Стандартизация успешных ответов и ошибок в Metabot

Общий подход к обработке результатов между модулями, API и бизнес-логикой

В Metabot разные части системы постоянно обмениваются результатами выполнения операций:

- JS-плагины вызывают другие плагины;
- low-code сценарии запускают кастомный код;
- API-эндпоинты принимают запросы извне;
- внутренние модули возвращают данные друг другу.

Если каждый компонент будет возвращать ответы **в своём формате**, система очень быстро превращается в хаос:

- где-то `true / false`,
- где-то `throw error`,
- где-то `{ status: "ok" }`,
- где-то просто строка.

Чтобы этого избежать, в Metabot используется **единый стандарт ответов** — плагин `Common.Utils.Response`.

Зачем вообще стандартизировать ответы

Стандартизация решает сразу несколько задач:

- □□ единый формат для **успешных результатов и ошибок**;
- □□ предсказуемое взаимодействие между модулями;
- □□ упрощение логики: не нужно каждый раз гадать, что вернёт функция;
- □□ удобный проброс результатов во внешние API;
- □□ возможность централизованно логировать и анализировать ошибки.

Ключевая идея простая:

> **Любая функция либо возвращает success-response, либо error-response. Исключения — редкость, а не норма.**

Где используется `Common.Utils.Response`

Плагин применяется практически везде:

- в JS-плагинах (`Common.Helpers.*`, `Common.CJM.*`, AI-модули);
- во внутренних API-эндпоинтах;
- в бизнес-логике;
- в интеграциях с внешними системами;
- в helper-функциях, которые могут вызываться цепочкой.

Если вы пишете свой модуль — **используйте этот подход**, либо создайте свой аналог по той же схеме.

Базовая идея формата ответа

Ответ всегда — это объект.

Успешный результат

```
{
  success: true,
  ...data
}
```

Ошибка

```
{
  success: false,
  error: true,
  message: "Описание ошибки",
  error_code: "OPTIONAL_CODE"
}
```

Это минимальный контракт, который:

- легко проверять (`if (!result.success)`);
- удобно сериализовать;
- безопасно передавать между слоями системы.

Подключение плагина `Common.Utils.Response`

Перед использованием стандартных функций для возврата успешных ответов и ошибок плагин необходимо подключить в вашем JS-модуле.

Пример подключения:

```
// Подключаем Response плагин для обработки ответов
const { getErrorResponse, getSuccessResponse } = require('Common.Utils.Response');
```

После подключения функции `getSuccessResponse` и `getErrorResponse` доступны для использования в:

- helper-плагинах;
- бизнес-логике;
- API-эндпоинтах;
- кастомных скриптах внутри Metabot.

◫ Плагин `Common.Utils.Response` является общим и доступен на всех серверах Metabot, поэтому дополнительной установки не требуется.

Основные функции плагина

Плагин `Common.Utils.Response` предоставляет набор helper-функций.

`getSuccessResponse()`

Используется при успешном выполнении операции.

Примеры использования:

```
return getSuccessResponse("Операция выполнена успешно")
```

или

```
return getSuccessResponse({
  message: "Скрипт создан",
  script_id: 12345
})
```

`getErrorResponse()`

Используется для возврата ошибки **без выбрасывания исключения**.

```
return getErrorResponse("Не удалось отправить сообщение")
```

или

```
return getErrorResponse(
  { message: "Ошибка валидации", field: "email" },
  "VALIDATION_ERROR"
)
```

Почему это важно: ошибка становится **данными**, а не неконтролируемым исключением.

Дополнительные типы ответов

Плагин также содержит готовые шаблоны для типовых ситуаций:

- `getValidationErrorResponse()` — ошибки валидации;
- `getNotFoundResponse()` — сущность не найдена;
- `getForbiddenResponse()` — доступ запрещён.

Их удобно использовать в API и сервисных методах, чтобы не изобретать форматы заново.

Почему мы не кидаем exceptions везде

В классическом backend-подходе ошибки часто бросаются через `throw`. В Metabot это **не основной сценарий**, по нескольким причинам:

1. JS-код выполняется внутри движка (V8), а не в отдельном Node-процессе.
2. Исключения могут:
   - прервать выполнение сценария;
   - привести к неочевидным последствиям;
   - быть плохо читаемыми в логах.
3. Большинство ошибок — **ожидаемые**:
   - не найден скрипт,
   - пустой параметр,
   - недоступен внешний сервис,
   - неверный формат данных.

Поэтому:

- **ошибки возвращаются как результат**;
- `throw` используется только для действительно критических ситуаций.

Как это работает в цепочках вызовов

Типичный сценарий:

```
const result = sendFormattedMessage(text, 'HTML')

if (!result.success) {
  // можно залогировать
  // можно показать fallback
  // можно вернуть ошибку выше
  return result
}

// продолжаем логику
```

Таким образом:

- каждый уровень решает, что делать с ошибкой;
- ошибка может «подняться» наверх без искажения;
- формат ответа остаётся единым на всём пути.

Использование в API-эндпоинтах

Внутренние и внешние API Metabot обычно выглядят так:

```
try {
  // бизнес-логика
  return getSuccessResponse({ result })
} catch (e) {
  return getErrorResponse(e.message)
}
```

Снаружи клиент **всегда** получает предсказуемый JSON, независимо от источника ошибки.

Это критично для:

- фронтендов;
- интеграций;
- автоматических сценариев;
- AI-агентов, которые анализируют ответы.

Исходный код плагина Common.Utils.Response

```
/**
 * □□Плагин: Common.Response.Wrapper
 * Автор: @ArtemGarashko
 * Версия: 1.4
 * Дата последнего обновления: 26 апр 2025
 *
 * Назначение:
 * - Формирование ответов об успешных и неудачных операциях.
 * - Используется в API, между компонентами, в бизнес-логике.
 */

function getErrorResponse(errorOrData, errorCode = null) {
  const result = { success: false, error: true }

  if (typeof errorOrData === "string") {
    result.message = errorOrData
  } else if (typeof errorOrData === "object") {
    Object.assign(result, errorOrData)
  }

  if (errorCode) {
    result.error_code = errorCode
  }

  return result
}

function getSuccessResponse(dataOrMessage = {}) {
  const result = { success: true }

  if (typeof dataOrMessage === "string") {
    result.message = dataOrMessage
  } else if (typeof dataOrMessage === "object") {
    Object.assign(result, dataOrMessage)
```

```
  }

  return result
}

function getValidationErrorResponse(details) {
  return {
    success: false,
    error: true,
    message: details.message || "Validation failed",
    fields: details.fields || []
  }
}

function getNotFoundResponse(entity = "Entity") {
  return {
    success: false,
    error: true,
    message: `${entity} not found`
  }
}

function getForbiddenResponse(reason = "Access denied") {
  return {
    success: false,
    error: true,
    message: reason
  }
}

exports.getErrorResponse = getErrorResponse
exports.getSuccessResponse = getSuccessResponse
exports.getValidationErrorResponse = getValidationErrorResponse
exports.getNotFoundResponse = getNotFoundResponse
exports.getForbiddenResponse = getForbiddenResponse
```

> ⚠ Версия `Common.Utils.Response`, приведённая в статье, может отличаться от версии, установленной на сервере Metabot. Используйте код как ориентир

> и архитектурный шаблон.

Связь с предыдущим уроком

В уроке про форматированные сообщения мы использовали этот подход:

- плагин `sendFormattedMessage` **отправляет сообщение**;
- но **возвращает стандартизированный результат**;
- вызывающая сторона может:
  - проигнорировать результат;
  - обработать ошибку;
  - использовать данные дальше.

То есть:

> helper-плагины делают действие response-плагин описывает результат

Это разделение ответственности — важный архитектурный принцип.

Можно ли сделать свой стандарт

Да, конечно.

`Common.Utils.Response` — это:

- рекомендованный стандарт;
- доступный на всех серверах Metabot;
- проверенный в бою.

Но если вам нужен:

- другой формат;
- дополнительные поля;

- специфичная структура под внешний API —

вы можете:

- скопировать подход;
- реализовать свой response-helper;
- использовать его внутри своего продукта.

Главное — **консистентность**.

Итог

В этом уроке мы зафиксировали ключевую практику Metabot:

- ошибки — это данные, а не исключения;
- успешные результаты и ошибки имеют единый формат;
- плагины и модули легко комбинируются;
- API и внутренняя логика говорят на одном языке.

Если при проектировании нового модуля у тебя возникает мысль «а может тут просто вернуть true?» — это хороший момент остановиться и проверить: **а не ломаешь ли ты этим общую архитектурную договорённость**.

Именно такие мелкие решения потом определяют, будет система масштабируемой или нет.

Версия #3
Artem Garashko создал 2026-01-02 09:03:30 UTC
Artem Garashko обновил 2026-01-03 08:15:46 UTC