Engineer

REST

🏛 архітектурний стиль вебу, побудований навколо ресурсів

Що це таке

REST, скорочення від Representational State Transfer, — це архітектурний стиль, який Рой Філдінг виклав у розділі 5 своєї дисертації 2000 року. Це не протокол і не стандарт. Це набір обмежень на те, як можуть взаємодіяти клієнт, сервер і посередники між ними, і Філдінг пояснює, що дає кожне обмеження і, для більшості з них, чого воно коштує. RESTful — це прикметник: у строгому сенсі він описує систему, яка дотримується всіх обмежень. У щоденному вжитку обидва слова розтягують значно далі, майже на будь-який HTTP API, що обмінюється JSON, незалежно від того, чи дотримується він обмежень, описаних нижче.

Центральне поняття — ресурс. Філдінг визначає його широко: ресурсом може бути будь-яка інформація, якій можна дати назву, — документ, зображення, колекція інших ресурсів, людина, навіть сьогоднішня погода в певному місті. Мережею ніколи не йде сам ресурс, лише його представлення, тобто послідовність байтів разом із метаданими, що ці байти описують, наприклад HTML-документ або зображення JPEG. Клієнт діє на ресурс, надсилаючи або отримуючи представлення його поточного чи бажаного стану.

Філдінг будує стиль від порожнього набору обмежень, додаючи їх по одному. Клієнт-сервер відокремлює користувацький інтерфейс від зберігання даних, щоб ці дві частини могли розвиватися незалежно. Відсутність стану означає, що кожен запит містить усе потрібне, щоб його зрозуміти, і не спирається на жоден контекст, збережений на сервері, тож стан сеансу повністю лежить на клієнті. Кеш вимагає, щоб кожну відповідь було явно чи неявно позначено як придатну або непридатну для кешування, щоб клієнт міг повторно використати придатну відповідь для пізнішого рівноцінного запиту. Уніфікований інтерфейс, описаний нижче, дає всім компонентам той самий загальний інтерфейс. Багатошарова система

означає, що кожен компонент бачить лише той шар, з яким говорить безпосередньо, і це лишає місце для проксі, шлюзів і брандмауерів посередині без змін в інтерфейсах з обох боків. Код на вимогу дає серверу змогу розширити клієнт, надіславши йому скрипти для виконання, і Філдінг робить його єдиним необов'язковим обмеженням, бо воно зменшує видимість.

Уніфікований інтерфейс Філдінг називає центральною рисою, що відрізняє REST від інших мережевих стилів, і розкладає його на чотири обмеження: ресурси ідентифікуються, у вебі — через URL; ресурсами маніпулюють через представлення; повідомлення самоописові, тож їхній зміст видно кожному посередникові; а гіпермедіа є рушієм стану застосунку, тобто клієнт переходить з одного стану в наступний, обираючи серед переходів, запропонованих у представленнях, які він уже має, так само як людина переходить за посиланнями в браузері.

Застосуйте поняття до рішення

Застосуйте «Що таке REST?» до конкретного рішення. Ми можемо пов'язати поняття із системою, яку ви створюєте або купуєте. Та частина, якої користувач не бачить: де зберігаються дані, які діють правила і як інші системи можуть із нею взаємодіяти.

Переглянути наші послуги Зв'язатися з нами →

Перевірте припущення за терміном

Використайте пояснення та пов'язані роботи, щоб сформулювати запитання для проєкту. Ми можемо перевірити припущення разом до вибору інструмента. Спроєктувала й підтримувала 6 пайплайнів ETL/ELT — 6 пайплайнів ETL на трьох рушіях баз даних

Спроєктувала й підтримувала 6 пайплайнів ETL/ELT Зв'язатися з нами →

Чому це важливо

Кожне обмеження дає натомість щось конкретне. Відсутність стану дає системі моніторингу змогу зрозуміти запит, не дивлячись далі за нього, полегшує відновлення після часткових збоїв і дає серверу змогу звільнити ресурси, щойно він відповів. Кеш може повністю прибрати частину взаємодій. Шари дають посередникам змогу розподіляти навантаження між мережами й машинами, а одному кешу на межі організації — обслуговувати всіх усередині неї. І оскільки уніфікований інтерфейс відокремлює реалізації від послуг, які вони надають, клієнт і сервер можуть змінюватися кожен окремо, і жодному з них не доводиться встигати за іншим.

Так само відверто Філдінг пише про ціну. Відсутність стану повторює в кожному запиті дані, які спільний контекст на сервері заощадив би. Кеш може віддати застарілі дані. Кожен шар додає накладні витрати й затримку.

Уніфікований інтерфейс знижує ефективність, бо інформація передається в стандартизованій формі, а не в такій, що скроєна під потреби конкретної програми. Філдінг проєктував його для передавання великих порцій гіпермедійних даних, звичайного випадку у вебі, і прямо каже, що для інших видів взаємодії він не оптимальний. Саме на цьому ґрунті стоять альтернативи. Сервер GraphQL приймає всі запити в одній кінцевій точці, ідентифікує свої сутності чимось іншим, ніж URL, і дає клієнтові обрати, які саме поля повернуться, а gRPC викликає методи, оголошені у визначенні сервісу. Кожен із них відмовляється від частини уніфікованого інтерфейсу, щоб пасувати іншому виду трафіку.

Для HTTP API значна частина уніфікованого інтерфейсу вже готова. HTTP визначає методи з узгодженим значенням, коди стану й поля заголовка для кешування та умовних запитів. Оскільки це значення спільне, проксі чи кеш на шляху може діяти щодо повідомлення, нічого не знаючи про конкретний API, а API, що використовує ці частини так, як їх визначено, отримує весь цей механізм задарма. API, який дає клієнтам шаблони URL для заповнення замість посилань, за якими можна перейти, відмовляється від обмеження гіпермедіа, і його клієнти прив’язуються до структури URL на сервері: коли вона змінюється, вони ламаються, а саме цей зв’язок уніфікований інтерфейс і має прибрати.

Як це виглядає на практиці

Обмін нижче створює замовлення, і у відповідь приходить адреса нового ресурсу в `Location` і тег сутності (entity tag) в `ETag`. RFC 9110 сам наводить `ETag` у відповіді 201 як приклад: тег може слугувати валідатором у подальших умовних запитах, щоб два клієнти не могли непомітно перезаписати зміни одне одного.

```
POST /orders HTTP/1.1
Host: api.example.com
Content-Type: application/json

{"item": "notebook", "quantity": 2}

HTTP/1.1 201 Created
Location: /orders/7421
ETag: "v1"
Content-Type: application/json

{"id": 7421, "item": "notebook", "quantity": 2, "status": "open"}
```

Кожен метод має значення, на яке може покладатися будь-який посередник. RFC 9110 визначає GET, HEAD, OPTIONS і TRACE як безпечні: клієнт не просить і не очікує жодної зміни стану на сервері. PUT і DELETE, як і безпечні методи, ідемпотентні, тож кілька однакових запитів мають той самий очікуваний ефект, що й один. POST не є ні тим, ні іншим, і RFC 5789 каже те саме про PATCH. Саме завдяки цим двом властивостям краулер може ходити за посиланнями, не завдаючи шкоди, кеш — завантажувати наперед, а клієнт — повторити PUT, з'єднання якого обірвалося, перш ніж прийшла відповідь.

Щоб керувати REST API, досить універсального HTTP-клієнта, і на цьому сайті видно і що це заощаджує, і чого це коштує. Коли другий сервер підготували з коду, брандмауери обох провайдерів налаштували безпосередньо через їхні REST API за допомогою універсального HTTP-завдання замість колекції постачальника. Так зникає залежність разом із дрейфом її версій, а платою стає ручний опис структури кожного запиту. Та сама сторінка називає й другу ціну: універсальне HTTP-завдання не оголошує підтримки режиму перевірки (check mode), тож пробний прогін плею, який створює машину, пропускає в ньому всі завдання.

Де виникають проблеми

GET щось змінює. Адреса на кшталт `/orders/7421/cancel`, яка скасовує замовлення, коли її відкривають, порушує обіцянку, що GET безпечний. RFC 9110 пояснює, навіщо існує ця обіцянка: щоб краулери й кеші, які завантажують наперед, могли працювати, не боячись завдати шкоди. Перший краулер чи кеш із попереднім завантаженням, що натрапить на посилання, скасує замовлення. Натомість перенесіть дію в метод: POST до замовлення або PATCH його статусу.

Кожна відповідь — 200 OK. Деякі API повертають 200 на все, а справжній результат, помилка це чи ні, кладуть у тіло повідомлення. Повідомлення перестають бути самоописовими: кеші, проксі, моніторинг і клієнтські бібліотеки читають код стану, і всі вони бачать успіх. RFC 9457 визначає стандартне тіло для помилок, `application/problem+json`, з полями на кшталт `type`, `title`, `status` і `detail`. Поле `status` має лише дорадчий характер: сервер мусить надіслати той самий код стану в самій відповіді HTTP, щоб універсальне програмне забезпечення HTTP, яке нічого не знає про цей формат, поводилося правильно.

Повторений POST створює два замовлення. Коли з'єднання обривається після того, як сервер обробив POST, але до того, як клієнт прочитав відповідь, клієнт не може знати, чи існує замовлення. RFC 9110 радить клієнтові не повторювати неідемпотентний запит автоматично, якщо він не знає, що запит усе ж ідемпотентний, або не може встановити, що першу спробу так і не було виконано, а проксі взагалі не має права цього робити. Там також зазначено, що деякі клієнти обирають ризикованіший шлях і все одно повторюють POST, коли з'єднання закривається, перш ніж надійшла бодай частина відповіді, особливо на постійному з'єднанні, яке стояло без діла. Один вихід — дати клієнтові самому обрати ідентифікатор і надіслати замовлення за цією адресою через PUT, бо PUT ідемпотентний: перший запит створює замовлення з кодом 201, а повтор лише замінює його тим самим станом.

Два клієнти перезаписують один одного. Обидва читають замовлення, обидва змінюють різні поля, й обидва надсилають усе назад через PUT: другий запис непомітно скасовує перший. RFC 9110 називає це проблемою втраченого оновлення (lost update). Розв'язок — тег сутності, який кожен клієнт отримав, коли читав замовлення, надісланий назад у `If-Match`. Тоді сервер застосовує зміну, лише якщо ресурс досі в тому стані, який клієнт бачив востаннє; якщо ні, сервер не має права її застосовувати і може повідомити про це кодом 412 Precondition Failed. Після цього клієнт знову читає замовлення, повторно вносить свою зміну й пробує ще раз.

```
PUT /orders/7421 HTTP/1.1
Host: api.example.com
If-Match: "v1"
Content-Type: application/json

{"item": "notebook", "quantity": 3, "status": "open"}

HTTP/1.1 412 Precondition Failed
```

Сервер пам'ятає розмову. Сервер, який між запитами тримає стан для кожного клієнта, наприклад поточну сторінку списку чи напівзаповнену форму в пам'яті, відмовився від обмеження відсутності стану. Запит більше не має сенсу сам по собі, балансувальник навантаження мусить відправляти кожного клієнта назад на ту саму машину, а перезапуск викидає стан. Причини, з яких Філдінг обирає відсутність стану, — саме те, що тут утрачено: видимість, легше відновлення після часткових збоїв і сервер, який може

звільнити ресурси, щойно відповів. Покладіть стан у запит, як токен сторінки в URL, або зробіть незавершене окремим ресурсом, наприклад чернеткою замовлення, яку може прочитати будь-який сервер.

Що почитати далі

- Fielding, chapter 5: Representational State Transfer (REST) — визначення: кожне обмеження, властивості, які воно дає, і чим воно за них платить.
- RFC 9110: HTTP Semantics — методи, які з них безпечні та ідемпотентні, коди стану й умовні запити з `ETag` та `If-Match`.
- RFC 5789: PATCH Method for HTTP — часткова зміна ресурсу і чому PATCH не є ні безпечним, ні ідемпотентним.
- RFC 9457: Problem Details for HTTP APIs — стандартне тіло JSON для відповідей з помилкою, яке зберігає значення коду стану.

Словник

Обирайте за фактичними умовами

Опишіть робочий процес, наявні інтерфейси та користувачів, яким потрібна зміна. Ми можемо визначити перевірюваний результат і спосіб передачі, який дасть вашій команді змогу підтримувати його.

Дивитися, скільки це коштує Зв'язатися з нами →

Зробіть термінологію практичною

Запитайте, як «Що таке REST?» стосується вашої організації. Конкретного прикладу з поточної системи достатньо для початку розмови.

Переглянути наші послуги

welcome@engineer.company +45 50 10 61 84 Копенгаген, Данія

Візьміть цю сторінку із собою

Візитівка

Дбайливо зроблено з любов'ю — D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Авторське право © 2025 – дотепер · Engineer ApS · Усі права захищено.

Надруковано з https://engineer.company/uk/glossary/rest/