов, то система выдаст ошибку:

|  |  |
| --- | --- |
|  |  |

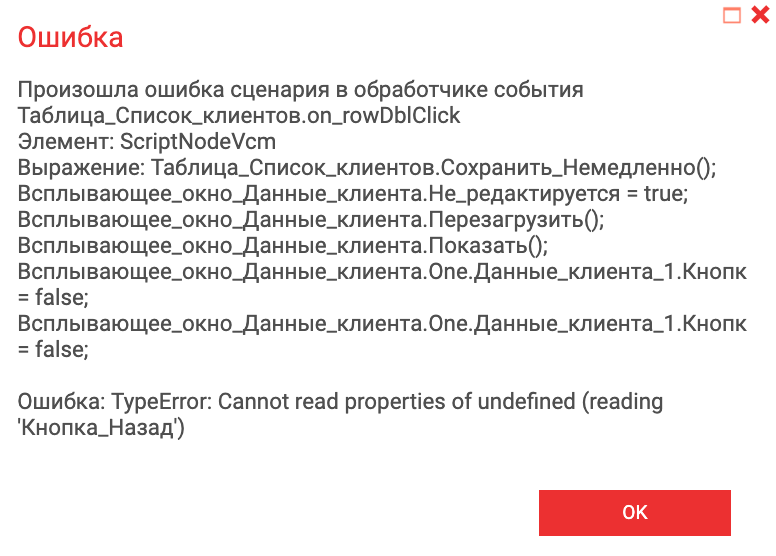
✅ **И еще одна важная заметка**: в данной реализации произойдет **батчинг** (*batching — пакетная обработка*) всех вызываемых методов, и на сервер улетит **один** запрос.

#### ****Решение 2. Через Сценарий формы****

Аналогичную логику можно выстроить в сценарии формы, написав методы последовательно друг за другом

|  |
| --- |
| Скрипт в сценарии формы:  Таблица\_Список\_клиентов.Сохранить\_Немедленно(); Всплывающее\_окно\_Данные\_клиента.Не\_редактируется = true; Всплывающее\_окно\_Данные\_клиента.Перезагрузить(); Всплывающее\_окно\_Данные\_клиента.Показать(); Всплывающее\_окно\_Данные\_клиента.One.Данные\_клиента\_1.Кнопка\_Назад.Видимость = false; Всплывающее\_окно\_Данные\_клиента.One.Данные\_клиента\_1.Кнопка\_Завершить.Видимость = false; |

❌ Но в этом случае система выдаст уже знакомую нам ошибку. Это случилось как раз потому, что методы *Сохранить\_немедленно()* и *Перезагрузить()* **асинхронные**. Порядок исполнения указанных операций нарушился.



✅ В данной реализации также произойдет батчинг всех вызываемых методов в сценарии формы, и на сервер улетит **один** запрос.

#### ****Решение 3. Через Сценарий формы с коллбэком****

В Сценарии формы мы можем принудительно сказать системе, в какой очередности выполнять запросы. Это можно сделать с помощью коллбэков\*.

\*Callback Hell — код, в котором есть обратный вызов, который находится в обратном вызове, который находится в обратном вызове... то есть ситуация, при которой обратные вызовы вложены друг в друга на несколько уровней.

При этом скрипт в Сценарии формы изменится следующим образом:

|  |
| --- |
| Таблица\_Список\_клиентов.Сохранить\_Немедленно(function(){ Всплывающее\_окно\_Данные\_клиента.Не\_редактируется = true; Всплывающее\_окно\_Данные\_клиента.Перезагрузить(function(){ Всплывающее\_окно\_Данные\_клиента.Показать(function(){ Всплывающее\_окно\_Данные\_клиента.One.Данные\_клиента\_1.Кнопка\_Назад.Видимость = false; Всплывающее\_окно\_Данные\_клиента.One.Данные\_клиента\_1.Кнопка\_Завершить.Видимость = false; }); }); }); |

✅ С такой настройкой никаких ошибок уже не будет, кейс отработает корректно.

❌ В данной реализации из-за использования коллбэков на сервер будет отправлено **два** запроса: на сохранение таблицы и на перезагрузку всплывающего окна (*Показать()* отправит только один запрос на сервер при первом открытии, при повторном открытии окна запросы на сервер отправляться не будут, так как это не метод взаимодействия с сервером).

#### ****Подводим итоги****

Давайте соберем все плюсы и минусы каждого из решений, представленных выше:

|  | **Решение 1** | **Решение 2** | **Решение 3** |
| --- | --- | --- | --- |
| **Зеленые блоки** | **Сценарий формы** | **Сценарий формы с коллбэком** |
| **✅** | ***1*** *запрос на сервер* *решение рабочее* | ***1*** *запрос на сервер* | *решение рабочее* |
| **❌** | *отсутствуют* | *получаем ошибки* *при исполнении* | ***2*** *запроса на сервер* |

Таким образом, получается, что **не всегда** можно заменить Зеленые блоки Сценарием формы. В данном случае Решение 1 через Зеленые блоки оказалось лучшим, среди всех остальных.

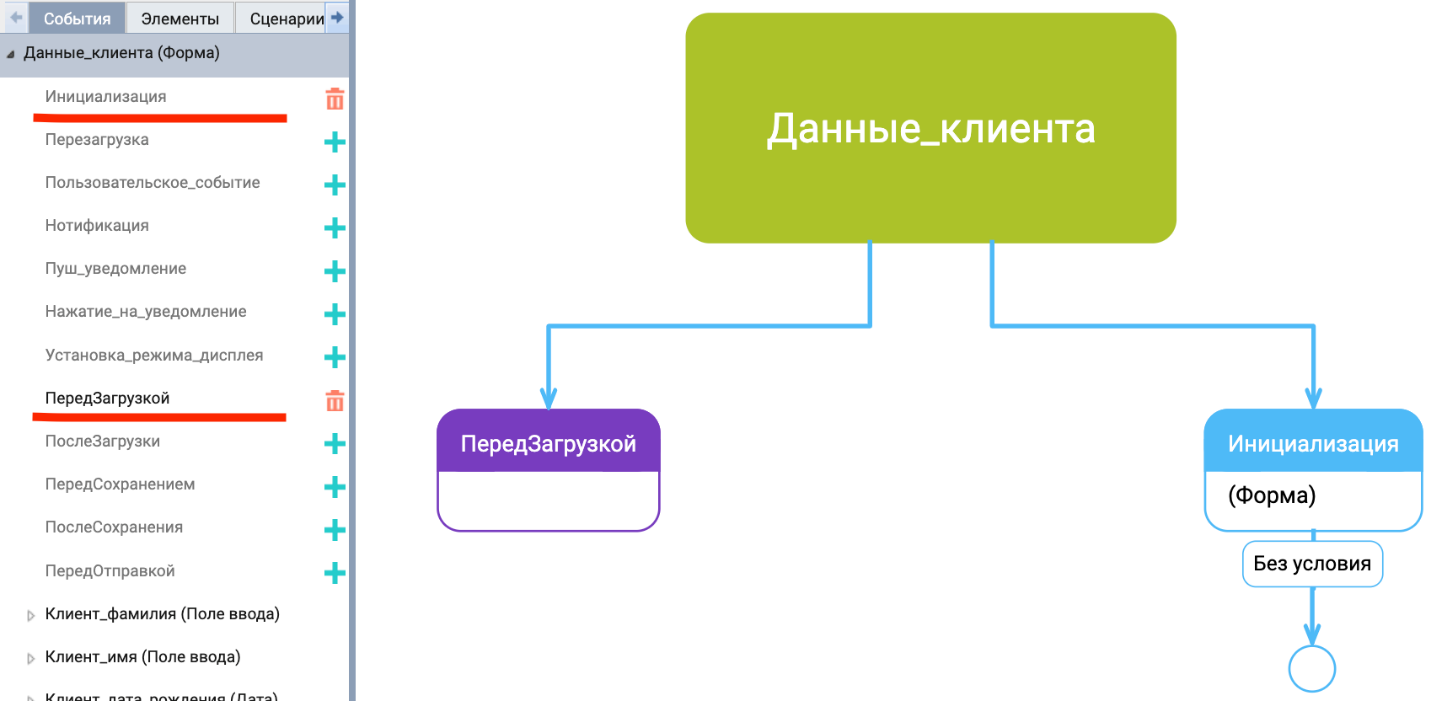
**Немного выводов напоследок:**

Для обеспечения последовательности исполнения **асинхронных** методов элементов в сценариях формы не рекомендуется использовать коллбэки, так как нарушается батчинг запросов, в связи с чем вместо 1 запроса, отправляется несколько запросов. Более того, большой уровень вложенности коллбэков усложняет восприятие кода.

Вместо них рекомендуется использовать отдельные зеленые блоки вызова метода, или же указать последовательно методы в сценарии формы, если требуется вызвать **только** асинхронные методы.

❗️❗️❗️ **И глобальный вывод: всегда диаграмму формы настраиваем вдумчиво. Рассмотренный выше пример не говорит о том, что не нужно использовать коллбэки в коде, или вообще использовать Сценарии формы. Это был всего лишь один пример, который показывает, что прежде, чем что-то 1 раз сделать, надо 7 раз подумать.**

**Инициализация vs ПередЗагрузкой**



Давайте рассмотрим два события — **Инициализация** и **ПередЗагрузкой**. С одной стороны кажется, что это одно и то же, оба что-то делают на старте просмотра пользовательской формы и, кажется, нет разницы, что именно использовать. На самом же деле, разница, конечно, есть.

Формально:

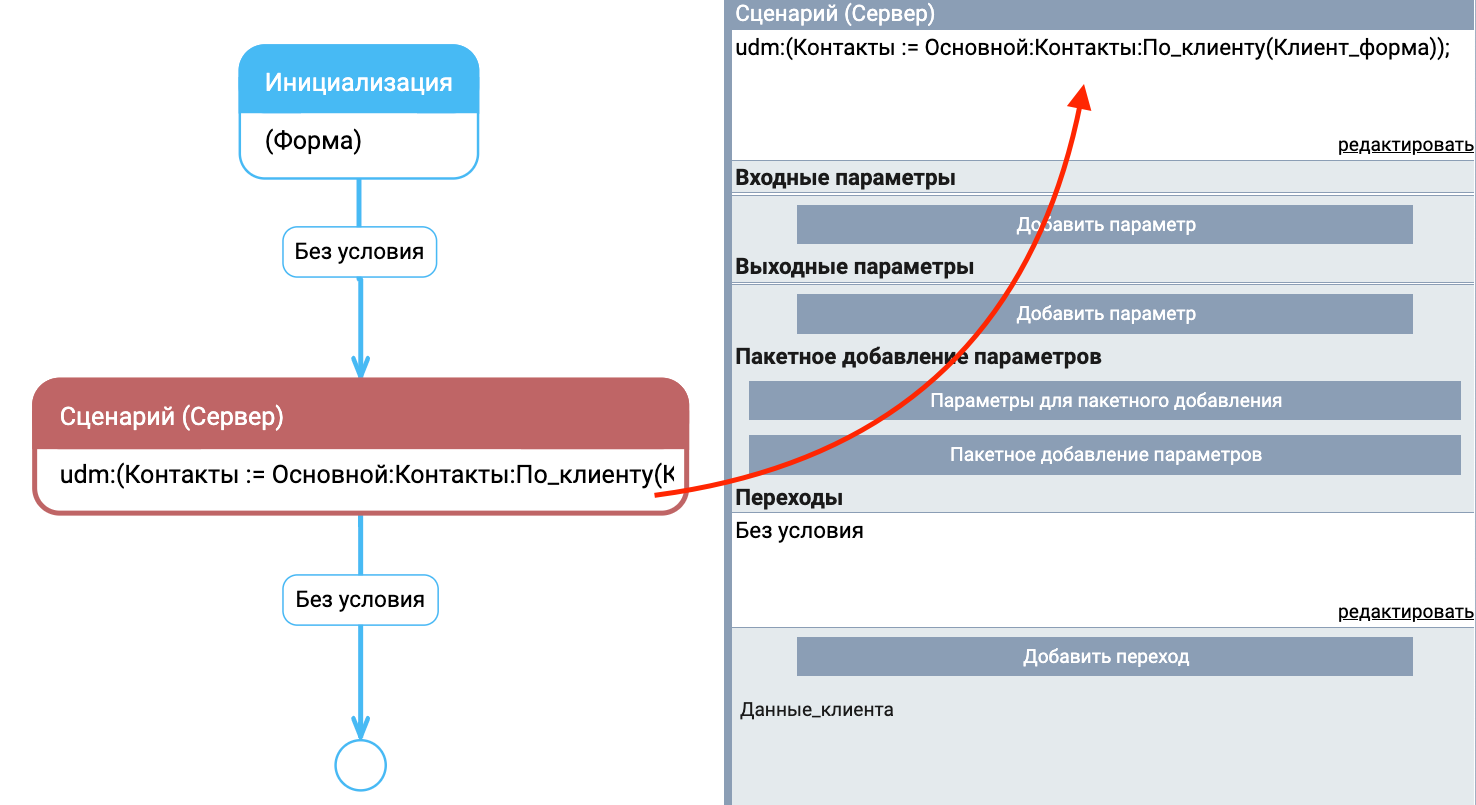
**Инициализация** (*on\_init*) — событие срабатывает **на первой загрузке** (инициализации) формы.   
**ПередЗагрузкой** (*on\_before\_render*) — событие срабатывает **перед загрузкой** формы.

❗️Важно понимать, что сначала отрабатывает событие **ПередЗагрузкой**, потом **Инициализация**. Но есть примеры-исключения, например, если у нас Компонента с событием **ПередЗагрузкой** (дочерняя форма) добавлена во Всплывающее окно, а потом мы это Всплывающее окно несколько раз перезагрузим, то это событие сработает во всех случаях перезагрузки, а **Инициализация** только в первый раз.

**Рассмотрим пример:**

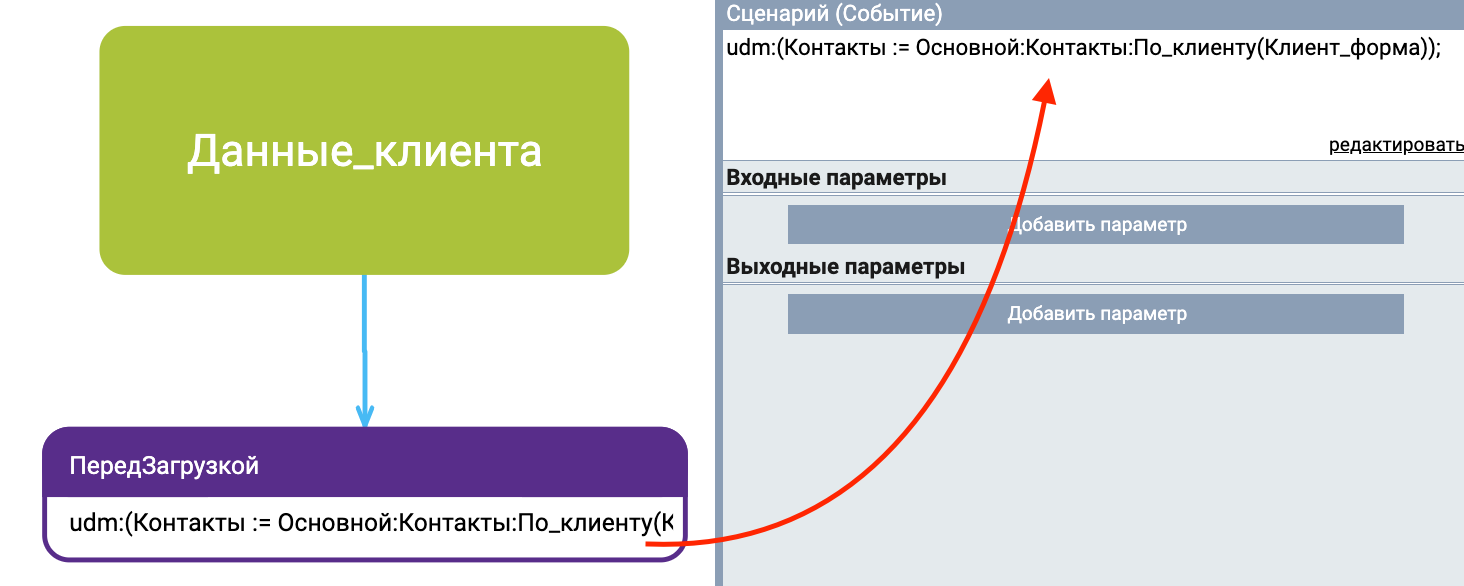
Пусть есть форма **Данные\_клиента** с коллекционным атрибутом *Контакты[]* (тип данных — Тип Контакты) и атрибут *Клиент\_форма* (тип данных — Тип Клиент). При открытии формы необходимо в этот атрибут поместить результат выполнения поискового метода *Основной:****Контакты****:По\_клиенту(Клиент\_форма)*, то есть найти все контакты клиента.

**Решение 1** (плохое): заполнить атрибут через событие Инициализация и блок Сценарий сервера



✅ В данном решении сначала на сервер полетит запрос для загрузки формы и всех выражений в ней, и уже после этого, при исполнении события Инициализация, от Сценария сервера полетит еще один запрос на сервер. Итого улетит **два** запроса.

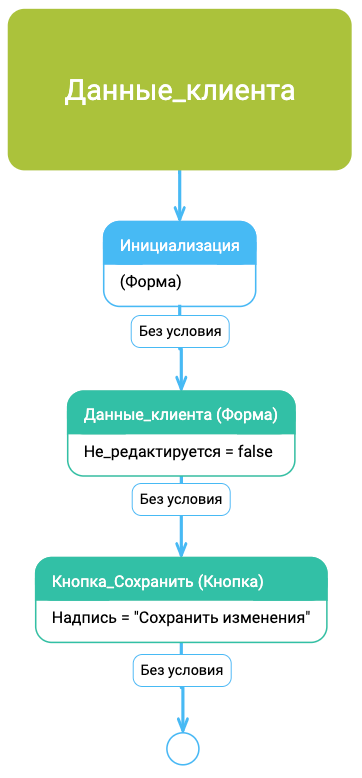
**Решение 2** (хорошее): заполнить атрибут через событие ПередЗагрузкой



✅ В данном решении на сервер полетит запрос для загрузки форм и всех выражений в ней, в том числе вместе с ним и скрипт из блока ПередЗагрузкой. Итого улетит **один** запрос.

❗️ **Вывод: если нам нужно провести расчет выражений с использованием UDMS-скрипта, то лучше это делать с помощью блока ПередЗагрузкой, чтобы не слать лишние запросы от клиента на сервер. То есть, схема “*Инициализация → Сценарий сервера”* не рекомендуется к использованию.**

В явном виде отметим, что блок ПередЗагрузкой это UDMS скрипт, исполняемый на сервере, а значит в нем нет возможности обратиться к свойствам экранных виджетов. Таким образом, если при открытии формы вам нужно поменять свойства виджетов на клиенте, то используйте Инициализацию и набор блоков Элементы (зеленых блоков).



Также рекомендуем не злоупотреблять событием Инициализация, зачастую требуемую логику можно настроить в самих свойствах виджетов на вкладке Дизайн.

**Сохранить vs Сохранить\_немедленно**

Метод **Сохранить()** (save) — это метод виджета и формы, который сохраняет редактируемые виджеты на форме.

Метод **Сохранить\_немедленно()** (force\_save) — это метод виджета, который сохраняет редактируемые и нередактируемые виджеты на форме.

*\*Далее эти методы упоминаются без скобочек для простоты восприятия*

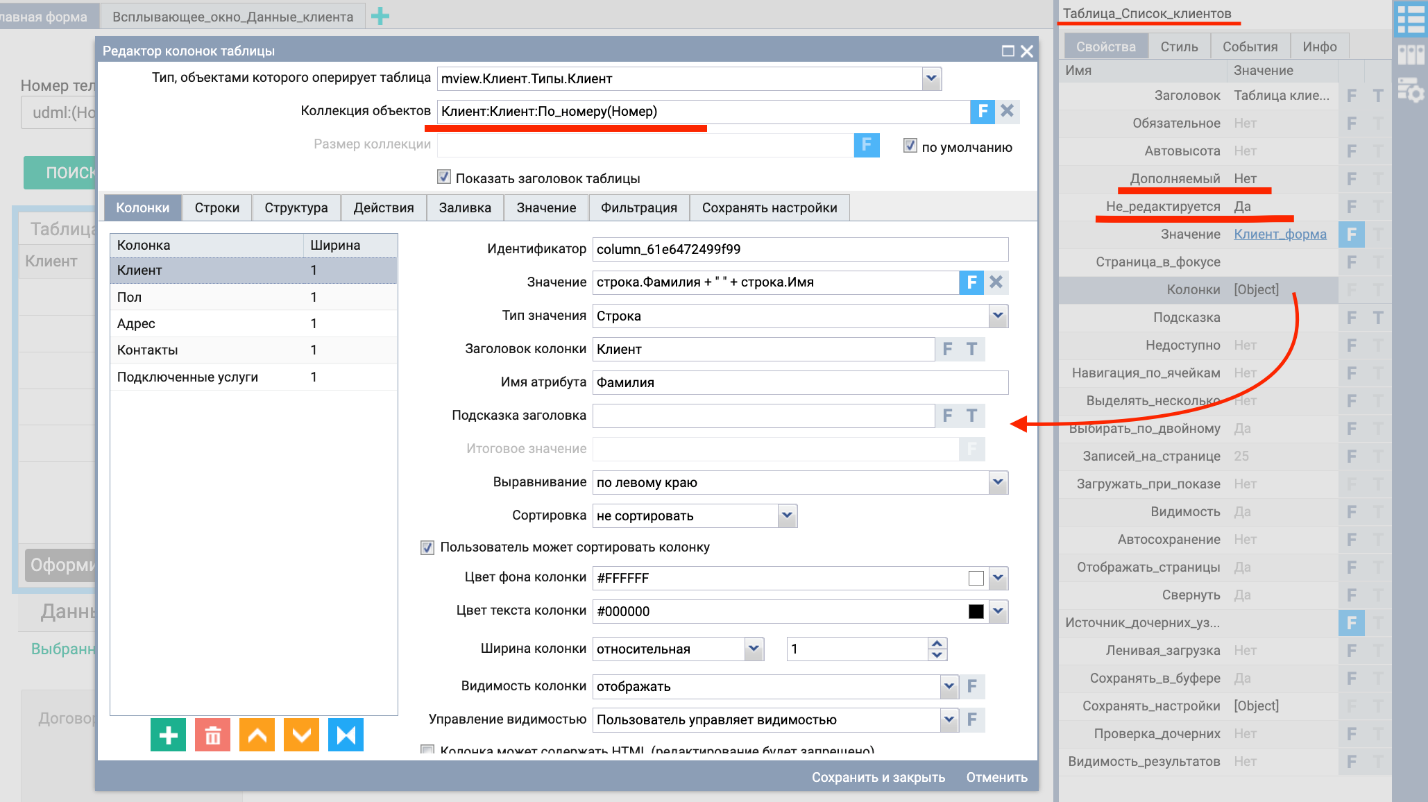
❗️Для редактируемых полей, выпадающих списков и подобных виджетов достаточно использовать Сохранить, а для сохранения выделенной строки в нередактируемой таблице надо использовать Сохранить\_немедленно.

Сейчас вы могли подумать “*Буду везде использовать Сохранить\_немедленно и не надо запоминать все эти тонкости*”. Но мысль неверная. Всегда используем те методы, которые необходимы и достаточны в конкретной ситуации.

**Почему вообще возникает ситуация, когда надо использовать Сохранить\_немедленно?**

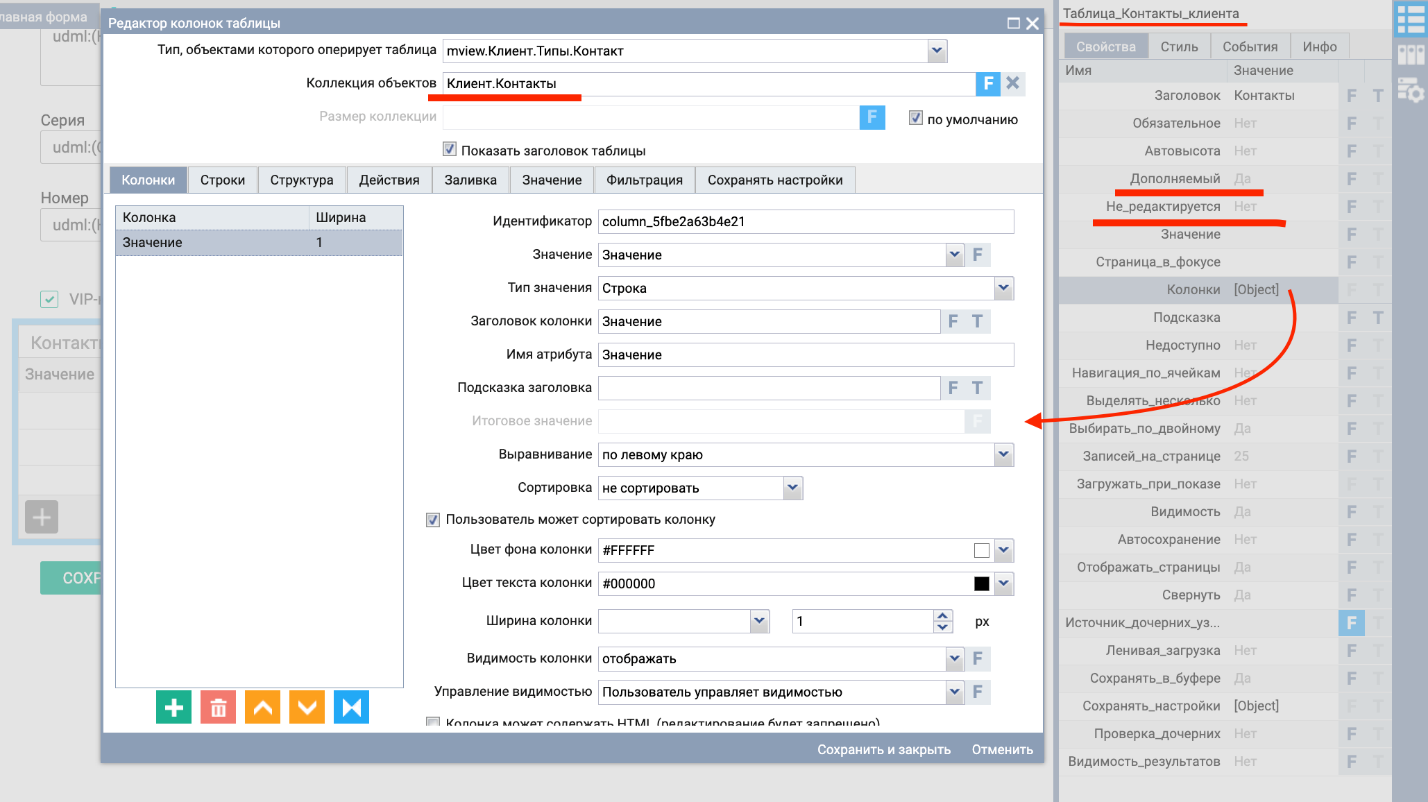
Рассмотрим пример с таблицей.

**Кейс 1 (поисковый метод):** коллекция объектов таблицы формируется **с помощью поискового метода** Клиент:**Клиент**:По\_номеру(Номер). Так как это динамически рассчитываемое выражение, то мы не можем таблицу сделать редактируемой и дополняемой. А это значит, что наша таблица становится нередактируемым виджетом.



Поэтому для сохранения выбранного значения (строки) в таблице ее надо **принудительно сохранять** (Сохранить\_немедленно), а не просто сохранять (Сохранить).

**Кейс 2 (коллекционный атрибут формы):** коллекция объектов таблицы берется **из коллекционного атрибута формы** *Клиент.Контакты[]*. Эта коллекция статичная, рассчитанная заранее, до того как таблица отрендерилась на форме. В такую коллекцию мы можем добавлять объекты, удалять и редактировать, таким образом таблица остается дополняемой и редактируемой.

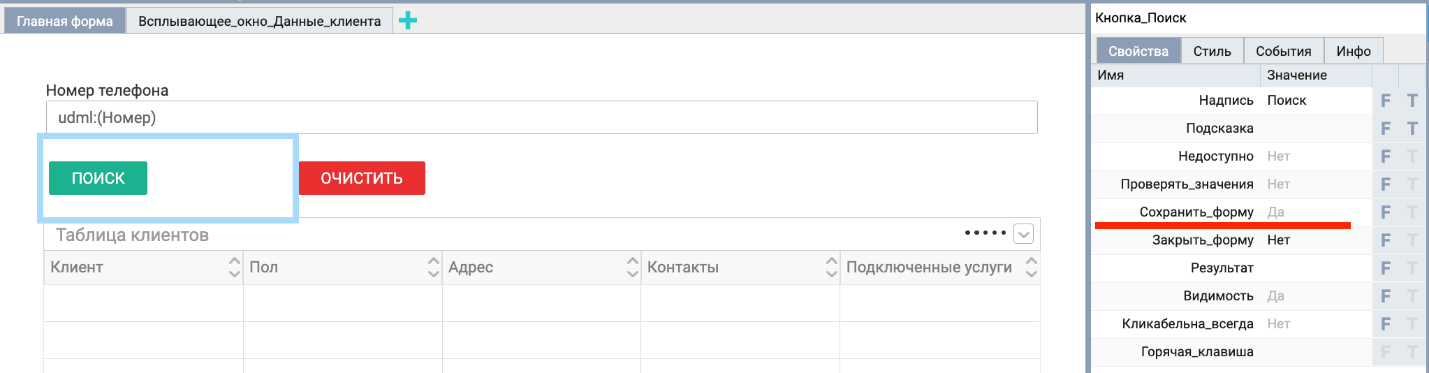


Поэтому для сохранения выбранного значения (строки) в таблице ее достаточно просто **сохранить** (Сохранить).

**Что еще полезно знать про Сохранить?**

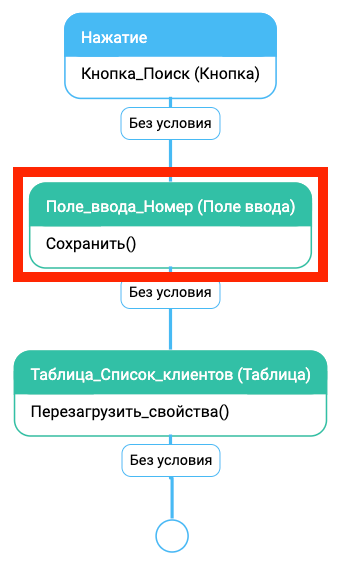
Опять же, рассмотрим пример 😉

У каждой кнопки есть дефолтное свойство *Сохранить\_форму*, принимаемые значения — Да/Нет (true/false). При нажатии на кнопку, если выставлено *Сохранить\_форму = Да*, будут вызваны методы *Сохранить* для всех редактируемых виджетов.



На скрине выше кнопка Поиск должна сохранить введенный номер телефона и перезагрузить таблицу, для отображения подходящих записей по фильтру.

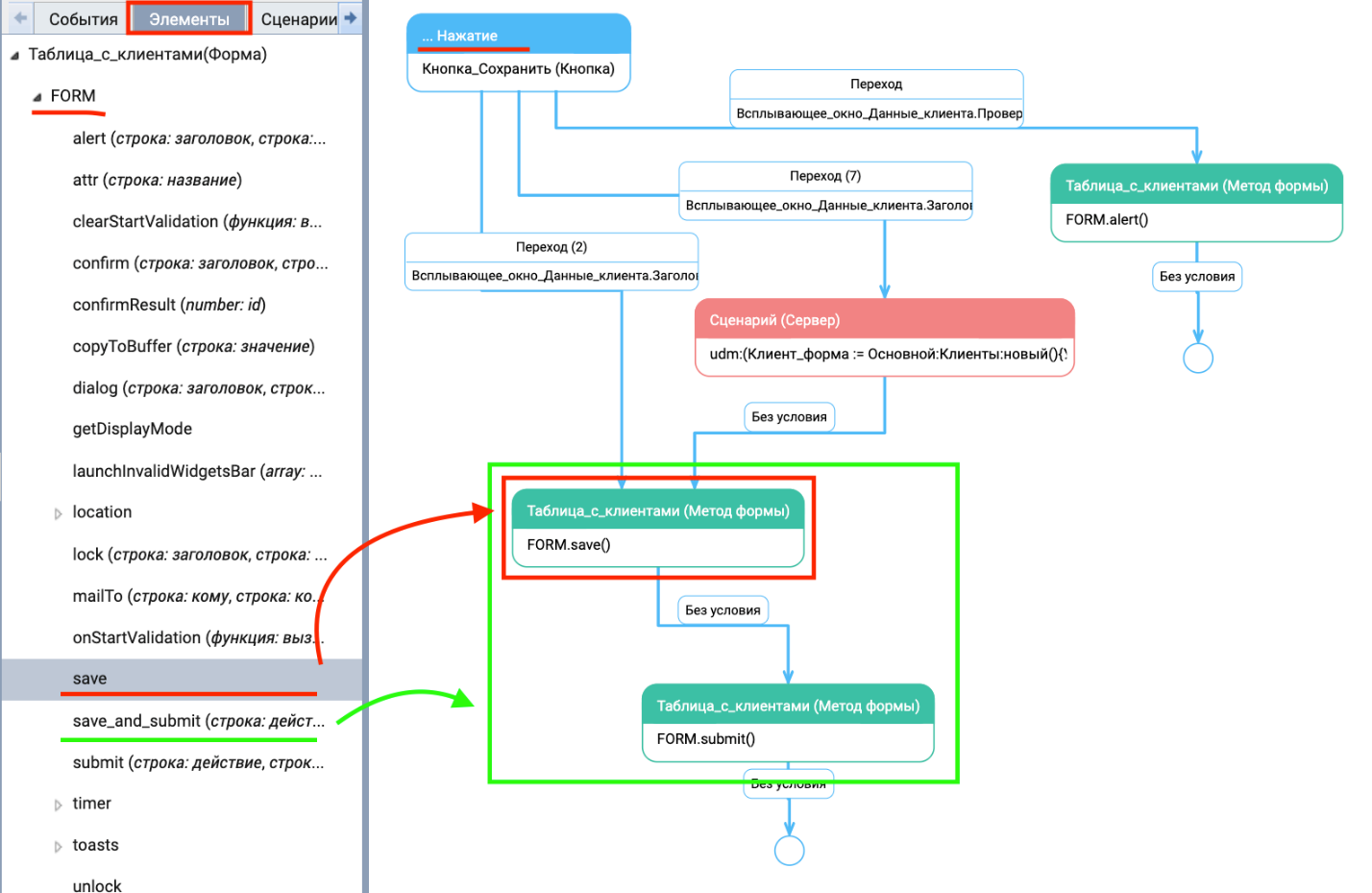
* Если в данном случае оставляем *Сохранить\_форму = Да*, то при нажатии будут вызваны методы Сохранить для поля ввода номера и *других редактируемых виджетов* на форме (а их может быть сколь угодно много).
* Если выставим *Сохранить\_форму = Нет*, то необходимо для корректной работы на ветку диаграммы формы добавить сохранение поля ввода после нажатия на кнопку и перед перезагрузкой таблицы. **Такая настройка будет более оптимальной для такого кейса.**



**Что еще предложите?**

Что если при нажатии на кнопку рановато сохранять форму, а сделать это нужно после выполнения какой-нибудь логики на диаграмме?

В этом случае у кнопки выставляем свойство *Сохранить\_форму = Нет*, и уже в диаграмме вызываем метод формы ***FORM.save()***:



**→** Также, есть метод ***FORM.save\_and\_submit(),*** который сохранит все редактируемые виджеты и закроет форму. Его можно использовать вместо двух отдельных зеленых блоков, если логика предполагает сохранение и закрытие формы.

❗️ Альтернативой методу ***FORM.save()*** является просто метод ***Сохранить()*** у формы (выбирается из группы Методы формы). Если нужно сохранить данные только с этой формы (или компоненты), то лучше использовать его. А если нужно сохранить все-превсе данные формы (как блока процесса), то используем ***FORM.save()***.

**Перезагрузить vs Перезагрузить\_свойства**

Метод **Перезагрузить()** (reload) — это метод виджета, который перезагружает все свойства виджета. Порядок перезагрузки свойство зависит от того, в каком порядке эти свойства указаны на вкладке Свойства.

Метод **Перезагрузить\_свойства()** (reload\_properties) — это метод виджета, который перезагружает свойства виджета, указанные в виде входных параметров этого метода.

*\*Далее эти методы упоминаются без скобочек для простоты восприятия*

❗️**Рекомендуется избегать большого количества перезагрузок виджетов на формах. Лучше применять перезагрузку свойств виджета. Потому что каждый reload на форме — это спиннер. Обилие спиннеров нервирует пользователя и занимает время.**

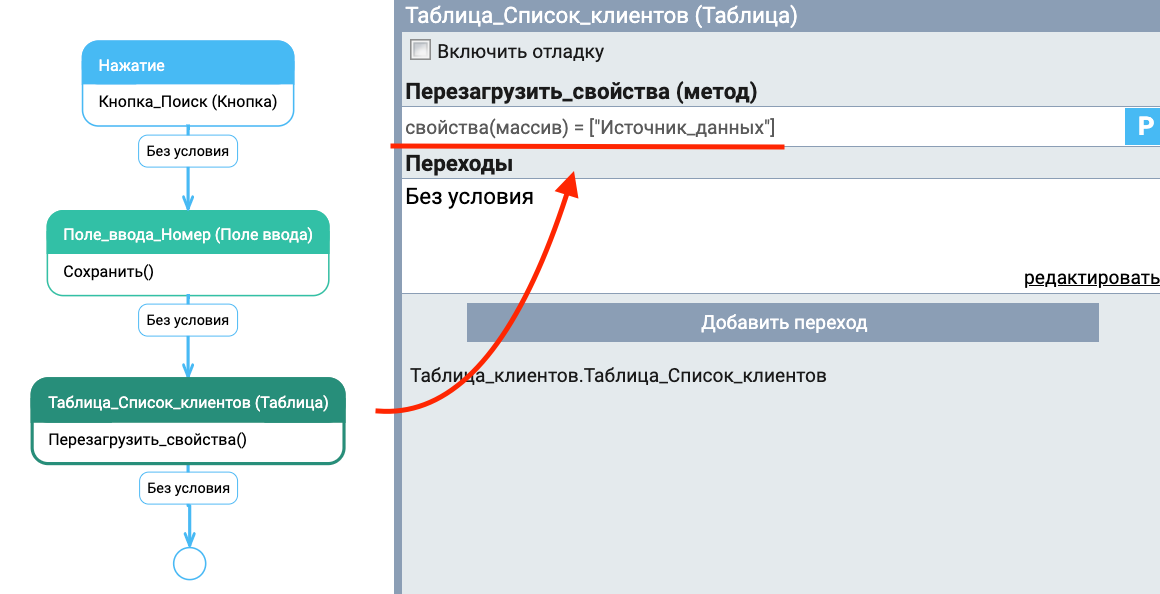
Сравните как работают оба метода:

| ***Перезагрузить\_свойства()*** | ***Перезагрузить()*** |
| --- | --- |
|  |  |
| При использовании *Перезагрузить\_свойства* на интерфейсе отсутствует спиннер. В качестве перезагружаемого свойства в данном случае выступает *Значение* (Value). | *Перезагрузить* при изменении значения раскрывающегося списка показывает спиннер. В данном случае это не критично, но если таких полей на форме много, то это уже будет выглядеть некрасиво и раздражать пользователя. Более того, при сложных расчетах продолжительность отображения спиннера ощутимо увеличивается. |

**Популярный пример**

Представим, у нас есть таблица клиентов и настроенные фильтры для поиска клиентов в таблице. После ввода данных в поля фильтрации и нажатия на кнопку Поиск, должна произойти перезагрузка таблицы, чтобы актуализировать отображаемые данные в таблице.

В данном случае рекомендуется перезагружать только свойство “*Источник\_данных*”:



☝️ Если свойств несколько, то они указываются так же в кавычках через запятую, например, *Перезагрузить\_свойства["Источник\_данных", "Значение"].*

**Идущие подряд одинаковые блоки**

Здесь отметим парочку моментов, которые важно знать при работе с диаграммой.

📌 **Момент 1. Подряд идущие зеленые блоки**

В прошлом уроке мы уже говорили про **батчинг** (*batching — пакетная обработка*) вызываемых операций и отправку их в **одном** запросе на сервер. Давайте немного подробнее еще раз это обсудим.

Это встроенная оптимизация работы диаграммы: чем меньше запросов от клиента летит на сервер, тем клиенту и серверу живется лучше. Подробнее про батчинг запросов на диаграмме формы можно прочитать в [документации](https://doc.caseplatform.tech/#0:0:0:user_guide/form_builder/form_diagram/form_diagram_batching.html:0:ru).

Важно понимать, что не все операции (зеленые блоки) выполняются на сервере. Например, метод ***Таблица.Сохранить\_немедленно*** имеет дело с UDML выражением *udml:(Клиент\_форма)* в свойстве Значение, которое надо перерассчитать, и этот перерасчет будет на сервере. А вот ***Всплывающее\_окно.Видимость = true*** *—* это изменение свойства виджета на клиенте, здесь на сервере делать нечего.

Так вот, предположим у нас есть два метода, которые должны рассчитаться на сервере *—* ***Таблица.Сохранить\_немедленно*** и ***Всплывающее\_окно.Перезагрузить***, если эти методы вызваны через зеленые блоки и между ними **нет условного перехода**, то тогда на сервер полетит один запрос. А если между блоками будет **условие** (например, *!!Таблица.Значение* или *!!FORM.attr("Клиент\_форма")*), то в первый запрос попадут все операции до этого запроса, а во второй *—* после, таким образом на сервер полетит два запроса.

|  |  |  |
| --- | --- | --- |
| на сервер отправится **один** запрос | на сервер отправится **два** запрос | на сервер отправится **два** запроса |

📌 **Момент 2. Подряд идущие Сценарии сервера**

Каждый сценарий сервера *—* это один запрос на сервер.

Не рекомендуется использовать несколько подряд идущих сценариев сервера, рекомендуется их объединить в один. Таким образом будет сокращено количество обращений к серверу, что положительно влияет на оперативность работы формы.

|  |  |
| --- | --- |
| на сервер отправится **один** запрос | на сервер отправится **два** запроса |

📌 **Момент 3. Подряд идущие Сценарии формы**

Все зависит от конкретного кейса и логики в сценарии формы. Работа блоков сценария формы аналогична работе зеленых блоков. Приведенные ниже случаи, по сути являются повтором уже изложенного материала:

а) Если в сценарии формы есть методы взаимодействия с сервером и они асинхронные: в этом случае, при разделении вызова этих методов на разные блоки сценария формы, будет отправлено два запроса на сервер. Правильным решением будет сделать один блок сценария формы и в нем написать подряд вызов асинхронных методов.

|  |  |
| --- | --- |
| *Таблица.Сохранить\_Немедленно();* *Всплывающее\_окно.Перезагрузить();* | |
| на сервер отправится **один** запрос | на сервер отправится **два** запроса |

б) Если вызываемые методы/свойства не требуют расчета на сервере, то можно использовать несколько подряд идущих блоков сценария формы, **но зачастую в этом нет необходимости**. Помним, что диаграмма должна быть понятной и легко читабельной.

В качестве примера, можно рассмотреть следующий код:

|  |  |
| --- | --- |
| *Всплывающее\_окно.Не\_редактируется = true;* *Всплывающее\_окно.Заголовок = "Данные клиента";* | |
| нет запросов на сервер | нет запросов на сервер |

В обоих случаях на сервер не полетит ни одного запроса (методы выполняются на клиенте), но легче воспринимается вариант с одним блоком формы.

**Очередь сообщений в FormRunner**

Для удобства приведем здесь содержание урока:

Вводная. что такое очередь сообщений

Пример про очереди сообщений

Настройки формы

Действия пользователя

Исполнение сценария формы

Сохранение формы

Сообщения в очереди

Профилирование в UDM-консоли

Инструменты разработчика браузера

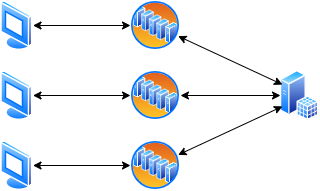
Ожидание исполнения выражений

Коллбэки для методов save() и reload()

Пример

Клиентская часть приложений взаимодействует с сервером через очереди сообщений. Для каждой пользовательской сессии создаётся отдельная очередь UDM.

Эта очередь поддерживает как отправку сообщений от клиента — серверу (например, сохранение виджета, перезагрузка его свойств, исполнение сценариев сервера), так и от сервера — клиенту (например, уведомление о срабатывании нотификации).



Очередь UDM обрабатывает последовательные операции в строгом порядке по принципу [FIFO (First In - First Out)](https://ru.wikipedia.org/wiki/FIFO): первое поступившее выражение будет рассчитано первым, и следующее выражение будет рассчитано только после выполнения первого, даже если эти выражения поступили в одном запросе.

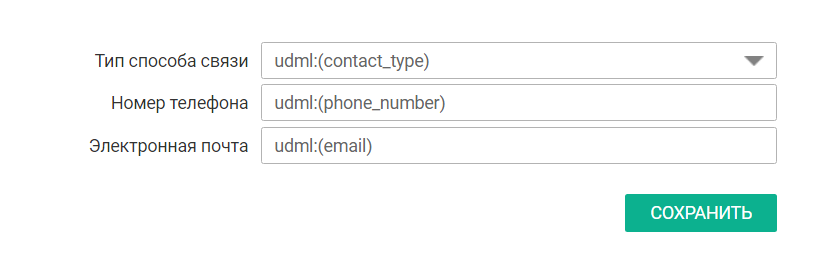
Таким же образом, если в очередь был отправлен запрос, то следующие запросы не могут быть отправлены, пока все выражения из предыдущего не будут вычислены.

## 

## **Пример**

В этом уроке мы рассмотрим детально работу UDM-очереди на примере простой экранной формы. На рисунке ниже представлена экранная форма ввода контакта, состоящая из 4 виджетов:

* **contact\_type** – раскрывающийся список для выбора типа способа связи;
* **phone\_number** – поле ввода с шаблоном для ввода номера телефона;
* **mail** – поле ввода с проверкой для ввода электронной почты;
* **save\_button** – кнопка для сохранения введенных значений.



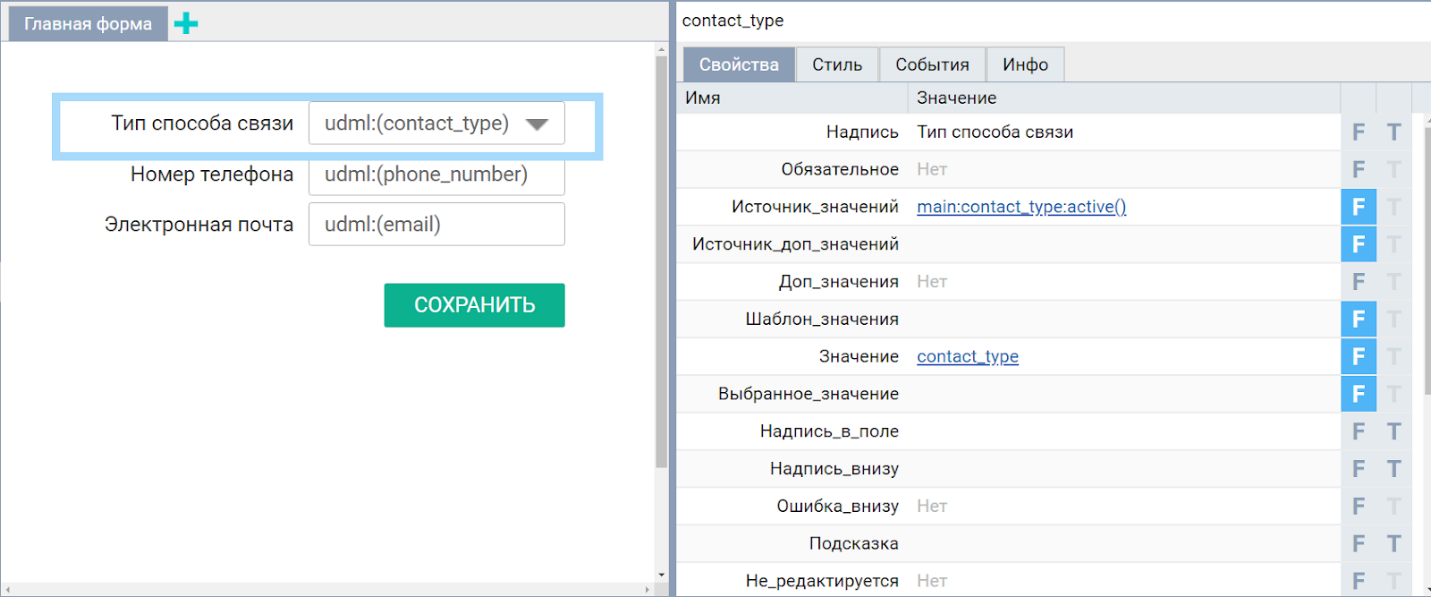
### **Настройки формы**

Для формы определены 3 атрибута:

* **contact\_type** — атрибут, ссылающийся на перечисление, перед загрузкой формы в него записывается значение **main:contact\_type:email**;
* **mail** — строковый атрибут, перед загрузкой формы в него записывается значение **example@mail.com**;
* **phone\_number** — изначально пустой строковый атрибут***.***

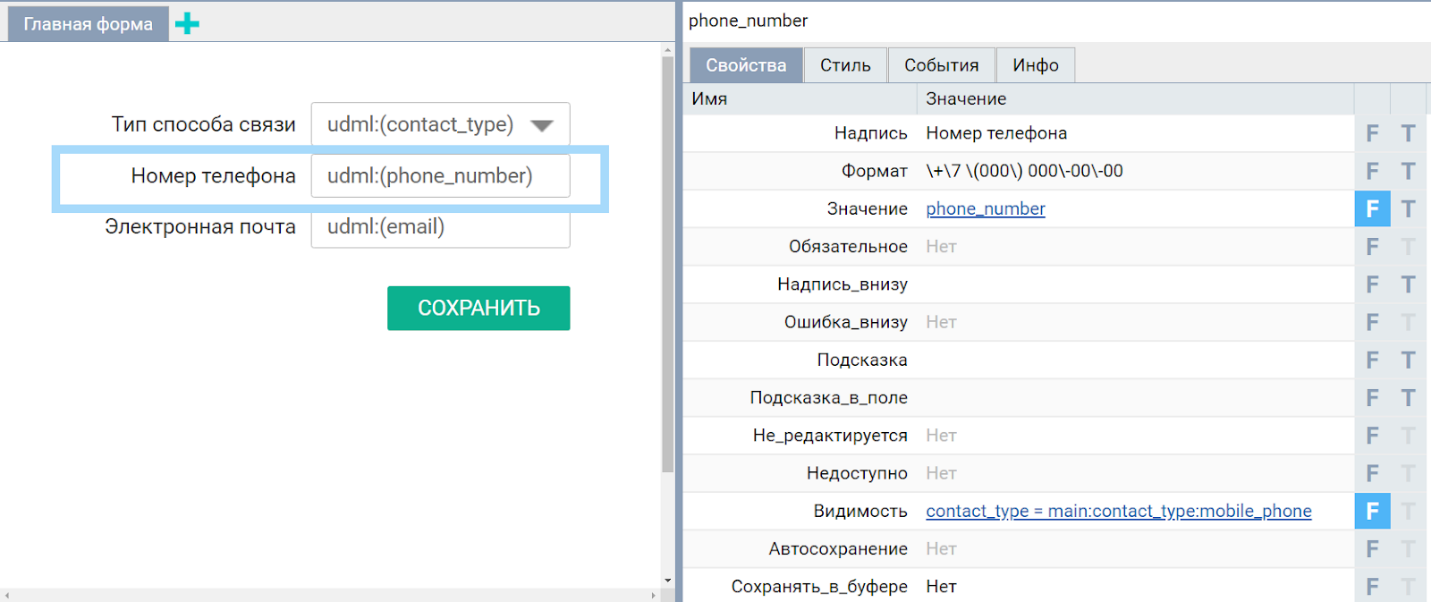
Свойства списка **contact\_type**:

* **Значение** — UDML-выражение: **contact\_type**;
* **Источник значений** — UDML-выражение: main:**contact\_type**:**active**() .



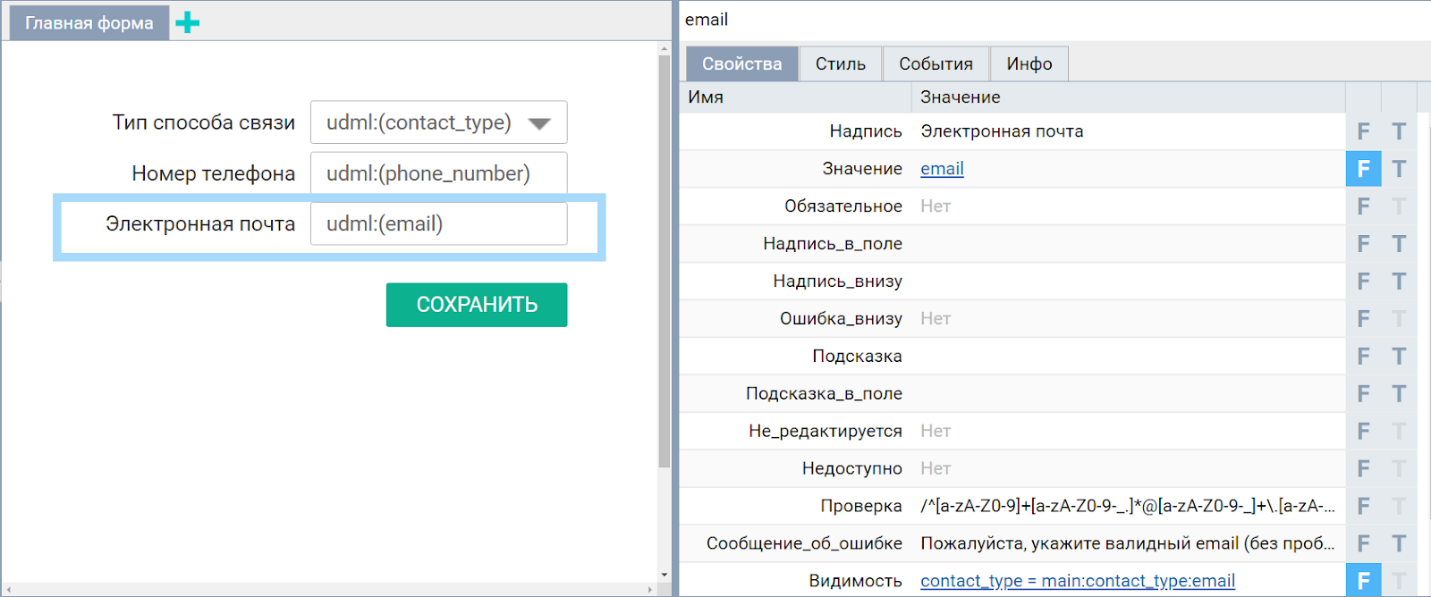
Свойства виджета **phone\_number**:

* **Значение** — UDML-выражение: **phone\_number**;
* **Видимость** — UDML-выражение: contact\_type = main:**contact\_type**:mobile\_phone

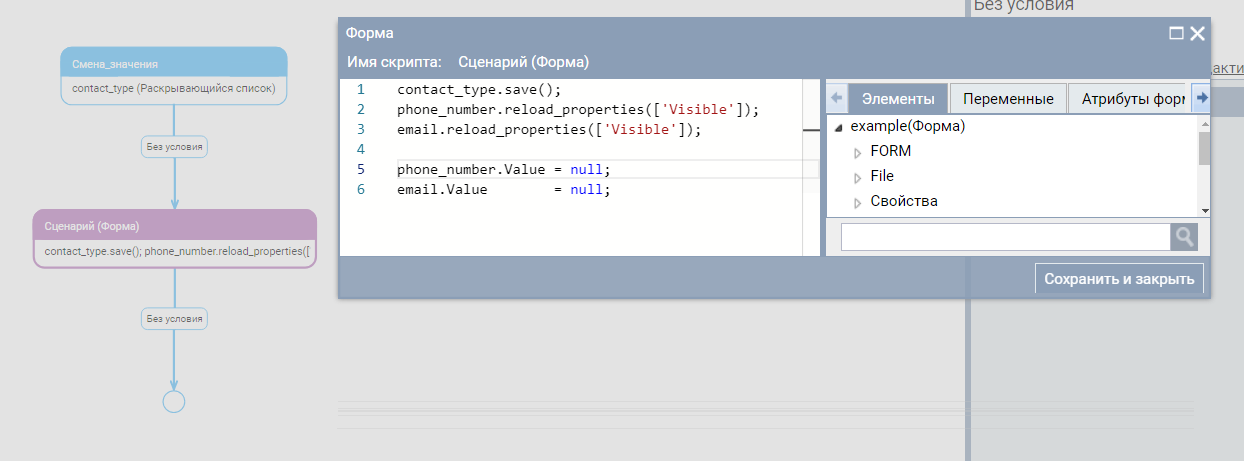


Свойства виджета **email**:

* **Значение** — UDML-выражение: **email**;
* **Видимость** — UDML-выражение: contact\_type = main:**contact\_type**:email



На диаграмме формы для виджета **contact\_type** настроен обработчик события **Смена\_значения**, в котором исполняется следующий сценарий формы:



Т.е. при смене типа способа связи сохраняется значение из списка, обновляется видимость полей для ввода контакта и сами значения полей очищаются.

### 

### **Действия пользователя**

Предположим, пользователь выполняет следующие действия:

1. В раскрывающемся списке **contact\_type** выбирает **Мобильный телефон** (т.е. main:**contact\_type**:mobile\_phone).
2. В поле **phone\_number** вводит номер телефона **+7 (923) 123-45-67.**
3. Нажимает на кнопку **Сохранить**.

### 

### **Исполнение сценария формы**

Разберём как исполняется сценарий формы:

1. В первой строке исполняется метод:

contact\_type.save();

Вызов метода **save()** или **force\_save()** для виджета генерирует UDML-выражение, в котором указанное значение виджета в присваивается атрибуту формы, указанному в свойстве **Значение**:

|  |
| --- |
| <Атрибут, указанный в свойстве **Значение**> := <указанное значение в виджете> |

В описанном выше кейсе будет сгенерировано выражение рода:

|  |
| --- |
| contact\_type := main:**contact\_type**:mobile\_phone |

2. Далее идут вызовы методов, которые перезагружают свойства виджетов:

phone\_number.reload\_properties(['Visible']);

email.reload\_properties(['Visible']);

Вызовы методов **reload()** или **reload\_properties()** генерируют те же самые UDML-выражения, что указаны в свойствах виджета. В кейсе будут сгенерированы выражения рода:

|  |
| --- |
| contact\_type = main:**contact\_type**:mobile\_phone |

и

|  |
| --- |
| contact\_type = main:**contact\_type**:email |

3. И в самом конце исполняются строки, которые очищают значения полей контактов:

phone\_number.Value = null;

email.Value = null;

Однако, их исполнение не приведет к генерации UDML-выражения, т.к. они исполняются только в клиентской части выражения, и значения атрибутов форм, указанных в свойстве **Значение** у этих виджетов, останутся неизменными

По завершении сценария формы все сгенерированные UDML-выражения будут отправлены в очередь для дальнейшего расчета на сервере. И только после окончания этого расчета обработчик завершит свою работу.

### 

### **Сохранение формы**

При нажатии на кнопку, у которой включено свойство **Сохранить\_форму**, или вызове методов **save()** для формы, или при вызове метода **FORM.save()**,генерируются UDML-выражения, присваивающие значения атрибутам формы, как и при вызове **save()** или **force\_save()** для одного виджета. Но при сохранении всей формы генерируется набор UDML-выражений для всех редактируемых виджетов на форме, например:

|  |
| --- |
| contact\_type := main:**contact\_type**:mobile\_phone phone\_number := ‘+7 (923) 123-45-67’ email := **udmlnull**() |

### 

### **Сообщения в очереди**

Увидеть сообщения, отправленные в очередь, можно при помощи [профилирования в UDM-консоли](https://doc.caseplatform.tech/#0.0.0:0:0:user_guide/debugging_application/udm_console/udm_console_profiling.html:0:ru) или при помощи [инструментов разработчиков браузера](https://developer.chrome.com/docs/devtools/network/).

#### ****Профилирование в UDM-консоли****

**Профилирование** — это процедура, позволяющая измерить время исполнения операции. На основе этой информации можно выделить проблемные места в исходном коде приложения.

Чтобы выполнить профилирование:

1. Убедитесь, что вам доступна работа с UDM-консолью. Для этого в свойствах приложения должен быть проставлен флажок **Запустить приложение в ‘Режиме отладки’**.



2. В FormRunner откройте UDM-консоль с помощью комбинации клавиш **CTRL+Q**. Консоль состоит из вкладок со следующим назначением:

* **Консоль** — тестирование UDML-выражений в рамках контекста формы.
* **UDML функции** — получение справочной информации о системных UDML-функциях.
* **Структура формы** — просмотр и отладка свойств формы и ее виджетов.
* **Профилирование** — измерение времени исполнения UDML-выражений.

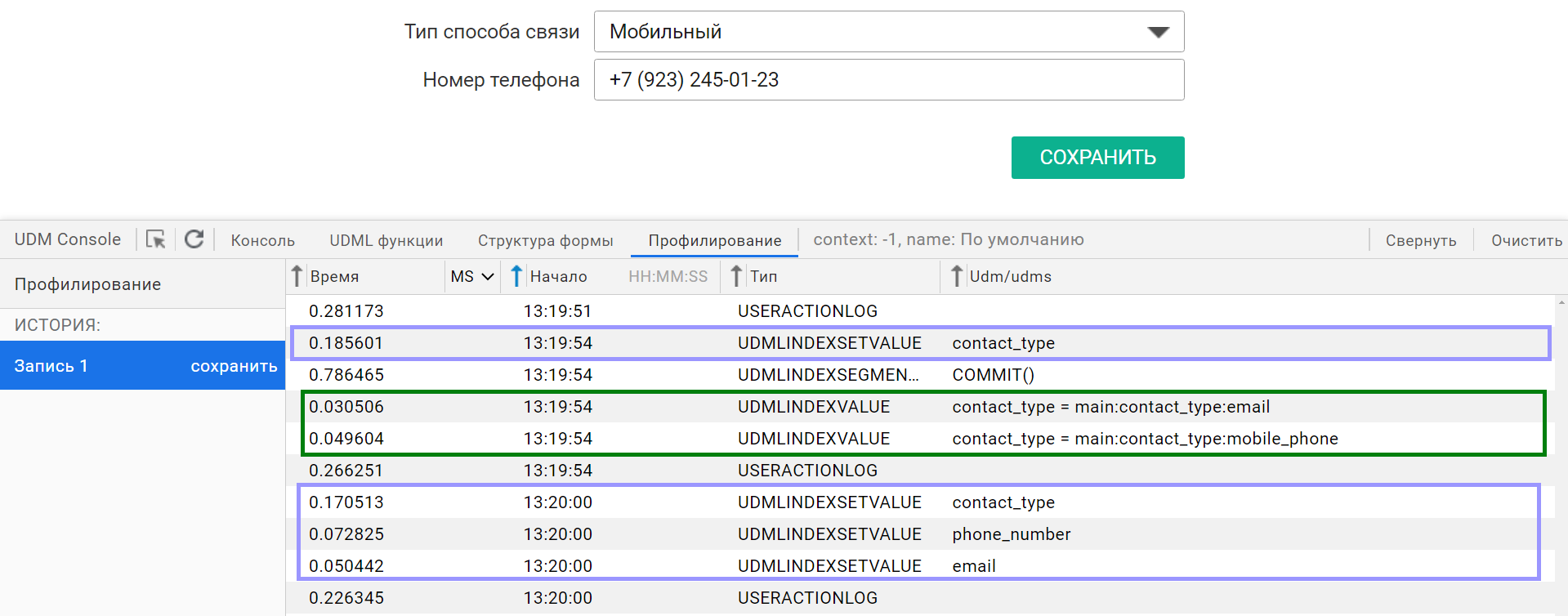
3. Перейдите на вкладку **Профилирование**.

4. Нажмите **Начать профилирование**.

5. Выполните действия, чью длительность выполнения необходимо измерить (например, нажать на кнопку, выбрать запись из таблицы и т.д.).

6. После выполнения всей цепочки действий завершите профилирование, нажав **Остановить профилирование**.

После выполнения профилирования описанного выше кейса мы можем увидеть запись со всеми выполненными UDML-выражениями. В этой таблице отображается длительность исполнения UDML-выражения, дата-время начала его расчета, тип UDML-выражения и само выражение.



На рисунке выше фиолетовой рамкой выделены выражения с типом **UDMLINDEXSETVALUE**. Эти выражения генерируются при вызовах методов типа **save()**, и они отвечают за присвоение значений атрибутам форм.  Например, первое выражение было сгенерировано при смене значения типа способа связи, а последние 3 – при нажатии на кнопку **Сохранить**.

Зеленой рамкой выделены выражения с типом **UDMLINDEXVALUE**. Они генерируются при вызовах метода типа **reload()**, а также перед загрузкой самой формы. Эти выражения отвечают только за получение значений UDML-выражений. Например, эти 2 выражения были рассчитаны в рамках перезагрузки свойства **Видимость** у полей ввода почты и мобильного телефона.

#### ****Инструменты разработчика браузера****

Для отладки клиентской части приложения чаще всего пользуются  **Dev Tools (инструментами разработчика) браузера**.Их функциональность отличается в зависимости от браузера, но большинство включает в себя такие инструменты как:

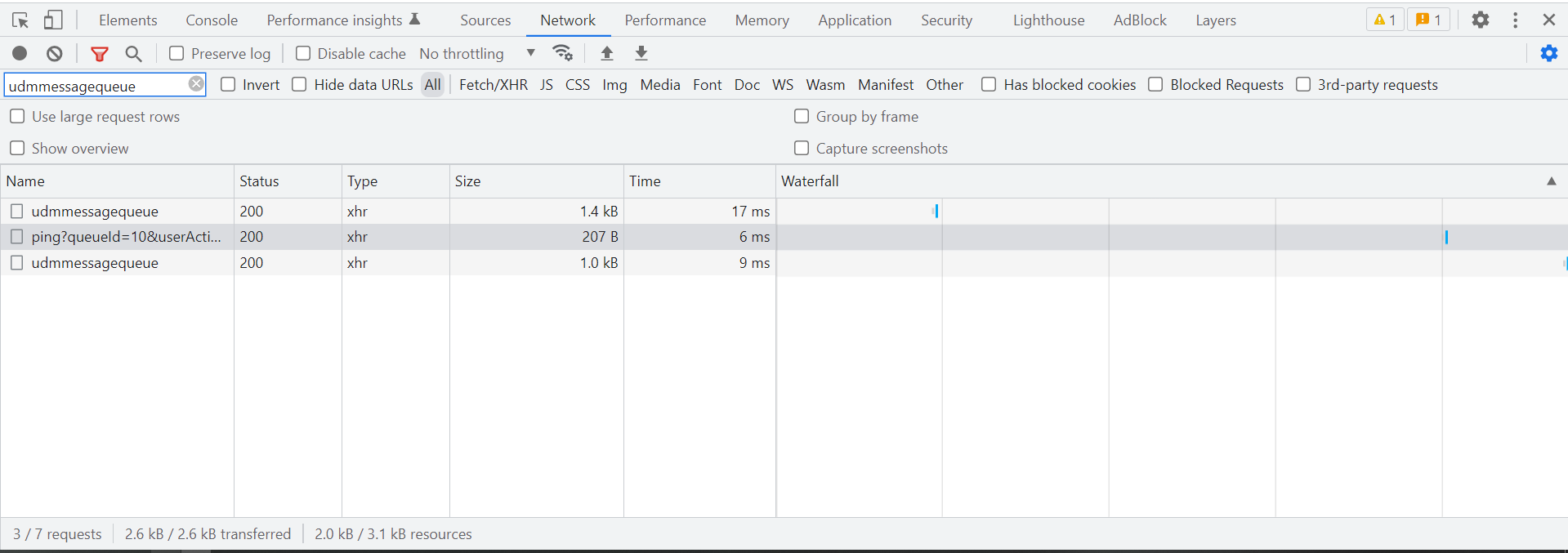
* **Elements** – изучение структуры DOM и связанных CSS классов.
* **Console** – отслеживание логов, а также исполнение JS кода.
* **Source** – отладчик исполняемых JS файлов.
* **Network** – отслеживание отправляемых запросов.

Чтобы открыть инструменты браузера в Google Chrome / Mozilla Firefox, нажмите на клавишу **F12**.

В рамках этого урока мы рассмотрим вкладку **Network**. После её открытия мы также повторим действия из кейса, а затем остановим запись, нажав на красную кнопку **Stop Recording Network Log** в левом верхнем углу.

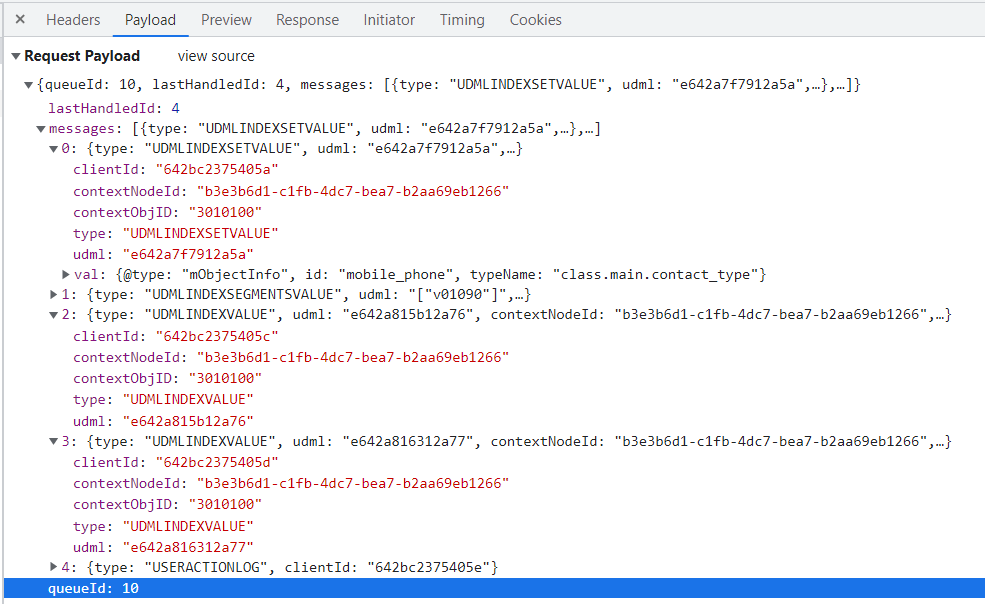
Каждая строка в **Network Log** – это ресурс, на который был отправлен запрос. По умолчанию ресурсы перечислены в хронологическом порядке. Каждый столбец хранит в себе информацию о ресурсе: название ресурса, код ответа на запрос, размер запроса и т.д.

В поле фильтрации введем **udmmessagequeue**, чтобы увидеть все отправленные запросы в очередь сообщений

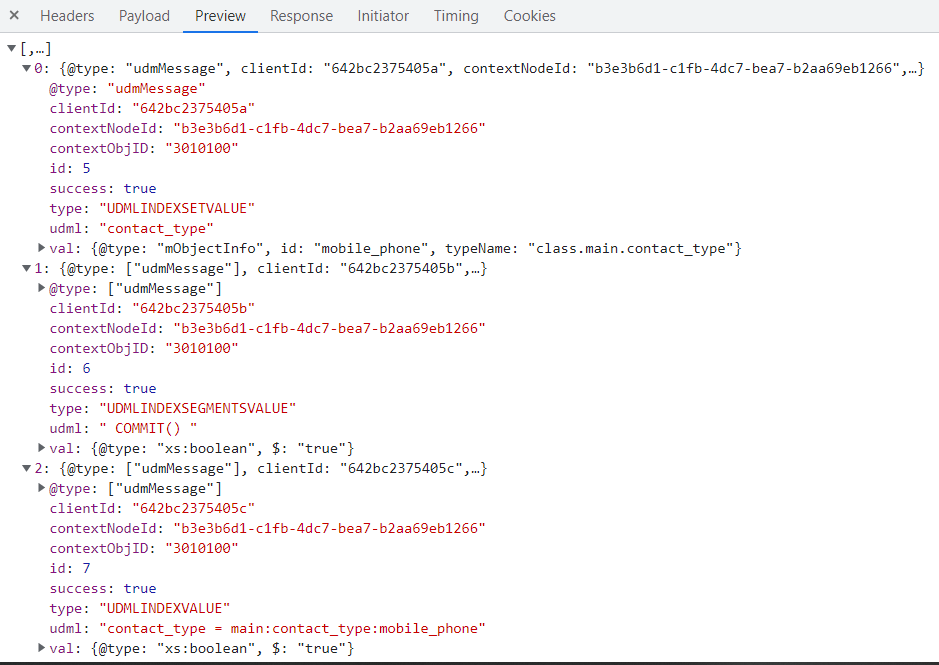


Мы можем увидеть 2 запроса. Выберем самый первый и перейдем в раздел **Payload** в правой панели.

В нем мы можем увидеть объект с массивом **messages**, которое хранит в себе UDML-выражения для расчета. Их тип идентичен тем же типам выражений, что мы видели в нотификации.



А в разделе **Response** мы можем увидеть результаты расчетов этих выражений: в рамках первого запроса были отправлены UDML-выражения, подготовленные при исполнении сценария формы, т.е. сохранение раскрывающего списка с типом контактов и обновление видимости полей.



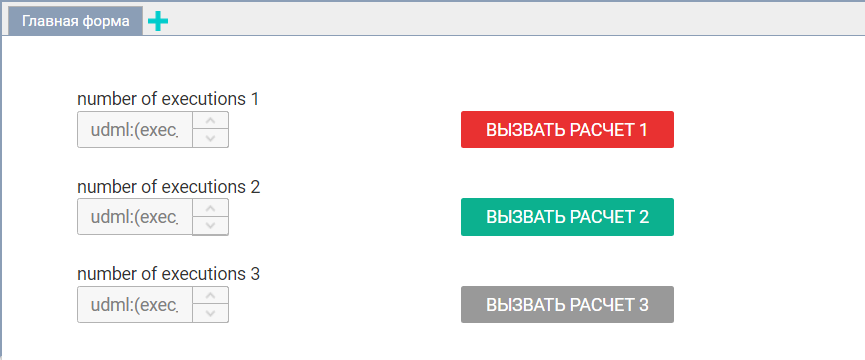
### 

## **Ожидание исполнения выражений**

Будьте аккуратны при настройке логики на диаграмме формы, чтобы исключить ситуации, подобные тем, когда пользователь нажал на кнопку, и пока выполняется долгая операция, у него остаётся возможность нажимать другие кнопки, которые будут формировать запросы к серверу.

В этом случае у пользователя будет ощущение, что ничего не происходит: он нажал кнопку, затем еще 25 кнопок, но ничего не происходит! 😮 Однако в это время в очередь отправляется много запросов, они накапливаются, и после исполнения первого запроса, который привел к небольшому запаздыванию, на сервер отправляется большая пачка других запросов, что может снизить производительность сервера.

Рассмотрим пример такого кейса. На рисунке ниже представлена форма с 3 числовыми полями и 3 кнопками, которые запускают обработчики диаграмм.



В рамках этих обработчиков исполняется сценарий сервера, где значение атрибута формы увеличивается на единицу, а затем числовое поле ввода перезагружается.

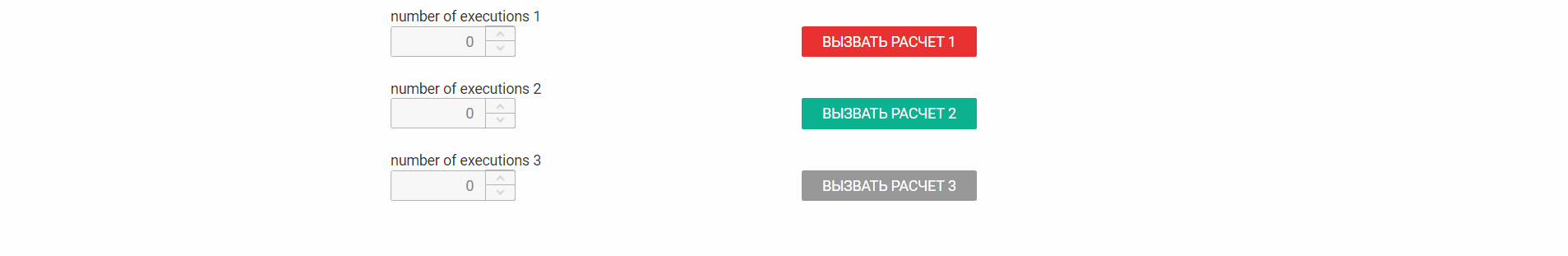
В сценарий сервера обработчика события нажатия на кнопку **Вызвать расчет 1** мы добавим задержку при помощи метода **sleep()** на 3 секунды, чтобы имитировать исполнение долгой операции:

Packages.java.lang.Thread.sleep(1000 \* 3);

udm:(exec\_number := exec\_number + 1);

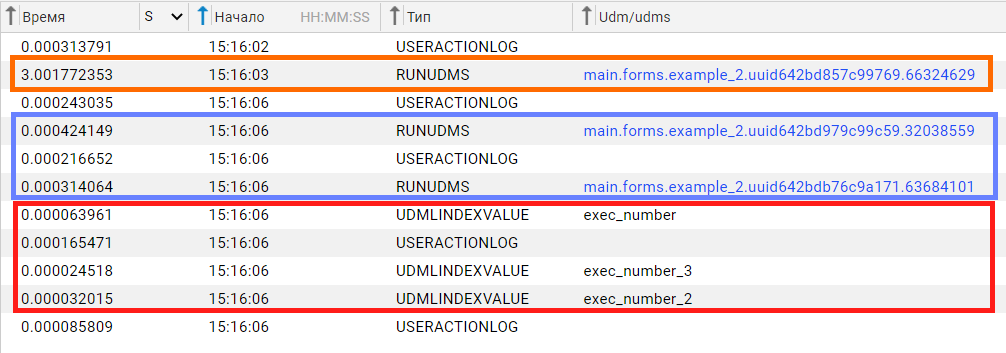
Далее мы будем нажимать на эти кнопки по порядку, т.е. сначала нажмем на кнопку с задержкой **Вызвать расчет 1**, а затем – на кнопку **Вызвать расчет 2,** и потом – на кнопку **Вызвать расчет 3.**

На рисунке ниже представлено то, как отработало приложение.



Можно увидеть, что после нажатия на первую кнопку обработчики событий других кнопок не были запущены, и только по истечении 3 секунд они были выполнены.

На рисунке ниже представлена запись исполненных UDML-выражений в UDM-консоли.



Здесь можно увидеть, в каком именно порядке были исполнены выражения. Оранжевой рамкой выделен первый отправленный запрос на исполнение сценария сервера, на расчет которого ушло 3 секунды. Затем синей рамкой выделены запросы для расчета сценариев сервера из обработчиков событий 2 других кнопок. И только потом были выполнены запросы на перерасчет значений в числовых полях ввода (красная рамка).

Поэтому рекомендуется при проектировании взаимодействия учитывать эту особенность работы с очередями, и при необходимости добавлять блокировку формы при помощи метода [FORM.lock()](https://doc.caseplatform.tech/#0.0.0:0:0:user_guide/form_builder/form_diagram/form_diagram_elements/form_diagram_methods_list.html:form-diagram-methods-list:ru), чтобы у пользователя отсутствовала возможность добавлять множество сообщений в очередь.

## 

## **Коллбэки для методов save() и reload()**

Если вы попытаетесь выполнить в сценарии формы код с использованием асинхронных методов, то он будет работать с тем состоянием, которое было при отправке сообщений в очередь, а не с тем состоянием, которое получается после запроса на обработку UDM и получение данных.

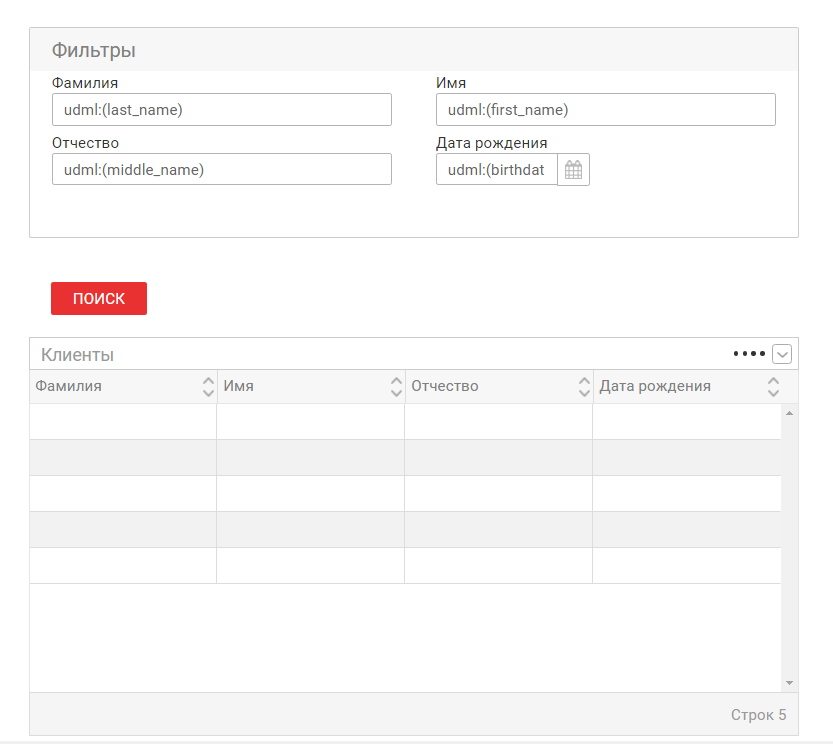
Напомним, что асинхронными методами в Case Platform являются: **Перезагрузить()**, **Перезагрузить\_свойства(),** **Отрисовать(), Очистить(),** **Сохранить()** и **Сохранить\_Немедленно()**. Также асинхронным методом будет **Показать()**, если его используем для виджета Всплывающее окно.

Для обеспечения правильной работы в сценарии формы эти методы принимают в качестве входного аргумента функцию, т.е. **коллбэк** (callback, функция обратного вызова). Когда очередь возвращает результаты вычисления UDML-выражений, она запустит указанный коллбэк.

Если нужно, чтобы в сценарии формы какая-либо функция была выполнена после сохранения или после перезагрузки— укажите эту функцию в методе. Понимание этой логики позволяет решить огромное количество проблем, связанных со сценариями форм.

#### ****Пример****

Представим пример настройки коллбэка для метода **reload()**. На рисунке ниже представлена форма для поиска клиентов по заданным параметрам в фильтрах.



При нажатии на кнопку **Поиск** запускается сценарий формы, в котором сохраняются поля для поиска клиента, а затем перезагружается таблица. После перезагрузки нужно подсчитать количество найденных клиентов, если их количество равно 0, то нужно отобразить подсказку, что клиенты не были найдены.

filters.save();

clients.reload();

if (clients.rowCount() == 0) {

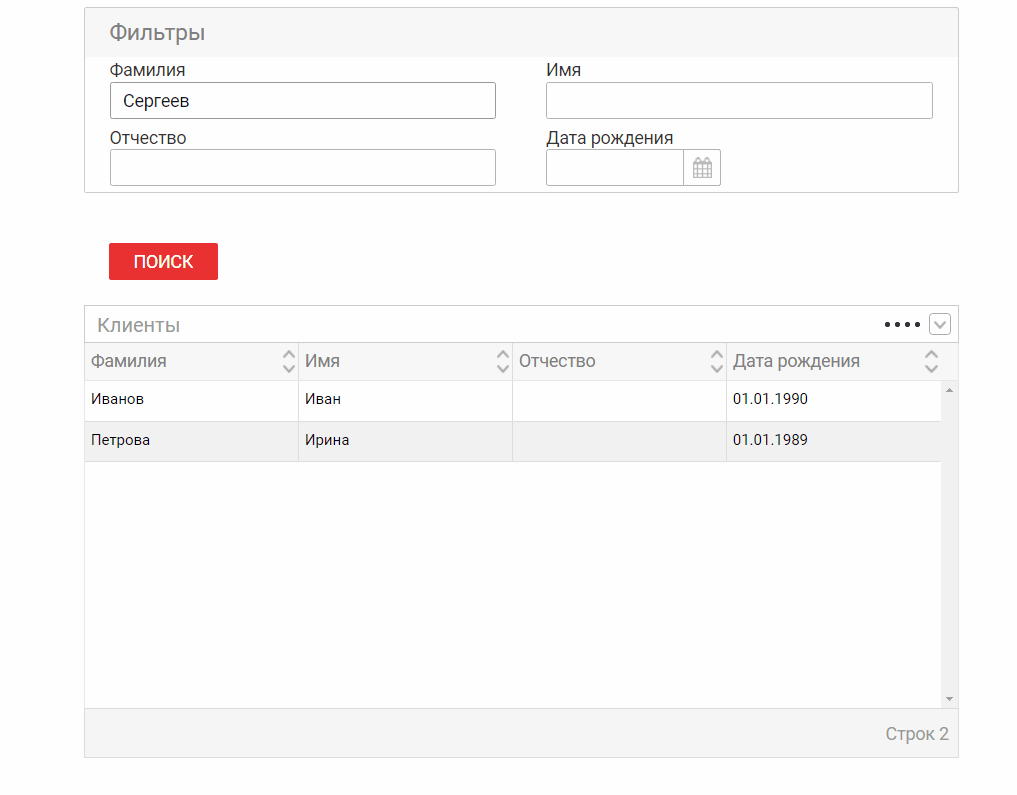
  clients.show\_balloon('Клиенты не найдены по параметрам');

} else {

  clients.hide\_balloon();

}

В связи с тем, что методы **save()** и **reload()** являются асинхронными, то количество строк будет подсчитано некорректно, и метод **rowCount()** вернет количество строк, актуальное на момент генерации UDML-выражения. Поэтому после запуска поиска с такими параметрами, по которым нет клиентов, подсказка отображаться не будет.



Поэтому мы вынесем проверку в качестве коллбэка метода перезагрузки таблицы:

filters.save();

clients.reload(function(){

  if (clients.rowCount() == 0) {

      clients.show\_balloon('Клиенты не найдены по параметрам');

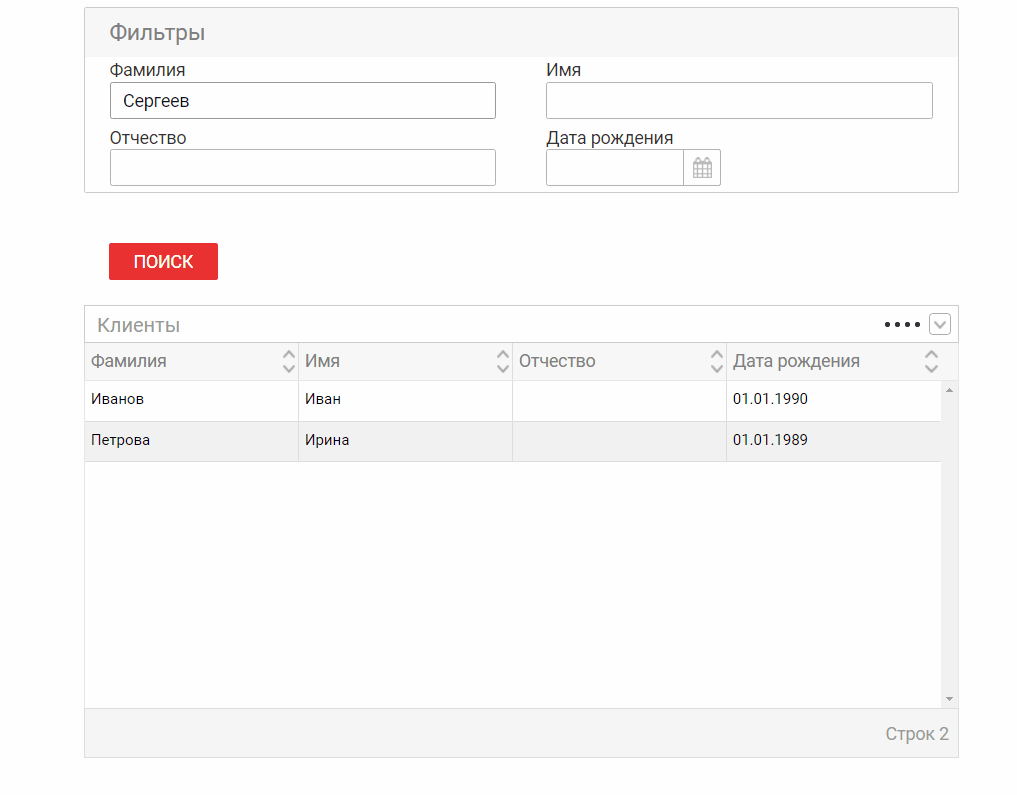
  } else {

      clients.hide\_balloon();

  }

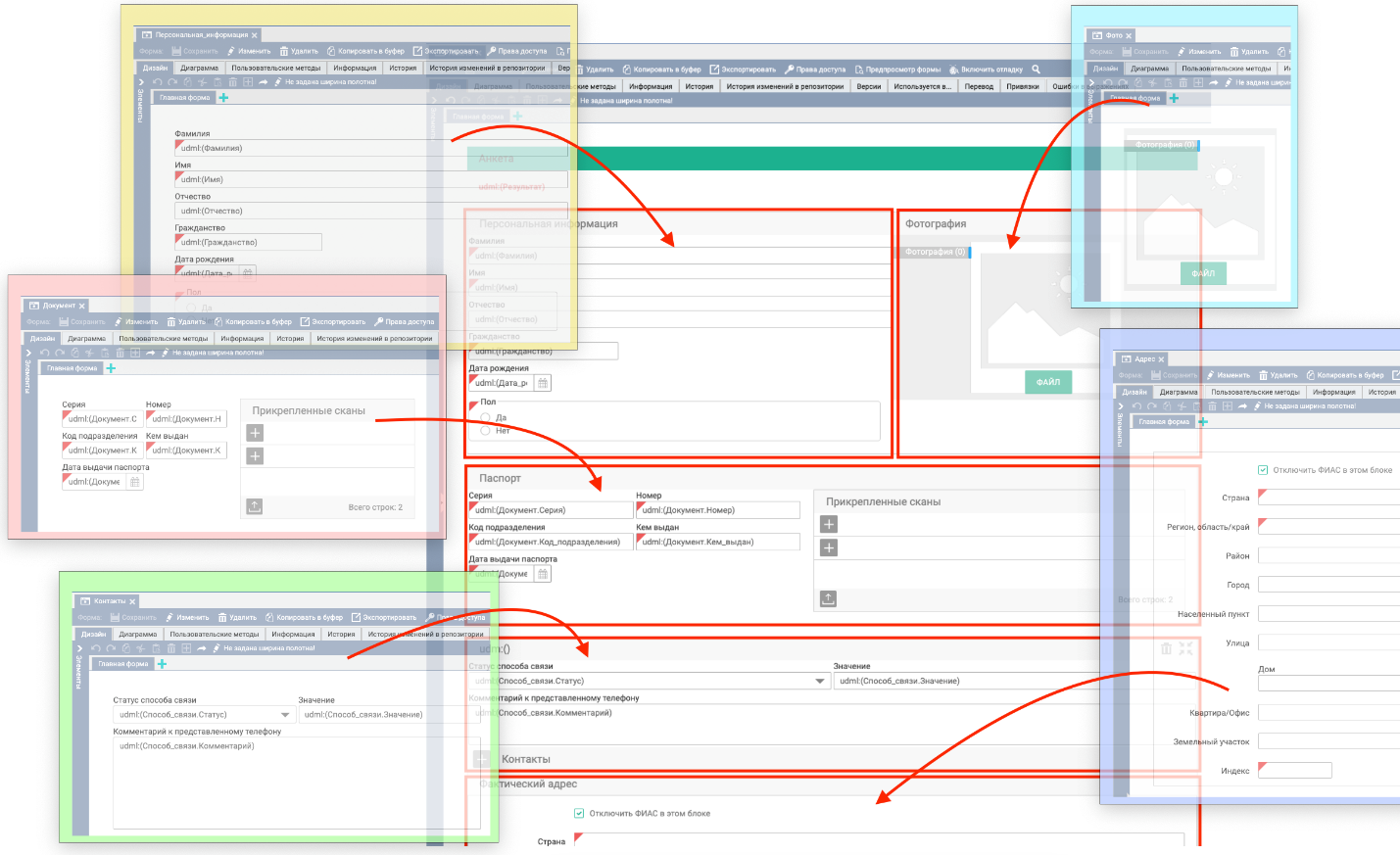
});

С таким сценарием проверка будет проводится уже с данными, актуализированными в результате расчета UDML-выражения.



**Диаграмма вложенных форм**

Как вы знаете из предыдущего курса 103 про формы — частая практика выделять функциональные формы и далее использовать их как компоненты на других (более общих) формах.

форма Анкета, состоящая из компонент других форм

Будем называть общую форму — *родительской*, а компоненты — *дочерними* формами.

**Как же будет устроена диаграмма такой формы?**

❗️ Вся логика, необходимая на виджете, должна быть настроена там, где этот виджет изначально расположен.

Нельзя напрямую с родительской формы обращаться к виджетам и событиям этих виджетов дочерней формы. Рекомендуется воспринимать [Произвольный компонент](https://doc.caseplatform.tech/#1.3.03:0:0:user_guide/form_builder/form_design/form_widgets/extras_form_widgets/embedded_component_widget.html:0:en), [Абстрактную форму](https://doc.caseplatform.tech/#1.3.03:0:0:user_guide/abstract_form_builder.html:0:en) и [Компонент](https://doc.caseplatform.tech/#1.3.03:0:0:user_guide/form_builder/form_design/form_widgets/nested_form_widget.html:0:en) (дочернюю форму) как цельный виджет с набором ее событий и методов. Это связано с тем, что формы создаются с галкой Строгий контекст и представляют из себя инкапсулированный компонент. Вся логика работы с элементами формы по возможности должна быть выстроена на этой же форме.

*Произвольный компонент и Абстрактная форма — это виджеты формы, подробнее о них можно самостоятельно почитать в документации. В данном курсе они рассматриваться не будут.*

Рассмотрим пример:

Есть форма **personal\_data** с виджетом Дата для ввода даты рождения клиента. Есть форма **client\_form** с компонентой формы **personal\_data**. Необходимо настроить проверку на совершеннолетие, то есть настроить событие Смена\_значения для виджета Даты. Данное событие необходимо настраивать на форме **personal\_data**, то есть там, где этот виджет изначально был расположен.

**Решение 1 (плохое):** настроить событие для виджета Дата на форме **client\_form**

|  |  |
| --- | --- |
|  |  |

Код в сценарии формы:

|  |
| --- |
| **var** now = **new** Date(); **var** date\_control = **new** Date(now.getFullYear() - 18, now.getMonth(), now.getDate()); **var** db = **personal\_data\_1**.birthday.Value;  db = **new** Date(db.getFullYear(), db.getMonth(), db.getDate()); **if** (db > date\_control) { *//Если клиенту меньше 18 лет* **personal\_data\_1**.birthday.show\_balloon('Ошибка ввода даты рождения', 'error', **true**); } **else** { **personal\_data\_1**.birthday.hide\_balloon('Ошибка ввода даты рождения', 'error'); } |

**Решение 2 (хорошее):** настроить событие для виджета Дата на форме **personal\_data**

|  |  |
| --- | --- |
|  |  |

Код в сценарии формы:

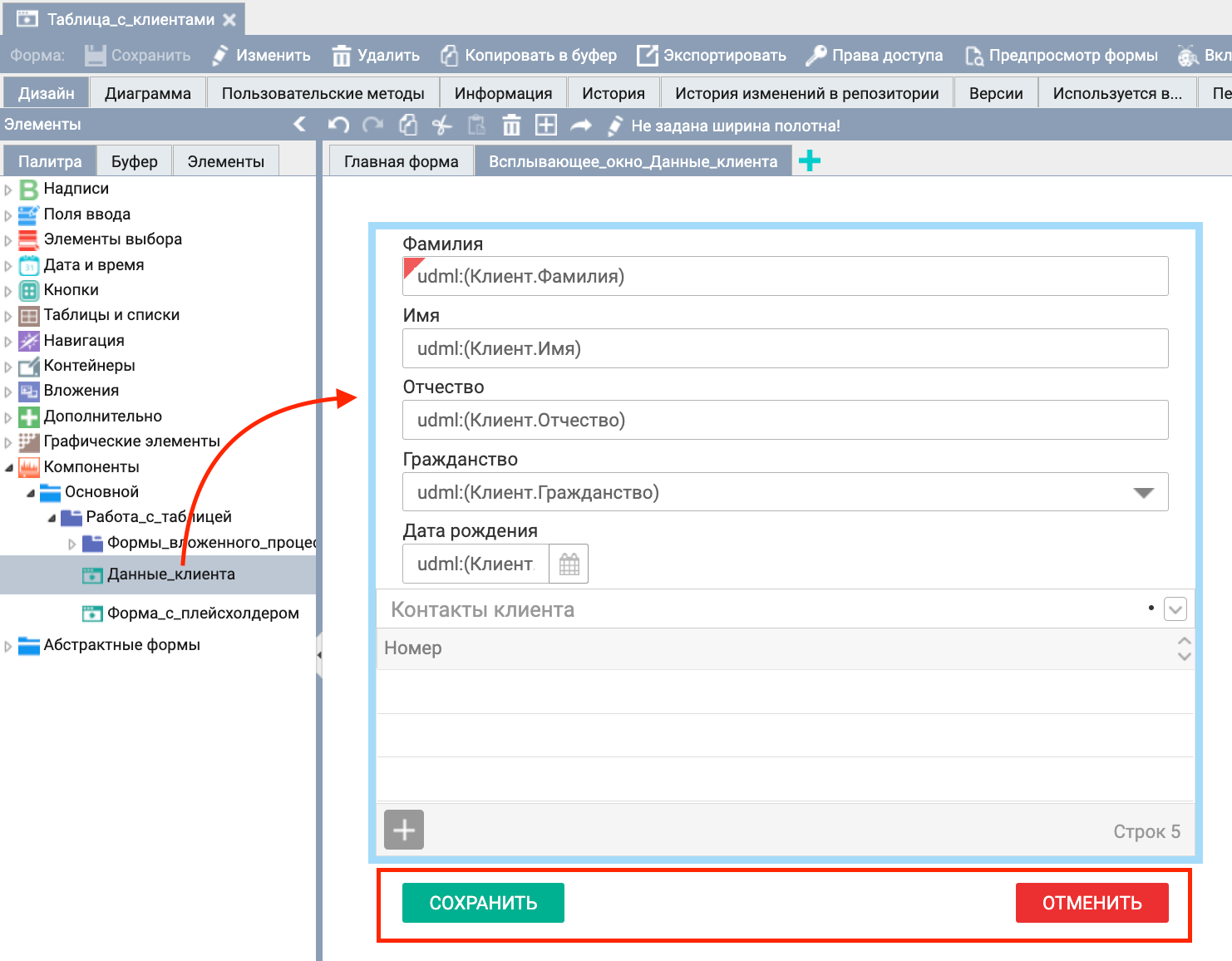
|  |
| --- |
| **var** now = **new** Date(); **var** date\_control = **new** Date(now.getFullYear() - 18, now.getMonth(), now.getDate()); **var** db = birthday.Value;  db = **new** Date(db.getFullYear(), db.getMonth(), db.getDate()); **if** (db > date\_control) { *//Если клиенту меньше 18 лет* birthday.show\_balloon('Ошибка ввода даты рождения', 'error', **true**); } **else** { birthday.hide\_balloon('Ошибка ввода даты рождения', 'error'); } |

☝️ Родительская и вложенная формы могут общаться также с помощью [Пользовательских методов](https://doc.caseplatform.tech/#1.3.03:0:0:user_guide/form_builder/form_custom_methods.html:0:en) и [Пользовательских событий](https://doc.caseplatform.tech/#1.3.03:0:0:user_guide/form_builder/form_diagram/form_diagram_events/user_event_example.html:0:en). Более того рекомендуется использовать преимущественно именно эти варианты настройки. Но в данном курсе мы их рассматривать не будем, а то будет сильно сложно 🤪

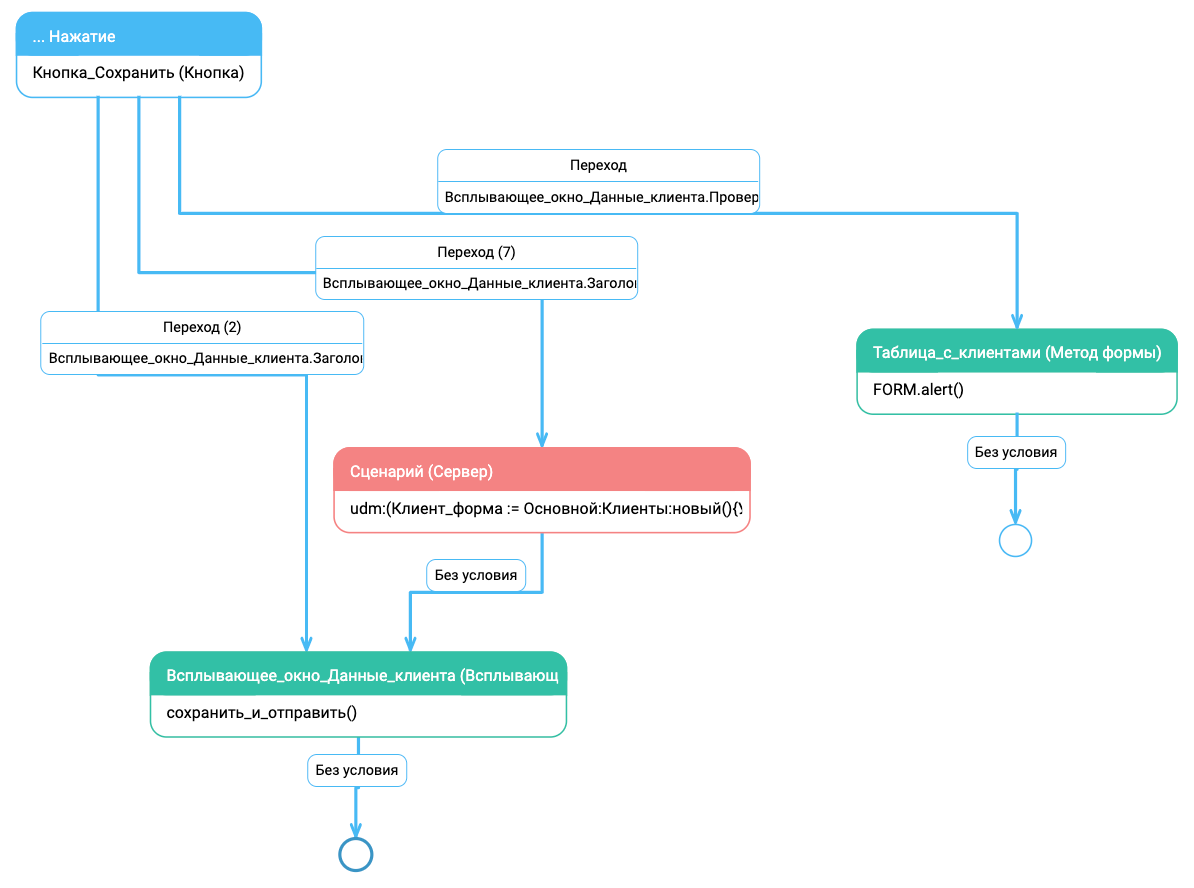
**Кнопки**

Рекомендуется кнопки настраивать на родительской форме, даже если они “управляют” метаданными на дочерней форме. Это связано с тем, что компонента должна быть универсальной и возможной для переиспользования в разных интерфейсных формах.

Например, есть главная форма Таблица\_с\_клиентами со всплывающем окном, на всплывающем окне расположена компонента Данные\_клиента с набором виджетов. Кнопки Сохранить и Отменить расположены на главной форме, на полотне всплывающего окна. Такая реализация позволит использовать компоненту формы Данные\_клиента на разных других формах.



При этом нажатие на кнопку обрабатывается здесь же, на главной форме. Пример диаграммы для данного кейса:



В зависимости от заголовка всплывающего окна и заполненных данных в компоненте:

1. просто сохраняем и закрываем форму;
2. создаем новый объект Клиент в базе данных, сохраняем и закрываем форму;
3. показываем предупреждение пользователю, что не все данные заполнены.