مختصر كتابي

Why an API Is Not All of Backend Engineering



لماذا لا يكفي أن تعمل الـAPI لكي تكون عملية الشراء صحيحة؟

تخيّل أنك تشتري سماعة من موقع. ضغطت زر الدفع، فظلت الدائرة تدور، ثم ظهرت رسالة: «حدث تأخير، حاول مرة أخرى». ضغطت مرة ثانية، وبعد دقيقة فتحت صفحة طلباتك فوجدت سماعتين بدل واحدة، مع أنك أردت الشراء مرة واحدة.

من جهة المبرمج نجحت كل خطوة في التجربة العادية: استقبل الـServer الـRequest، وحفظ الـOrder في الـDatabase، وأرسل Response فيه الـOrder ID. هنا يظهر الفرق بين أن تكتب API تعمل، وأن تفهم **هندسة الـ**(Backend Engineering) **Backend**. الـAPI تحدد كيف يكلّم برنامج برنامجًا آخر: ماذا يرسل، وإلى أين، وما شكل الـResponse. أما صحة التجربة كلها فتحتاج أن تفهم ما يحدث أثناء الانتظار، وعند انقطاع الاتصال، وعندما تُطلب العملية نفسها أكثر من مرة.

في المطعم ثلاثة أشياء مختلفة: أن تطلب، وأن يسجّل المطعم الطلب، وأن تعرف أنه سُجّل. وفي البرامج كذلك: يرسل الـClient الـRequest، وينفذ الـServer العملية، ثم يعود الـResponse. والـClient هو البرنامج الذي يرسل، مثل تطبيق الهاتف أو الكود في المتصفح، أما المستخدم فهو الشخص الذي ضغط الزر. والبرنامج لا يستطيع أن يخمّن نيتك من تشابه الطلبين؛ يحتاج طريقة واضحة يميّز بها.

مثال بالأرقام: في اللحظة صفر أرسل الـClient طلب إنشاء Order. بعد 20 millisecond حفظه الـServer، لكن الرد كان سيخرج بعد 400 millisecond. والـClient ينتظر 100 millisecond فقط، فحدث عنده **انتهاء المهلة** (Timeout). الأرقام للتوضيح، والمهم الترتيب: الحفظ وقع قبل الـTimeout.

الـTimeout لا يعني أن العملية فشلت، بل أن الـClient لم يستلم الرد في الوقت الذي حدده. ربما لم يصل الـRequest، وربما وصل والـServer ما زال يعمل، وربما تمت العملية وتأخر الرد. فإذا اعتبرته فشلًا مؤكدًا وأعدت المحاولة (Retry)، فقد تكرر عملية تمت بالفعل. ومنع شراء المنتج مرتين ليس حلًا، فقد يريد المستخدم سماعة له وأخرى هدية. المطلوب التمييز بين محاولة ثانية للعملية نفسها وعملية جديدة، وهذه هي المشكلة التي توجد من أجلها **الـIdempotency**. وإغلاق الصفحة لا يلغي الـOrder المحفوظ؛ الإلغاء يحتاج عملية مصممة بوضوح.

في المعمل يحفظ الـServer الـOrder ثم يؤخر الرد عمدًا، والـClient يتوقف عن الانتظار قبله، ثم يقارن عدد الـOrders قبل التجربة وبعدها. السطور الثلاثة الأولى من `server.mjs`، والسطران الأخيران من `client.mjs`:

```
const order=makeOrder(input); log('order_stored',{orderId:order.id});
await sleep(replyDelayMs); // Disconnect does not undo the stored order.
return json(201,order,{Location:`/orders/${order.id}`});

const before=await get('/health');console.log('before:',before.data.orders);
try{console.log(await post(`/orders?replyDelayMs=${replyMs}`,
{signal:AbortSignal.timeout(abortMs)}));}catch(e){console.log('client result
unknown:',e.name);}
```

وأدق رسالة للمستخدم في هذه اللحظة: «لم نتأكد بعد من حالة العملية، ونحن نراجعها»، على أن يراجع التطبيق الحالة فعلًا بعدها.

- الـTimeout يعني أن النتيجة غير معروفة للـClient، لا أن العملية فشلت.
- الـServer قد يعرف أن العملية تمت والـClient لا يعرف، ولا تناقض في ذلك.
- إعادة الإرسال دون تمييز قد تنشئ Order ثانيًا.
- إطالة الـTimeout تقلل المشكلة ولا تضمن وصول كل Response.
- ثلاثة أسئلة: هل النتيجة صحيحة؟ هل وصل الرد في وقت مناسب؟ هل الخدمة متاحة وقادرة على العمل؟

جرّب في المعمل تحت الدرس: شغّل `node server.mjs` في الطرفية الأولى، ثم `node client.mjs` في `Terminal 2`، ولاحظ أن العدد بعد الـ`timeout` زاد واحدًا.

الخلاصة: هندسة الـBackend تبدأ بسؤال: ماذا حدث فعلًا، ومن يعرف أنه حدث، وما التصرف الصحيح إذا فشل جزء من الرحلة؟