Установка

1. Определите рабочий каталог, где будут расположены ваши библиотеки:
    a. Для серверных редакций — в рабочем каталоге Loginom Server (в папке пользователя или в общей папке пользователей);
    b. Для настольных редакций — в любой папке на локальном диске.
2. Создайте в нем подкаталог **libs**.
3. Разместите папку **JsonParse v1.2** в каталоге **libs**.
4. Добавьте ссылку на пакет **ParseJSON_component.lgp** в своем пакете и используйте компоненты библиотеки.

Описание компонентов

Реализовано два вида компонента:

- **parseJSON variable** - компонент принимает один JSON в переменную;
- **parseJSON table** - компонент принимает набор данных с JSON;

Компонент работает в версиях Loginom 6.5.4 и выше.

Входные порты

-  Входной JSON (таблица для **parseJSON table** или

   порт переменных для **parseJSON variable**)

-  Настройки

Входной JSON

Имя поля	Метка	Комментарий
ab **JSON**	JSON	Поле принимает JSON для парсинга

Настройки

Имя переменной	Метка	Комментарий
0/1 **DuplicationUnitValues**	Дублировать единичные значения	Дублирование родительских узлов и узлов-братьев. Опция имеет приоритет перед опцией "Дублировать значения родительских узлов" (дополнительно см. раздел "Реализация дублирования значений узлов").
0/1 **DuplicationParentNodeValues**	Дублировать значения родительских узлов	Дублирование значений родительских узлов так же происходит при включении флага "Дублировать единичные значения". Опция имеет меньший приоритет перед опцией "Дублировать единичные значения". (дополнительно см. раздел "Реализация дублирования значений узлов")
0/1	Генерировать	при наличии данного флага в каждой метке

GenerateCompoundFieldLabels	составные метки полей	поля будет отражена иерархия относительно корневого элемента.
0/1 **GenerateVariableTypeField**	Генерировать поля переменного типа данных	Все поля результирующей таблицы генерируются с типом данных "Переменный".
ab **Path**	Выражение jmespath	Текст запроса, применяемый к исходному JSON перед трансформацией в таблицу. (см. раздел "Запросы к json")

Выходные порты

-  Выходной набор
-  Ошибки

Выходной набор

Состав полей в порту формируется динамически (используется автосинхронизация полей) и зависит от структуры JSON. Набор данных содержит обязательное поле Error (Исключение), исключение выводится в строке с проблемным элементом JSON.

Ошибки

В переменную Error (Исключение) выводится текст перехваченной ошибки в ходе трансформации данных.

Реализация дублирования значений узлов

Рассмотрим ситуацию:

объект **Root** имеет атрибуты и дочерние узлы **O**(объект) и **M**(массив). Вопрос - должны ли при выводе значений массива **M** дублироваться не только атрибуты **Root**, но и атрибуты дочернего к **Root** объекта **O**?

В зависимости от ответа имеются два различных варианта реализации. Оба варианта реализованы:

- Поведение, реализованное в "Дерево в таблицу", предусматривает дублирование атрибутов объекта **O** и всех его дочерних объектов. Такое поведение включается при выборе настройки *Дублировать единичные значения*.
- Атрибуты объекта **O** и всех его дочерних объектов не будут дублироваться при выборе настройки *Дублировать значения родительских узлов*.

Обработка и вывод ошибок

Практически вся логика обработки заключена в блок `try... catch...`. Ошибки, возникающие в процессе обработки, выводятся в выходное поле/переменную `Исключение`.

Запросы к json

В компоненте возможна предварительная модификация исходного json с помощью запроса к json (https://jmespath.org/tutorial.html). Реализующий данный функционал библиотека (https://github.com/jmespath/jmespath.js , https://github.com/daz-is/jmespath.js) подключается как внешний модуль. Если в параметре `Path` компонента задан текст запроса, то он будет применен к

исходному JSON, и уже результат запроса преобразуется в таблицу. Если запрос не задан (параметр запроса пустой), то в таблицу преобразовывается исходный JSON.

Запросы позволяют предопределить имена узлов в результирующем json. Пример: запрос `people[].{Name: name, State: state.name}`, примененный к json:

```
{
  "people": [
    {
      "name": "a",
      "state": {"name": "up"}
    },
    {
      "name": "b",
      "state": {"name": "down"}
    },
    {
      "name": "c",
      "state": {"name": "up"}
    }
  ]
}
```

определяет узлы с именами `Name` и `State` для вывода результатов запроса:

```
[
  {
    "Name": "a",
    "State": "up"
  },
  {
    "Name": "b",
    "State": "down"
  },
  {
    "Name": "c",
    "State": "up"
  }
]
```

Данный функционал можно использовать для предопределения структуры полей результирующей таблицы компонента.

Вывод значений с различным типом данных, вывод null-значений

В общем случае ячейки выводимые в одном столбце выходной таблицы могут иметь различные типы данных. Пример такого JSON:

```
[
  {"Key": 1},
  {"Key": "Строка"},
  {"Key": true},
  {"Key": null}
]
```

По аналогии с логикой преобразования "Дерево в таблицу" для данного примера должна сформироваться одна колонка выходной таблицы с идентификатором `root.Key`. Используя параметр "Генерировать поля переменного типа данных" (= `true` или `false`), пользователь может выбрать:

- вариант вывода значений разного типа в одной колонке переменного типа данных. Колонка будет иметь идентификатор `root.Key:object`:

Key
1
Строка
true
<null>

- вариант вывода всех значений в разных колонках с соответствующим типом данных. Колонки будут иметь идентификаторы `root.Key:namber`, `root.Key:string` и т.д.

Поскольку `null` значения нельзя отнести к определенному типу данных, то эти значения выводятся в поле с переменным типом данных. Однако, если в выходной таблице уже присутствует столбец для вывода ячеек с другим типом данных (как в приведенном примере), то отдельный столбец для вывода `null` сформирован не будет и результирующая таблица будет иметь вид:

Key	Key	Key
1		
	Строка	
		true
<null>	<null>	<null>

В последней (пустой) строке таблицы выводится объект `{"Key": null}`.

Для примера:

```
[
  {"Key": null},
  {"Key": null}
]
```

сформируется таблица:

Key
<null>
<null>

Именование полей

Исходя из того, что:

- внутренние идентификаторы полей (пример: `root.Key:namber`, `root.Key:string`) потенциально могут содержать запрещенные символы, поэтому они не могут использоваться в качестве имен динамически созданных полей;
- нумерация динамически сгенерированных полей (типа COL1, COL2 и т.д. как это делается в компоненте "Кросс-таблица") может зависеть не только от структуры JSON, но и от данных, поэтому не гарантирует неизменность маппинга с последующими узлами.

в качестве имени поля решено использовать результат хеш-функции от строки идентификатора поля. Пример: из идентификатора `root.[powers].g:number` в результате применения хеш-функции будет сформировано имя поля `C_1702078878`. Данное имя будет стабильным, независимым от данных, наполняющих json. Добавление или удаление соседних узлов и потомков для `root.[powers].g` так же не повлияет на сгенерированное имя.

Формирования меток полей

Логика формирования меток полей несколько отличается от логики в "Дерево в таблицу". Это связано с тем, что в JSON могут присутствовать не именованные массивы, например, если массив находится внутри массива, то внутренний массив не имеет KEY-идентификатора. Не именованным массивам присваивается идентификатор `[Array]`. В обычных (не составных) метках для таких массивов выводиться путь до ближайшего именованного родителя. Для удобства восприятия идентификаторы всех массивов заключены в `[]` квадратные скобки.

Вывод массивов

Логика вывода нескольких массивов в составе одного родительского объекта реализована так же как и в стандартном компоненте "Дерево в таблицу.