

8 правил проектирования API

1. Используйте глобально уникальные идентификаторы

Хороший тон при разработке API — использовать для идентификаторов сущностей глобально уникальные строки, либо семантические (например, "lungo" для видов напитков), либо случайные (например UUID-4). Это может чрезвычайно пригодиться, если вдруг придётся объединять данные из нескольких источников под одним идентификатором.

Мы вообще склонны порекомендовать использовать идентификаторы в urn-подобном формате, т.е. `urn:order:<uuid>` (или просто `order:<uuid>`), это сильно помогает с отладкой legacy-систем, где по историческим причинам есть несколько разных идентификаторов для одной и той же сущности — тогда неймспейсы в urn помогут быстро понять, что это за идентификатор и нет ли здесь ошибки использования.

Отдельное важное следствие: **не используйте инкрементальные номера как идентификаторы**. Помимо вышесказанного, это плохо ещё и тем, что ваши конкуренты легко смогут подсчитать, сколько у вас в системе каких сущностей и тем самым вычислить, например, точное количество заказов за каждый день наблюдений.

NB: в этой книге часто используются короткие идентификаторы типа "123" в примерах — это для удобства чтения на маленьких экранах, повторять эту практику в реальном API не надо.

2. Клиент всегда должен знать полное состояние системы

Правило можно ещё сформулировать так: не заставляйте клиент гадать.

Плохо:

```
// Создаёт заказ и возвращает его id
POST /v1/orders
{ ... }
→
{ "order_id" }
```

```
// Возвращает заказ по его id
GET /v1/orders/{id}
// Заказ ещё не подтверждён
// и ожидает проверки
→ 404 Not Found
```

— хотя операция будто бы выполнена успешно, клиенту необходимо самостоятельно запомнить идентификатор заказа и периодически проверять состояние `GET /v1/orders/{id}`. Этот паттерн плох сам по себе, но ещё и усугубляется двумя обстоятельствами:

- клиент может потерять идентификатор, если произошёл системный сбой в момент между отправкой запроса и получением ответа или было повреждено (очищено) системное хранилище данных приложения;
- потребитель не может воспользоваться другим устройством; фактически, знание о сделанном заказе привязано к конкретному юзер-агенту.

В обоих случаях потребитель может решить, что заказ по какой-то причине не создался — и сделать повторный заказ со всеми вытекающими отсюда проблемами.

Хорошо:

```
// Создаёт заказ и возвращает его
POST /v1/orders
{ <параметры заказа> }
→
{
  "order_id",
  // Заказ создаётся в явном статусе
  // «идёт проверка»
  "status": "checking",
  ...
}
```

```
// Возвращает заказ по его id
GET /v1/orders/{id}
→
{ "order_id", "status" ... }
```

```
// Возвращает все заказы пользователя
// во всех статусах
GET /v1/users/{id}/orders
```

3. Избегайте двойных отрицаний

Плохо: `"dont_call_me": false`

— люди в целом плохо считают двойные отрицания. Это провоцирует ошибки.

Лучше: `"prohibit_calling": true` или `"avoid_calling": true`

— читается лучше, хотя обольщаться все равно не следует. Насколько это возможно откажитесь от семантически двойных отрицаний, даже если вы придумали «негативное» слово без явной приставки «не».

Стоит также отметить, что в использовании законов де Моргана ошибиться ещё проще, чем в двойных отрицаниях. Предположим, что у вас есть два флага:

```
GET /coffee-machines/{id}/stocks
→
{
  "has_beans": true,
  "has_cup": true
}
```

Условие «кофе можно приготовить» будет выглядеть как `has_beans && has_cup` — есть и зерно, и стакан. Однако, если по какой-то причине в ответе будут отрицания тех же флагов:

```
{
  "beans_absence": false,
  "cup_absence": false
}
```

— то разработчику потребуется вычислить флаг `!beans_absence && !cup_absence` ⇔ `!(beans_absence || cup_absence)`, а вот этом переходе ошибиться очень легко, и избегание

двойных отрицаний помогает слабо. Здесь, к сожалению, есть только общий совет «избегайте ситуаций, когда разработчику нужно вычислять такие флаги».

4. Избегайте неявного приведения типов

Этот совет парадоксально противоположен предыдущему. Часто при разработке API возникает ситуация, когда добавляется новое необязательное поле с непустым значением по умолчанию. Например:

```
POST /v1/orders
{}
→
{
  "contactless_delivery": true
}
```

Новая опция `contactless_delivery` является необязательной, однако её значение по умолчанию — `true`. Возникает вопрос, каким образом разработчик должен отличить явное *нежелание* пользоваться опцией (`false`) от незнания о её существовании (поле не задано). Приходится писать что-то типа такого:

```
if (Type(order.contactless_delivery) == 'Boolean' &&
    order.contactless_delivery == false) { ... }
```

Эта практика ведёт к усложнению кода, который пишут разработчики, и в этом коде легко допустить ошибку, которая по сути меняет значение поля на противоположное. То же самое произойдёт, если для индикации отсутствия значения поля использовать специальное значение типа `null` или `-1`.

В этих ситуациях универсальное правило — все новые необязательные булевы флаги должны иметь значение по умолчанию `false`.

Хорошо

```
POST /v1/orders
{}
→
{
```

```
"force_contact_delivery": false
}
```

Если же требуется ввести небулево поле, отсутствие которого трактуется специальным образом, то следует ввести пару полей.

Плохо:

```
// Создаёт пользователя
POST /users
{ ... }
→
// Пользователи создаются по умолчанию
// с указанием лимита трат в месяц
{
  ...
  "spending_monthly_limit_usd": "100"
}
// Для отмены лимита требуется
// указать значение null
POST /users
{
  ...
  "spending_monthly_limit_usd": null
}
```

Хорошо

```
POST /users
{
  // true – у пользователя снят
  //   лимит трат в месяц
  // false – лимит не снят
  //   (значение по умолчанию)
  "abolish_spending_limit": false,
  // Необязательное поле, имеет смысл
  // только если предыдущий флаг
  // имеет значение false
  "spending_monthly_limit_usd": "100",
  ...
}
```

NB: противоречие с предыдущим советом состоит в том, что мы специально ввели отрицающий флаг («нет лимита»), который по правилу двойных отрицаний пришлось переименовать в `abolish_spending_limit`. Хотя это и хорошее название для отрицательного флага, семантика его довольно неочевидна, разработчикам придётся как минимум покопаться в документации. Таков путь.

5. Избегайте частичных обновлений

Плохо:

```
// Возвращает состояние заказа
// по его идентификатору
GET /v1/orders/123
→
{
  "order_id",
  "delivery_address",
  "client_phone_number",
  "client_phone_number_ext",
  "updated_at"
}
// Частично перезаписывает заказ
```

```
PATCH /v1/orders/123
{ "delivery_address" }
→
{ "delivery_address" }
```

— такой подход часто практикуют для того, чтобы уменьшить объёмы запросов и ответов, плюс это позволяет дёшево реализовать совместное редактирование. Оба этих преимущества на самом деле являются мнимыми.

Во-первых, экономия объёма ответа в современных условиях требуется редко. Максимальные размеры сетевых пакетов (MTU, Maximum Transmission Unit) в настоящее время составляют более килобайта; пытаться экономить на размере ответа, пока он не превышает килобайт — попросту бессмысленная трата времени.

Перерасход трафика возникает, если:

- не предусмотрен постраничный перебор данных;
- не предусмотрены ограничения на размер значений полей;
- передаются бинарные данные (графика, аудио, видео и т.д.).

Во всех трёх случаях передача части полей в лучшем случае замаскирует проблему, но не решит. Более оправдан следующий подход:

- для «тяжёлых» данных сделать отдельные эндпоинты;
- ввести пагинацию и лимитирование значений полей;
- на всём остальном не пытаться экономить.

Во-вторых, экономия размера ответа выйдет боком как раз при совместном редактировании: один клиент не будет видеть, какие изменения внёс другой. Вообще в 9 случаях из 10 (а фактически — всегда, когда размер ответа не оказывает значительного влияния на производительность) во всех отношениях лучше из любой модифицирующей операции возвращать полное состояние сущности в том же формате, что и из операции доступа на чтение.

В-третьих, этот подход может как-то работать при необходимости перезаписать поле. Но что делать, если поле требуется сбросить к значению по умолчанию? Например, как *удалить* `client_phone_number_ext`?

Часто в таких случаях прибегают к специальным значениям, которые означают удаление поля, например, `null`. Но, как мы разобрали выше, это плохая практика. Другой вариант — запрет необязательных полей, но это существенно усложняет дальнейшее развитие API.

Хорошо: можно применить одну из двух стратегий.

Вариант 1: разделение эндпоинтов. Редактируемые поля группируются и выносятся в отдельный эндпоинт. Этот подход также хорошо согласуется с принципом декомпозиции,

который мы рассматривали в предыдущем разделе.

```
// Возвращает состояние заказа
// по его идентификатору
GET /v1/orders/123
→
{
  "order_id",
  "delivery_details": {
    "address"
  },
  "client_details": {
    "phone_number",
    "phone_number_ext"
  },
  "updated_at"
}
// Полностью перезаписывает
// информацию о доставке заказа
PUT /v1/orders/123/delivery-details
{ "address" }
// Полностью перезаписывает
// информацию о клиенте
PUT /v1/orders/123/client-details
{ "phone_number" }
```

Теперь для удаления `client_phone_number_ext` достаточно *не* передавать его в `PUT client-details`. Этот подход также позволяет отделить неизменяемые и вычисляемые поля (`order_id` и `updated_at`) от изменяемых, не создавая двусмысленных ситуаций (что произойдёт, если клиент попытается изменить `updated_at`?). В этом подходе также можно в ответах операций `PUT` возвращать объект заказа целиком (однако следует использовать какую-то конвенцию именования).

Вариант 2: разработать формат описания атомарных изменений.

```
POST /v1/order/changes
X-Idempotency-Token: <токен идемпотентности>
{
  "changes": [{
    "type": "set",
    "field": "delivery_address",
```



```
"value": &lt;новое значение>
}, {
  "type": "unset",
  "field": "client_phone_number_ext"
}]
}
```

Этот подход существенно сложнее в имплементации, но является единственным возможным вариантом реализации совместного редактирования, поскольку он явно отражает, что в действительности делал пользователь с представлением объекта. Имея данные в таком формате возможно организовать и оффлайн-редактирование, когда пользовательские изменения накапливаются и потом сервер автоматически разрешает конфликты, «перебазируя» изменения.

6. Избегайте неатомарных операций

С применением массива изменений часто возникает вопрос: что делать, если часть изменений удалось применить, а часть — нет? Здесь правило очень простое: если вы можете обеспечить атомарность, т.е. выполнить либо все изменения сразу, либо ни одно из них — сделайте это.

Плохо:

```
// Возвращает список рецептов
GET /v1/recipes
→
{
  "recipes": [{
    "id": "lungo",
    "volume": "200ml"
  }, {
    "id": "latte",
    "volume": "300ml"
  }]
}

// Изменяет параметры
```

```
PATCH /v1/recipes
{
  "changes": [{
    "id": "lungo",
    "volume": "300ml"
  }, {
    "id": "latte",
    "volume": "-1ml"
  }]
}
```

→ 400 Bad Request

// Перечитываем список

```
GET /v1/recipes
→
{
  "recipes": [{
    "id": "lungo",
    // Это значение изменилось
    "volume": "300ml"
  }, {
    "id": "latte",
    // А это нет
    "volume": "300ml"
  }]
}
```

— клиент никак не может узнать, что операция, которую он посчитал ошибочной, на самом деле частично применена. Даже если индцировать это в ответе, у клиента нет способа понять — значение объёма лунго изменилось вследствие запроса, или это конкурирующее изменение, выполненное другим клиентом.

Если способа обеспечить атомарность выполнения операции нет, следует очень хорошо подумать над её обработкой. Следует предоставить способ получения статуса каждого изменения отдельно.

Лучше:

```
PATCH /v1/recipes
{
  "changes": [{
```

```
    "recipe_id": "lungo",
    "volume": "300ml"
  }, {
    "recipe_id": "latte",
    "volume": "-1ml"
  }]
}
// Можно воспользоваться статусом
// «частичного успеха», если он предусмотрен
// протоколом
→ 200 OK
{
  "changes": [{
    "change_id",
    "occurred_at",
    "recipe_id": "lungo",
    "status": "success"
  }, {
    "change_id",
    "occurred_at",
    "recipe_id": "latte",
    "status": "fail",
    "error"
  }]
}
```

Здесь:

- `change_id` — уникальный идентификатор каждого атомарного изменения;
- `occurred_at` — время проведения каждого изменения;
- `error` — информация по ошибке для каждого изменения, если она возникла.

Не лишним будет также:

- введение `sequence_id` в запросе, чтобы гарантировать порядок исполнения операций и соотнесение порядка статусов изменений в ответе с запросом;
- предоставить отдельный эндпойнт `/changes-history`, чтобы клиент мог получить информацию о выполненных изменениях, если во время обработки запроса произошла сетевая ошибка или приложение перезагрузилось.

Неатомарные изменения нежелательны ещё и потому, что вносят неопределённость в понятие идемпотентности, даже если каждое вложенное изменение идемпотентно.

Рассмотрим такой пример:

```
PATCH /v1/recipes
{
  "idempotency_token",
  "changes": [{
    "recipe_id": "lungo",
    "volume": "300ml"
  }, {
    "recipe_id": "latte",
    "volume": "400ml"
  }]
}
→ 200 OK
{
  "changes": [{
    ...
    "status": "success"
  }, {
    ...
    "status": "fail",
    "error": {
      "reason": "too_many_requests"
    }
  }]
}
```

Допустим, клиент не смог получить ответ и повторил запрос с тем же токеном идемпотентности.

```
PATCH /v1/recipes
{
  "idempotency_token",
  "changes": [{
    "recipe_id": "lungo",
    "volume": "300ml"
  }, {
    "recipe_id": "latte",
    "volume": "400ml"
  }]
}
```

```
}  
→ 200 OK  
{  
  "changes": [{  
    ...  
    "status": "success"  
  }, {  
    ...  
    "status": "success",  
  }]  
}
```

По сути, для клиента всё произошло ожидаемым образом: изменения были внесены, и последний полученный ответ всегда корректен. Однако по сути состояние ресурса после первого запроса отличалось от состояния ресурса после второго запроса, что противоречит самому определению идемпотентности.

Более корректно было бы при получении повторного запроса с тем же токеном ничего не делать и возвращать ту же разбивку ошибок, что была дана на первый запрос — но для этого придётся её каким-то образом хранить в истории изменений.

На всякий случай уточним, что вложенные операции должны быть сами по себе идемпотентны. Если же это не так, то следует сгенерировать внутренние ключи идемпотентности на каждую вложенную операцию в отдельности.

7. Соблюдайте правильный порядок ошибок

Во-первых, всегда показывайте неразрешимые ошибки прежде разрешимых:

```
POST /v1/orders  
{  
  "recipe": "lngo",  
  "offer"  
}  
→ 409 Conflict  
{  
  "reason": "offer_expired"  
}
```

```
// Повторный запрос
// с новым `offer`
POST /v1/orders
{
  "recipe": "lngo",
  "offer"
}
→ 400 Bad Request
{
  "reason": "recipe_unknown"
}
```

— какой был смысл получать новый `offer`, если заказ все равно не может быть создан?

Во-вторых, соблюдайте такой порядок разрешимых ошибок, который приводит к наименьшему раздражению пользователя и разработчика. В частности, следует начинать с более значимых ошибок, решение которых требует более глобальных изменений.

Плохо:

```
POST /v1/orders
{
  "items": [{ "item_id": "123", "price": "0.10" }]
}
→
409 Conflict
{
  "reason": "price_changed",
  "details": [{ "item_id": "123", "actual_price": "0.20" }]
}
// Повторный запрос
// с актуальной ценой
POST /v1/orders
{
  "items": [{ "item_id": "123", "price": "0.20" }]
}
→
409 Conflict
{
  "reason": "order_limit_exceeded",
  "localized_message": "Лимит заказов превышен"
```

```
}
```

— какой был смысл показывать пользователю диалог об изменившейся цене, если и с правильной ценой заказ он сделать все равно не сможет? Пока один из его предыдущих заказов завершится и можно будет сделать следующий заказ, цену, наличие и другие параметры заказа всё равно придётся корректировать ещё раз.

В-третьих, постройте таблицу: разрешение какой ошибки может привести к появлению другой, иначе вы можете показать одну и ту же ошибку несколько раз, а то и вовсе зациклить разрешение ошибок.

```
// Создаём заказ с платной доставкой
POST /v1/orders
{
  "items": 3,
  "item_price": "3000.00"
  "currency_code": "MNT",
  "delivery_fee": "1000.00",
  "total": "10000.00"
}
→ 409 Conflict
// Ошибка: доставка становится бесплатной
// при стоимости заказа от 9000 тугриков
{
  "reason": "delivery_is_free"
}

// Создаём заказ с бесплатной доставкой
POST /v1/orders
{
  "items": 3,
  "item_price": "3000.00"
  "currency_code": "MNT",
  "delivery_fee": "0.00",
  "total": "9000.00"
}
→ 409 Conflict
// Ошибка: минимальная сумма заказа
// 10000 тугриков
{
  "reason": "below_minimal_sum",
```

```
"currency_code": "MNT",  
"minimal_sum": "10000.00"  
}
```

Легко заметить, что в этом примере нет способа разрешить ошибку в один шаг — эту ситуацию требуется предусмотреть отдельно, и либо изменить параметры расчета (минимальная сумма заказа не учитывает скидки), либо ввести специальную ошибку для такого кейса.

8. Отсутствие результата — тоже результат

Если сервер корректно обработал вопрос и никакой внештатной ситуации не возникло — следовательно, это не ошибка. К сожалению, весьма распространён антипаттерн, когда отсутствие результата считается ошибкой.

Плохо

```
POST /search  
{  
  "query": "lungo",  
  "location": &lt;положение пользователя>  
}  
→ 404 Not Found  
{  
  "localized_message":  
    "Рядом с вами не делают лунго"  
}
```

Статусы `4xx` означают, что клиент допустил ошибку; однако в данном случае никакой ошибки сделано не было, ни пользователем, ни разработчиком: клиент же не может знать заранее, готовят здесь лунго или нет.

Хорошо:

```
POST /search  
{  
  "query": "lungo",  
  "location": &lt;положение пользователя>
```



```
}  
→ 200 OK  
{  
  "results": []  
}
```

Это правило вообще можно упростить до следующего: если результатом операции является массив данных, то пустота этого массива — не ошибка, а штатный ответ. (Если, конечно, он допустим по смыслу; пустой массив координат, например, является ошибкой.)

Версия #6

Seryak создал Fri, Jan 8, 2021 8:42 AM

Seryak обновил Fri, Jan 8, 2021 9:02 AM