Как писать API-эндпоинты в Metabot

Контракты ответов, обработка ошибок и рекомендации по архитектуре

В этом уроке мы разберём, **как правильно проектировать API-эндпоинты в Metabot**, чтобы они:

- были предсказуемыми для клиентов;
- корректно работали в цепочках сценариев;
- не ломали архитектурные договорённости платформы;
- легко масштабировались и поддерживались.

⚠ Этот урок **продолжает и опирается** на предыдущий:

📖 **Стандартизация успешных ответов и ошибок в Metabot**

Если вы ещё не знакомы с `Common.Utils.Response` — начните с него.

Архитектурный контекст

В Metabot API — это **граничный слой системы**:

- с одной стороны — HTTP / внешние клиенты / интеграции;
- с другой — бизнес-логика, плагины, сценарии, CJM, low-code.

Ключевой принцип:

> **API-эндпоинты используют тот же контракт ответов, что и внутренняя логика Metabot.**

Это означает:

- API **не изобретает свой формат ошибок**;
- API **не выбрасывает исключения наружу**;
- API **всегда возвращает предсказуемый JSON**.

Базовый шаблон API-эндпоинта

Каждый API-эндпоинт в Metabot — это **явная композиция зависимостей**:

- стандарт ответа;
- бизнес-модели;
- сервисы;
- авторизация (если требуется).

 Никакие классы не считаются “доступными по умолчанию” — всё подключается явно.

Базовый шаблон

Практически все API в Metabot строятся по одной схеме:

```
const { getSuccessResponse, getErrorResponse } = require('Common.Utils.Response')

// 1. Валидация входных данных
if (!request.json || !request.json.required_field) {
  return getErrorResponse("Missing required field: required_field")
}

// 2. Авторизация (если нужна)
// 3. Вызов бизнес-логики
try {
  const result = SomeService.doSomething(request.json)
  return getSuccessResponse({ result })
} catch (e) {
  return getErrorResponse(e.message || "Operation failed")
}
```

 Это не шаблон ради шаблона — это зафиксированная архитектурная договорённость.

Валидация входных данных

Простой случай

```
if (!request.json || !request.json.event_id) {
  return getErrorResponse("Missing required field: event_id")
}
```

Почему так:

- ошибка ожидаемая;
- это не исключение;
- клиент получает читаемое сообщение;
- выполнение сценария не ломается.

Расширенная валидация (через helper)

```
const { getValidationErrorResponse } = require('Common.Utils.Response')
const { validateRequiredRequestFields } = require('Common.Utils.Validation')

const validation = validateRequiredRequestFields(request.json, [
  'email',
  'password',
  'nickname'
])

if (validation !== true) {
  return getValidationErrorResponse(validation)
}
```

□□ Такой ответ можно:

- отобразить во фронте;
- проанализировать в AI-агенте;
- использовать в автоматических сценариях.

Авторизация внутри API

Авторизация **всегда делается до бизнес-логики**.

Пример с JWT:

```
const Auth = require('Common.Users.Auth')

const jwt = request.json?.jwt || null
const user = jwt ? Auth.verifyToken(jwt) : null

if (!user || !user.id) {
  return getErrorResponse("Unauthorized: invalid or missing token")
}
```

Почему именно так:

- авторизация — это тоже **ожидаемая ошибка**;
- API не кидает `throw`;
- клиент всегда получает `success: false`.

Работа с бизнес-логикой (корректные примеры)

Пример: получение судей события

```
const { getSuccessResponse, getErrorResponse } = require('Common.Utils.Response')
const Judges = require('Business.Events.Judges')

const { event_id } = request.json || {}

if (!event_id) {
  return getErrorResponse("Missing required field: event_id")
}
```

```
try {
  const judges = Judges.find({ eventId: event_id })
  return getSuccessResponse({ judges })
} catch (error) {
  return getErrorResponse(error.message || "Failed to load judges")
}
```

✔ Явно видно:

- откуда берётся `Judges`;
- где бизнес-логика;
- где контракт ответа.

□□ API не знает деталей реализации `Judges.find` □□ API знает только: *успех или ошибка*

Пример: работа с Event и Round (без магии)

```
const { getSuccessResponse, getErrorResponse } = require('Common.Utils.Response')
const Event = require('Business.Events.Event')
const Round = require('Business.Events.Round')

if (!request.json || !request.json.event_id) {
  return getErrorResponse("Missing required field: event_id")
}

let event
try {
  event = new Event().setEventById(request.json.event_id)
} catch (e) {
  return getErrorResponse("Event not found: " + e.message)
}

const currentRoundId = event.getCurrentRoundId()

if (!currentRoundId) {
  return getErrorResponse("Current round not set for this event")
}
```

```
let round
try {
  round = new Round().setRoundById(currentRoundId)
} catch (e) {
  return getErrorResponse("Round not found: " + e.message)
}

return getSuccessResponse(round.getAllData())
```

Здесь **принципиально важно**, что:

- `Event` и `Round` подключены явно;
- API-слой не «угадывает», что есть в окружении;
- пример можно **скопировать и использовать без сюрпризов**.

Каждый шаг:

- либо возвращает валидные данные;
- либо завершает API **предсказуемым error-response**.

Возврат данных

Один объект

```
return getSuccessResponse(event.getAllData())
```

Коллекция

```
return getSuccessResponse({ submissions })
```

Комбинированный результат

```
return getSuccessResponse({
  event_id,
  rounds,
```

```
  can_participate: true
})
```

□□ Нет жёсткого правила, *что* возвращать □□ Есть правило, *как* возвращать

Цепочки логики без исключений

```
const { getSuccessResponse, getErrorResponse } = require('Common.Utils.Response')
const Auth = require('Common.Users.Auth')

const result = Auth.login(email, password)

if (!result.success) {
  return getErrorResponse(result)
}

return getSuccessResponse(result)
```

□□ API **не пересобирает ошибку**, а прокидывает её дальше.

Почему это правильно:

- бизнес-метод уже использует `Common.Utils.Response`;
- API **не ломает контракт**;
- ошибка поднимается наверх без искажения.

HTTP-статусы и `success`

В Metabot принято:

- HTTP-статус почти всегда `200`;
- реальный статус операции — в теле ответа.

Почему:

- API используется не только браузерами;
- сценарии и AI-агенты не всегда работают с HTTP-кодами;
- единый формат упрощает автоматическую обработку.

□□ `success: false` **важнее, чем** `HTTP 400`

Антипаттерны (так делать не стоит)

□ Возвращать `true / false` □ Кидать `throw` для валидации □ Возвращать строки вместо объектов □ Мешать разные форматы ошибок □ Делать API, который возвращает «что попало»

Если ловишь себя на мысли:

> «Да тут проще просто вернуть true…»

— это хороший момент остановиться и проверить: **а не ломаешь ли ты контракт всей платформы?**

Итог

API в Metabot — это:

- не просто HTTP-эндпоинты;
- а **часть общей архитектуры выполнения**.

Ключевые принципы:

- API использует `Common.Utils.Response`;
- ошибки — это данные;
- исключения — редкость;
- формат ответа стабилен;
- бизнес-логика и API говорят на одном языке.

Если вы придерживаетесь этих правил:

- API легко читать;
- его просто документировать;
- его удобно использовать во фронте, интеграциях и AI-агентах;
- система остаётся масштабируемой.

Версия #1
Artem Garashko создал 2026-02-02 12:56:34 UTC
Artem Garashko обновил 2026-02-02 13:08:24 UTC