The bug where no availability meant always available

A small booking bug, an obvious fix that would have made it worse, and cleaning up state that lives in two places.

Igor Leal Antunes · LinkedIn · Published Nov 9, 2026 · Updated Nov 10, 2026

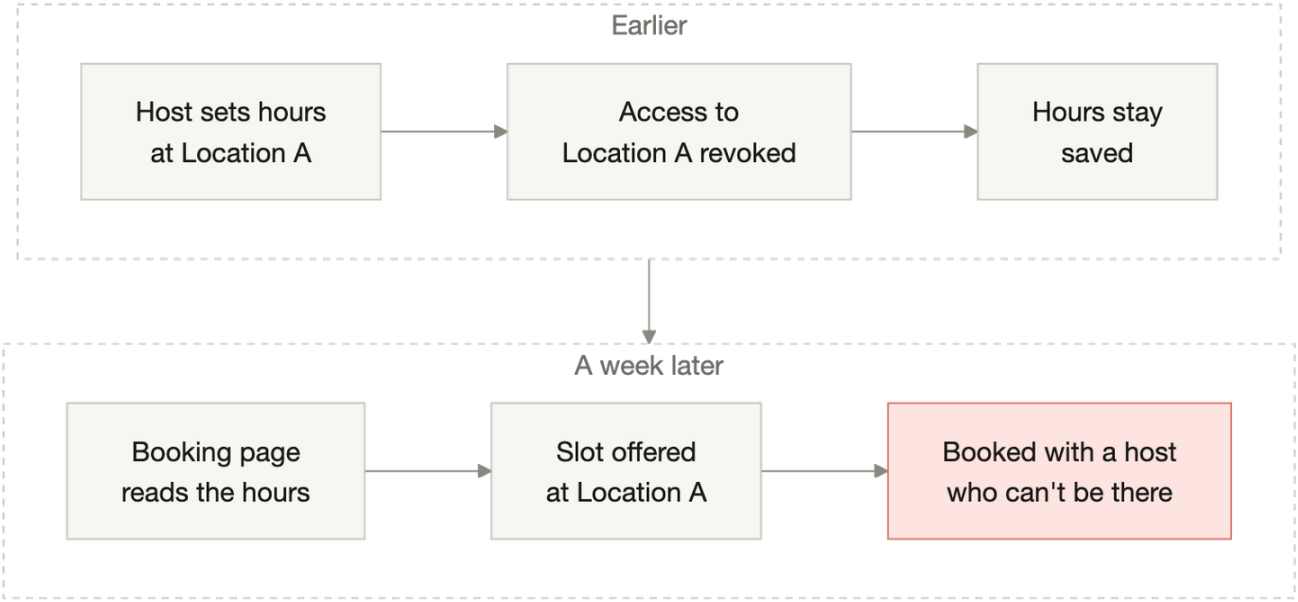
> **The takeaway**
>
> Removing a host's access to a location left their availability for that location behind, and the booking page kept offering those orphan hours. Hiding them wasn't enough on its own, because a host with no hours at all counts as free whenever their calendar is empty. The fix ignores the orphan rows and shows no times when a required host can't reach the location.

A booking that shouldn't exist

Last week I spent some time debugging an interesting bug in an interview booking system. The fix itself was small. What made it interesting was that the first fix I reached for would have made things worse.

Let's imagine how the system is used. Interviewers, the hosts, set the hours they're free at each location they work from. Candidates open a booking page, pick a slot, and the interview lands on the host's calendar at that location.

The bug: a host could be booked at a location they no longer had access to. A host sets their Tuesday hours at a location, their access to it is revoked, and a week later a candidate books a Tuesday slot with them there anyway. Availability was saved per location, and nothing cleaned it up when the access went away. Revoking access is rare, so this affected only a small share of bookings, but often enough to keep coming back.



The access goes away, but the hours stay. Nothing on the booking path checked whether the host could still use that location.

Empty rules mean an open calendar

The obvious fix was to filter those hours out: skip any hours saved for a location the host can no longer access. It's one extra condition on the query:

```
// The obvious fix: one line, green tests, home early.
// Don't. 
$availabilities = Availability::query()
    ->whereIn('host_id', $hostIds)
    ->whereHas('host.locations', fn (Builder $query) => $query
        ->whereColumn('locations.id', 'availabilities.location_id'))
    ->get();
```

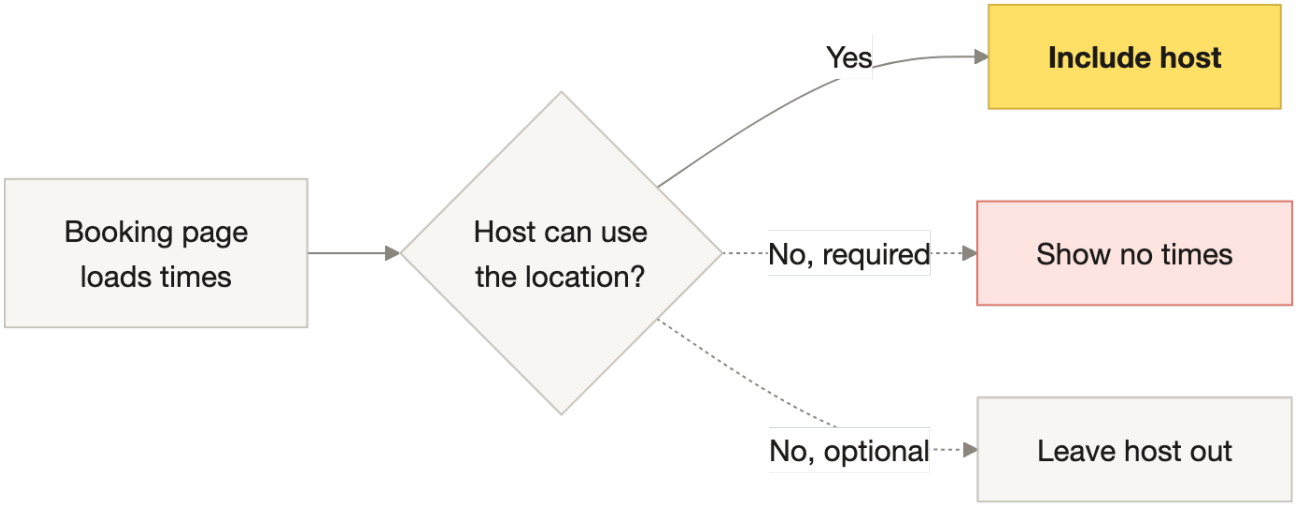
The urge to ship it and call it a day was real.

Before I ship a filter, I check what the next system does with what comes out the other end. Each host's hours live on a calendar service as rules, and it treats a host with no rules as having no limits: free whenever their calendar is empty. Filtering out a host's only hours at that location would have turned "free on Tuesdays" into "free any time".

Show no times instead of wrong times

The fix checks host access before the booking page asks the calendar service for times. If a required host can no longer use the location, the booking page shows no times for that event. An empty page is a visible problem: someone notices and fixes the host list. A wrong slot stays hidden until a candidate shows up at a location their host can't use.

Optional hosts are handled differently, since the event can go ahead without them. The booking step checks again too, because the page may have loaded before the host lost access.



A host without access never reaches the calendar service. Required hosts block the event; optional hosts drop out.

Prune the leftovers on both sides

Saving hours for a location the host can't access is now rejected, but revoking access still leaves rows behind. For those I used Laravel's `Prunable` trait: the model defines which rows are stale, and the scheduler deletes them.

```php
class Availability extends Model
{
    use Prunable;

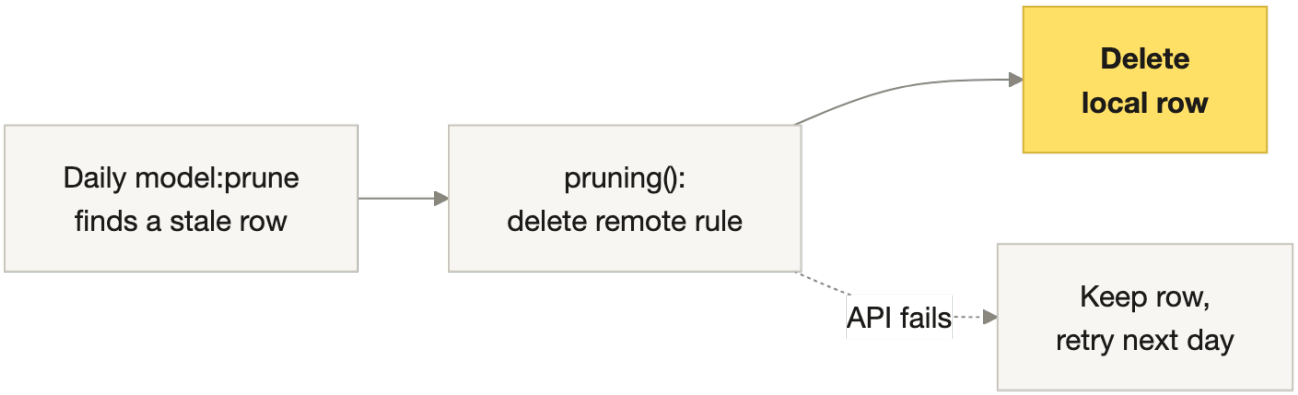
    public function prunable(): Builder
    {
        return static::query()
            ->whereNotNull('location_id')
            ->whereDoesntHave('host.locations', fn (Builder $query) => $query
                ->whereColumn('locations.id', 'availabilities.location_id'));
    }
}
```

`prunable()` returns an ordinary query, so the definition of "stale" lives in one place: any availability tied to a location its host is no longer linked to.

Deleting the row wasn't enough. Each availability has a matching rule stored on the calendar service, and a rule with no row behind it is the same orphan in a different place. The trait calls a `pruning()` hook before each delete, so I override it to remove the remote rule first:

```php
protected function pruning(): void
{
    app(CalendarClient::class)->deleteRule($this->external_rule_id);
}
```

Removing a host's last rule creates exactly the empty-rules case from earlier. It's safe here only because the access check keeps that host away from the calendar service entirely.



The remote rule goes first. If that call fails, the local row stays and the next run tries again.

Two details make this safe to run every day. `pruneAll()` catches and reports exceptions one row at a time, so a failed API call skips that row instead of stopping the run. And the client treats a 404 as done, so a row whose rule is already gone still gets cleaned up.

This is also why it's `Prunable` and not `MassPrunable`:

Only one of Laravel's two pruning traits runs code per row

Trait	How it deletes	Calls `pruning()`
`MassPrunable`	Bulk delete queries, no models loaded	No
`Prunable`	Loads and deletes each model	Yes

Only `Prunable` gives each row the chance to clean up its remote rule.

Laravel finds the model on its own. Once `model:prune` is on the schedule, it scans `app/Models` and prunes every class that uses `Prunable` or `MassPrunable`, so a new model only needs the trait.

A pruned row is gone from both sides and can't be brought back, so the safety net is the dry run. `--pretend` prints how many rows each model would lose and deletes nothing:

```
php artisan model:prune --pretend
```

A daily run still leaves a gap between revoking access and the cleanup. Bookings are safe in that window, since the access check reads live access every time, but I didn't want stale rows and remote rules sitting around for a day. So revoking access now dispatches a queued job that prunes that host's hours for that location right away. An observer on the host's location link is a tidy way to wire it up.

The daily prune stays as the safety net for anything the observer misses (a mass delete, a raw query, or any other change that skips the normal Eloquent model events).

Slow down, but not every time

Holding off on the obvious fix paid off here. The one-line filter would have shipped in minutes and turned a rare bug into a common one. Most of the time the obvious fix is the right one, and telling the two apart comes from knowing the system.

The harder part was the state living in two places. The database and the calendar service each kept their own copy of a host's hours, and neither knew when the other changed. Whenever I connect a third-party service, I ask the same question early: when this row goes away, what happens to its copy on the other side?

When state has to stay in sync across third parties, I usually reach for webhooks, so the other side tells us when something changes instead of us finding out on the next cleanup. They're quick to add (in Laravel, a route that verifies the signature and queues a job), and the delivery model is well understood (Stripe's docs are a good example):

- providers usually retry with backoff when our endpoint fails, so a deploy or a short outage doesn't lose events,
- the same event can arrive more than once, so the handler has to be idempotent, usually by storing the event ID and skipping duplicates,
- events can arrive out of order, so the handler refetches the current state from the API instead of trusting the order of the payloads.

Read online: https://igorlealantunes.com/articles/no-availability-meant-always-available