The Doctor Is In

Using checkups to find bugs in production

@rofreg

Hey there everybody!

Ryan Laughlin

@rofreg

My name is Ryan Laughlin, or @rofreg if you know me from the internet. I'm one of the cofounders of Splitwise, which is an app for splitting expenses with other people.

@rofreg

Right out the gate, I want to say that I am super excited to be giving this talk. This is not only my first RailsConf, this is the first conference talk I have EVER given!

And in particular, I'm really excited to give THIS talk. I'm really excited to talk about what I think is a important gap in the way that we think about testing and debugging our applications, both in the Rails community and beyond.

@rofreg

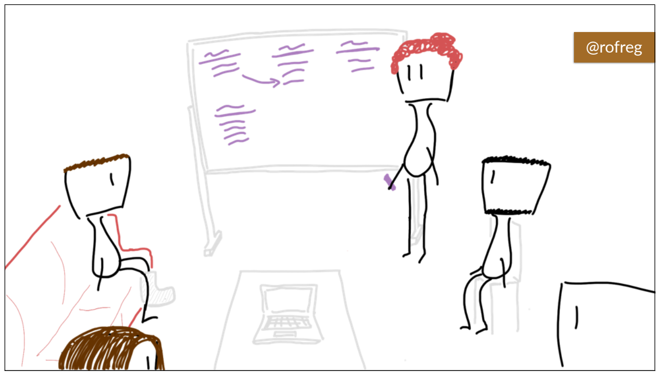
http://rofreg.com/talks

If you want to follow this talk at your own pace, or if you want to look back at it later, all the slides are up at http://rofreg.com/talks.

@rofreg

Let’s go!

So with that said, let’s get right into it!



Let’s say you’re building a new feature for your app. You plan out all the details about how the feature should work, and you think through all the edge cases and all the possible issues that you might encounter.

Then you go and you actually write the code.
You might write a suite of tests to make sure that the feature works as intended.
You might do code review, so that your fellow developers can help you spot a potential bugs and fix them.
You might have a staging server, or even a formal QA process, in order to catch as many bugs as possible before something gets shipped.
You take your time and you fix every bug that you can find.
Now it's time to deploy your code to production, for the whole world to see.

@rofreg

Success!

Aaaaaand congratulations! You're all done!

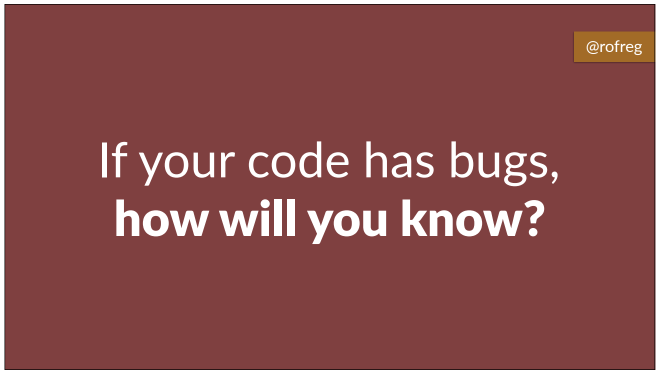
@rofreg

Success!

(probably...)

Well, I mean, MAYBE you're done. I don't know about you, but personally, I am someone who occasionally makes mistakes. And quite often when I make an update to an app, I will miss some minor bug or another, and I'll end up deploying that bug to production.

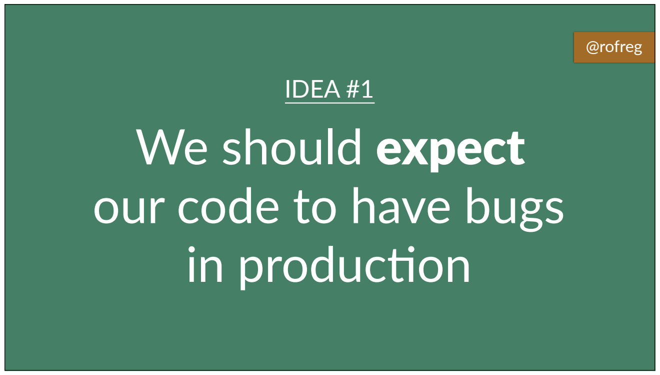
I mean, I've worked on the same one Rails app for 7 years now, and I have probably shipped hundreds of bugs in that time.


@rofreg

If your code has bugs,
how will you know?

And so this is a question that I ask myself a lot.
If my code has bugs in it, how would I even know?

And I want to be specific here.
Note that I am NOT asking how do we prevent bugs from happening in the first place.
I am asking how we DETECT those bugs WHEN they happen.

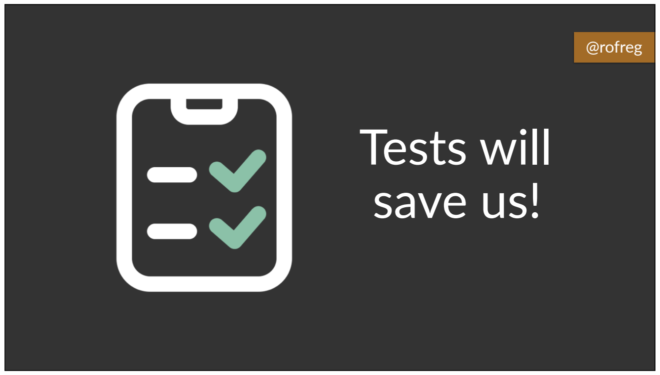

@rofreg

IDEA #1

We should **expect** our code to have bugs in production

This is the first thing that I really want to hammer home.
We should EXPECT to make mistakes, and we should EXPECT to make mistakes in production.

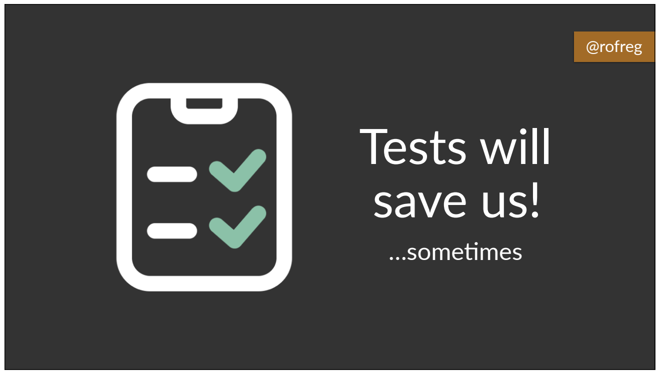
A quick show of hands: raise your hand if you have ever deployed something into production.
Okay, now keep your hand raised if you've ever deployed something with a bug in it. Everybody, right? Look around for a second. There are a lot of really good engineers in this room.
The best engineers that I know have ALL made these kinds of mistakes, and that's NOT something to be afraid of or ashamed of. It's part of being a engineer. Making a mistake is a chance to learn and to grow.
(You can put your hands down now.)


@rofreg

Tests will save us!

Now, you might think: this is what testing is for! Tests catch bugs so that we can fix those bugs before we ship!

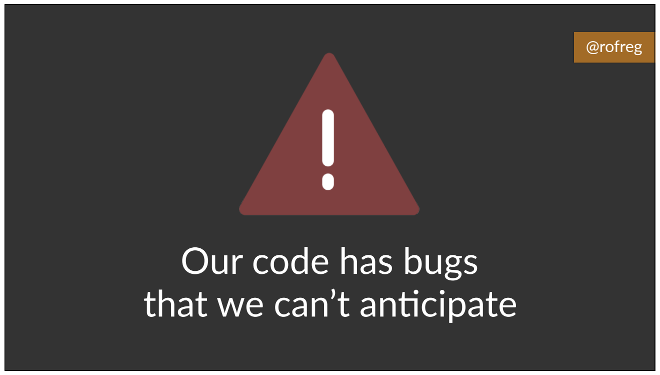
And testing IS a really important step in that process. Tests are really good at ensuring that our code generally works as expected, and they're really good at protecting our code against REGRESSIONS when we make updates.


@rofreg

Tests will save us!

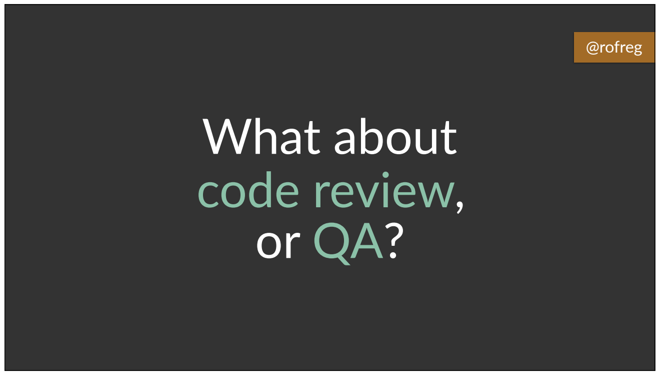
...sometimes

But tests don't catch everything.
In fact, it's sort of tautologically impossible for tests to catch everything!


@rofreg

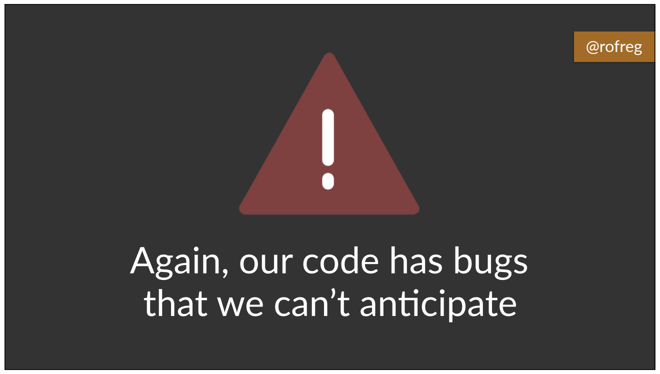
Our code has bugs that we can't anticipate

Because WE are the ones who write the tests, and most of the tests that we write are NOT exhaustive. They only test a handful of cases.
And so if there's an important edge case that we didn't think about, then there may not be any test for that edge case.


@rofreg

What about code review, or QA?

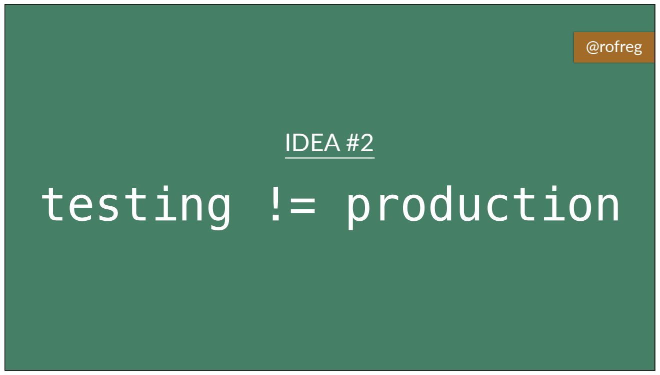
Now, you can improve your chances by including other people in your pre-release process, whether it's via code reviews or QA.
Other people can help you spot issues and problems that you might miss yourself.
And this is a super important part of development in my experience – 2 heads are almost always better than 1.



@rofreg

Again, our code has bugs that we can't anticipate

But again, it ultimately has the same limitations.
Even a room full of very smart people are occasionally going to miss something.
Especially because it's hard to hold an entire system in your head, and to think about how all the different parts of your app might interact with each other.

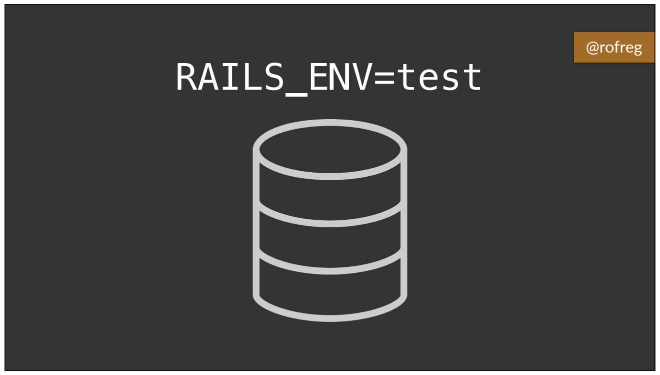


@rofreg

IDEA #2

```
testing != production
```

And that brings me to point #2!
Which is that your production environment is UNIQUE.
Your production environment is DIFFERENT than your test environment, or your development environment, or even your staging environment, and that means that you may have bugs that are UNIQUE to production.



@rofreg

```
RAILS_ENV=test
```

Here's one quick example.
If your app uses a database, I bet that most or all of your tests assume that the database is empty at the start of the test, with no pre-existing data.

```
RAILS_ENV=production
```

@rofreg

But that is NOT what your app experiences in production!
In production, you're working with months or years of existing data, and that can lead to edge cases that you might completely overlook in your test environment.

And that's just ONE way that those two environments differ.
There are always going to be differences between your local environment and your production environment, no matter how much effort you put into making them the same.

@rofreg

IDEA #3

We need to **monitor** our production environment

So if we know we're going to have bugs, and we know that production is a unique environment, then it's logical that we should be on the lookout for bugs that happen SPECIFICALLY in production. And that means that we need to monitor our production environment.

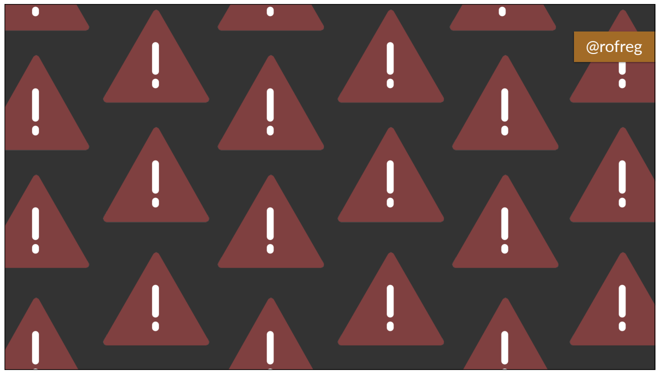
There are a few existing, standard tools for doing this, but they're not perfect, and I think they're a bit incomplete.

Exception reporting

@rofreg

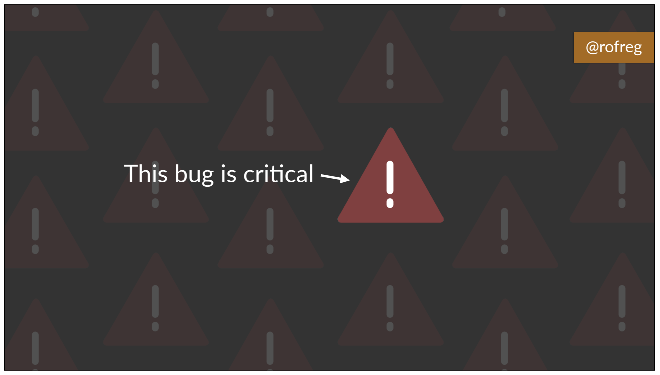
For a lot of apps, the first line of defense is production is exception reporting. Something like Rollbar, or Sentry, or Airbrake, or the standalone "exception_notification" gem. These are tools that can send you an alert any time that an unexpected Exception occurs somewhere in your app.

And this is great, right? If our app explodes in some unexpected way, we need to know!
But there are a few really big weaknesses to exception reporting.



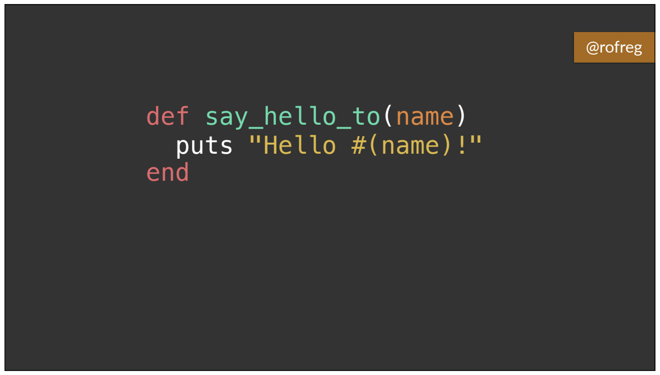
First of all, exception reports can be VERY noisy, especially for big apps. At scale, you will get a lot of errors that are not your fault. People will submit requests with invalid string encodings, or dates that don't exist. People will scan your app for vulnerabilities and submit tons of garbage data. Lots of odd stuff.

And while you can tune your exception reporting to screen out some of these false alarms, in my experience, there will always be new and exciting Exceptions caused by really odd, unimportant user behavior.



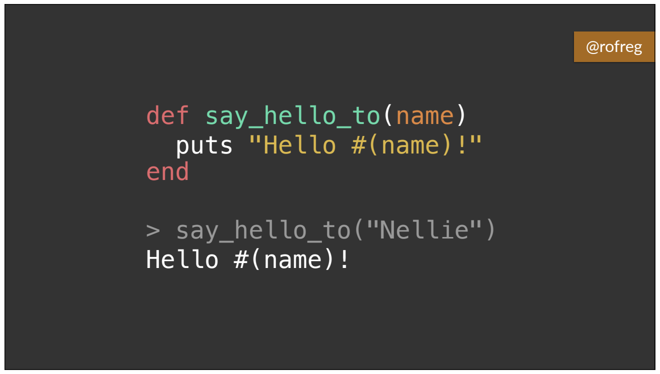
And because there are so many unimportant alerts, that means that the signal-to-noise ratio of exception reporting can sometimes be really low. When you have one critical Exception in the middle of 20 false alarms, it's actually pretty easy to overlook it.

It's like the boy who cried wolf. When something serious actually happens, you might not be paying full attention.



```
def say_hello_to(name)
  puts "Hello #(name)!"
end
```

Also, VERY importantly, exception reporting can only catch Exceptions!
If you're only looking for Exceptions, there are entire categories of bugs that you might miss, where the code DOES run without crashing…



```
def say_hello_to(name)
  puts "Hello #(name)!"
end

> say_hello_to("Nellie")
Hello #(name)!
```

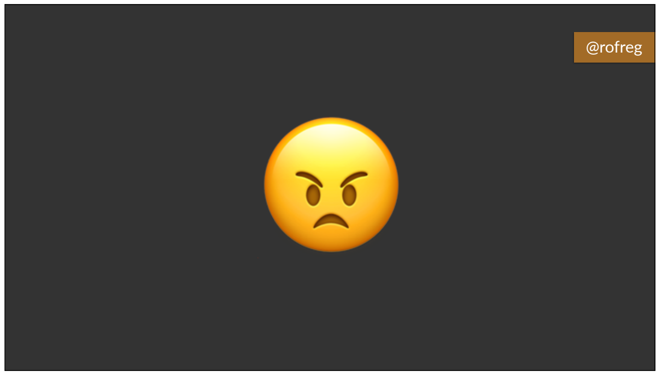
…but it returns the wrong result.

In this case, we have a method that's supposed to print a person's name, but because of a typo, it prints the wrong thing.

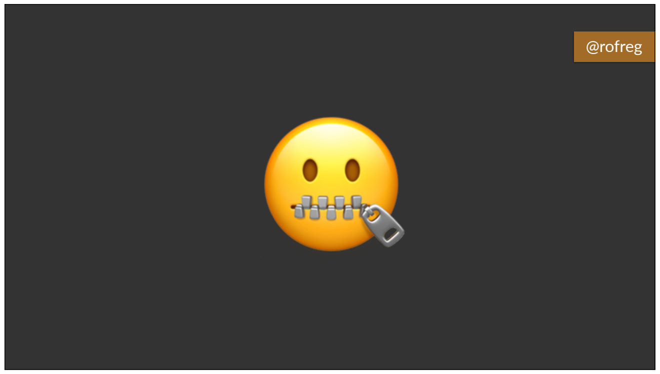
It's surprisingly easy for this kind of issue to slip by, because it's not throwing an Exception that would call attention to itself.

Bug reports

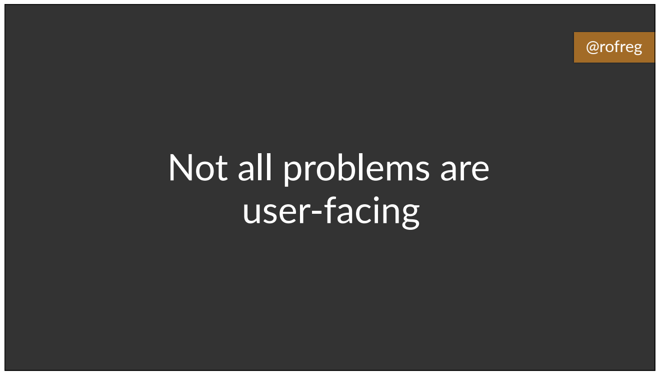
Besides exception reporting, the last line of defense in production is usually bug reports that come directly from you users.
If something is wrong enough with your app, your users WILL probably tell you about it.



But of course, there are big problems with this too.
First of all, it's a horrible experience. Bugs make people frustrated and angry and confused. It makes people lose trust in your app. No one likes using buggy software.



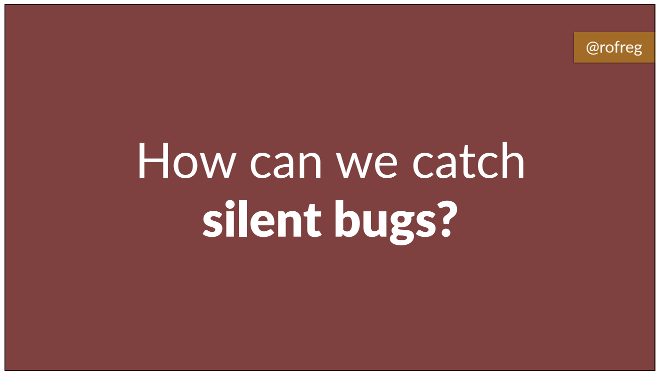
Second of all, a lot of people won't bother to report issues. It takes time to write somebody an email! If I see an obvious problem with your app or your website, 9 out of 10 times I'm just going to leave your site. I'm not necessarily going to send you a nice bug report with repro steps, y'know?



@rofreg

Not all problems are user-facing

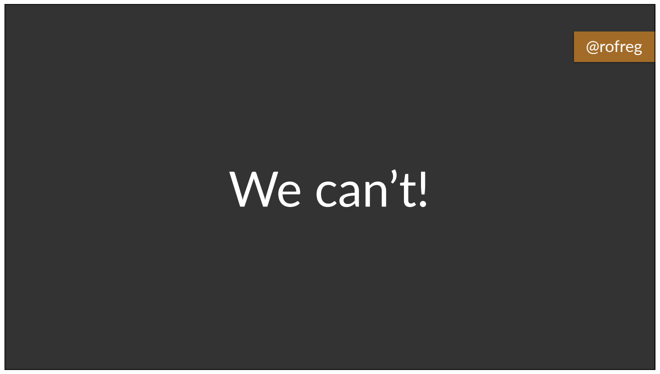
And of course, users can only report the problems that they can actually see. If you have a bug in an internal system, or a background job, or something like that, it's very possible that no one will notice for quite a long time, and that the bug could cause lots of damage before anyone even knows its there.



@rofreg

How can we catch **silent bugs?**

So if something wasn't caught by testing, or by QA, or by exception reporting, or by a user's bug report, then how the heck are we supposed to know about it? How can we catch silent bugs? And the answer is:


@rofreg

We can't!

We can't! Obviously we can't. We can't fix something that we don't know about.

@rofreg

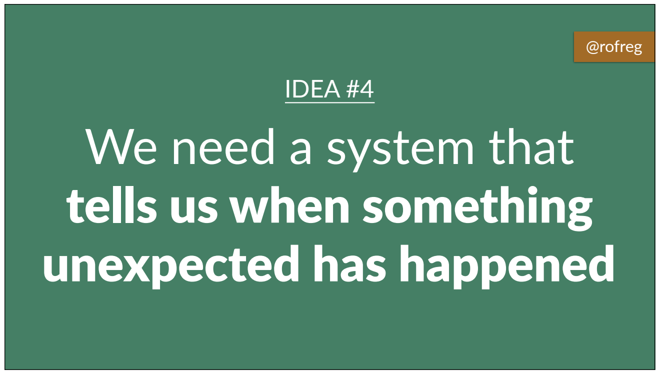
~~How can we catch **silent bugs?**~~

So instead of asking ourselves how to catch "silent bugs", we should ask ourselves this:

@rofreg

How can we turn **silent bugs** into **noisy bugs?**

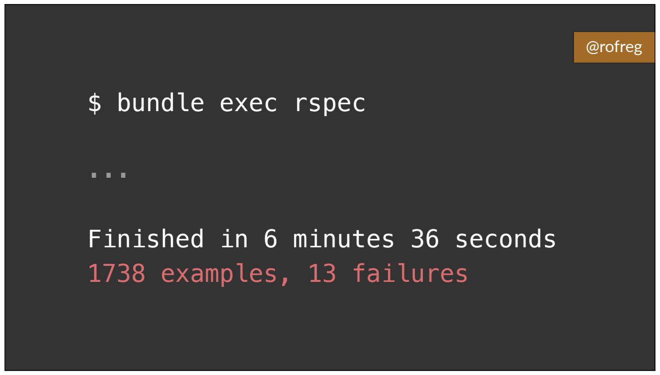
How can we turn "silent bugs" into "noisy bugs"?


@rofreg

IDEA #4

We need a system that **tells us when something unexpected has happened**

We need a system that makes noise. We need a system that tells us when something unexpected happens, so that we can investigate what went wrong.


@rofreg

```
$ bundle exec rspec

...

Finished in 6 minutes 36 seconds
1738 examples, 13 failures
```

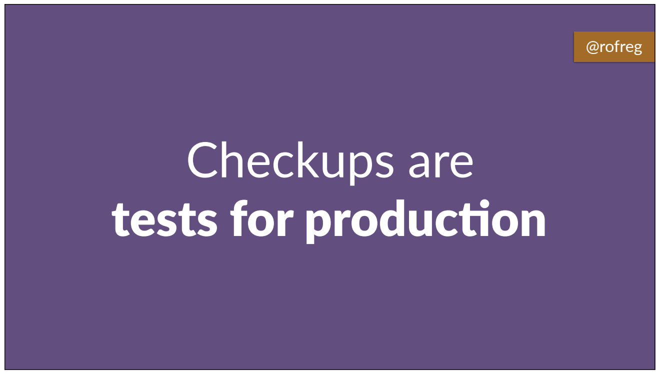
Now, we've gotten pretty good at this in development! This is where test suites really shine, right? When you make a change to your app and suddenly a dozen tests all fail, you know that something unexpected has gone wrong, and you know that you need to look into it further to fix it.

So what would be really useful is something that's LIKE a test suite, but focused on production. Something that doesn't test specific edge cases, but monitors your app for THE EXISTENCE OF ISSUES IN GENERAL.

@rofreg

Time for a checkup!

And that's where checkups come in.

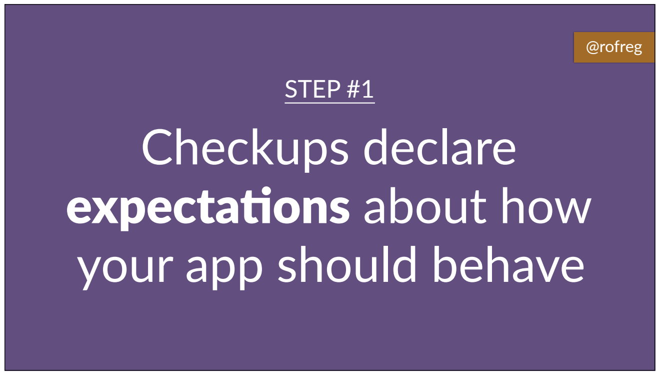

@rofreg

Checkups are
tests for production

Checkups are TESTS FOR PRODUCTION.

The same way that a TEST SUITE tells you when something breaks in DEVELOPMENT, a CHECKUP SUITE tells you when something has broken in PRODUCTION.

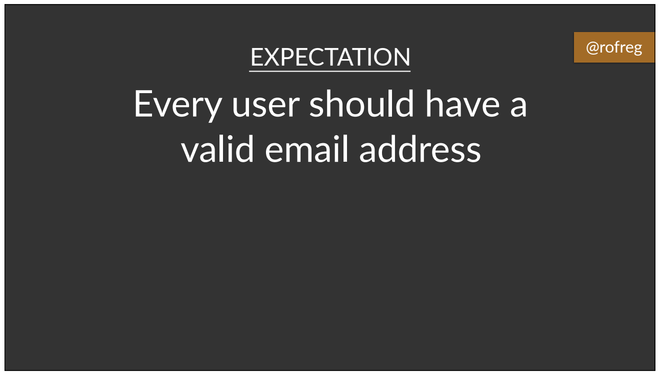
Let me walk you through this.


@rofreg

STEP #1

Checkups declare **expectations** about how your app should behave

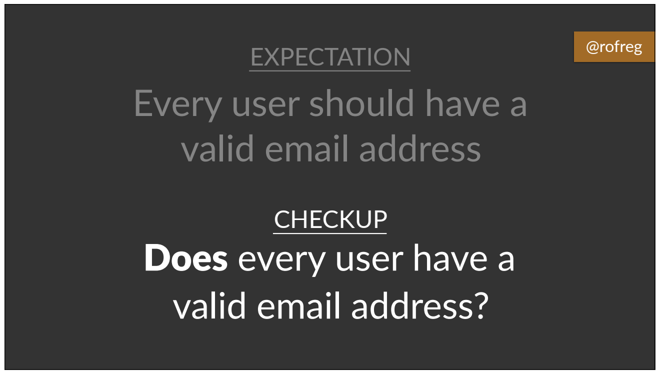
First of all, to write a checkup, we need to declare some EXPECTATIONS about how our app should behave.


EXPECTATION

@rofreg

Every user should have a valid email address

For example: I expect every user to have a valid email address.


@rofreg

EXPECTATION

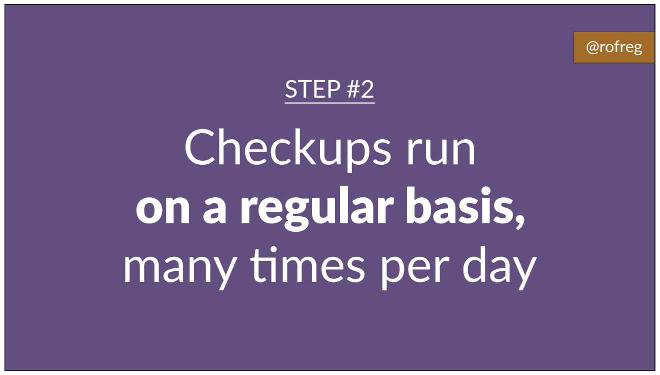
Every user should have a valid email address

CHECKUP

Does every user have a valid email address?

A “checkup” is a block of code that helps help me verify this: DOES every user have a valid email address?

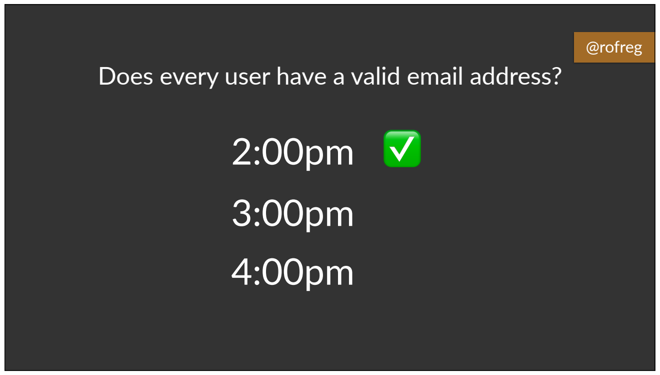
I don’t actually know unless I check.


@rofreg

STEP #2

Checkups run **on a regular basis,** many times per day

This “checkup” then runs on a REGULAR BASIS many times per day, checking to see if anything unusual has happened.


@rofreg

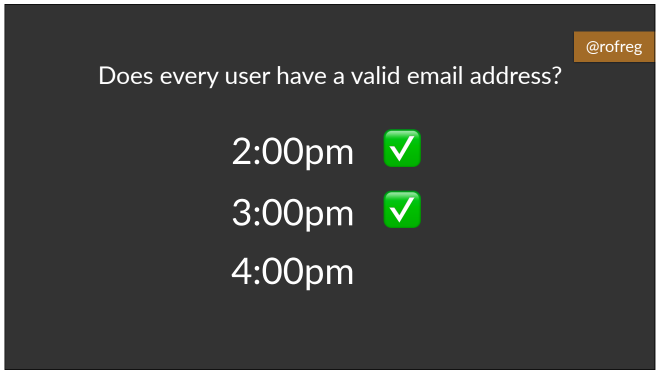
Does every user have a valid email address?

2:00pm ✅

3:00pm

4:00pm

And this is important in production! Because maybe all of my users had valid email addresses at 2pm…

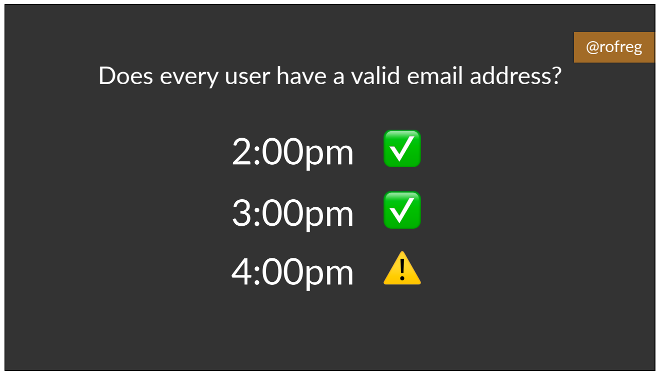


@rofreg

Does every user have a valid email address?

2:00pm ✅
3:00pm ✅
4:00pm

…



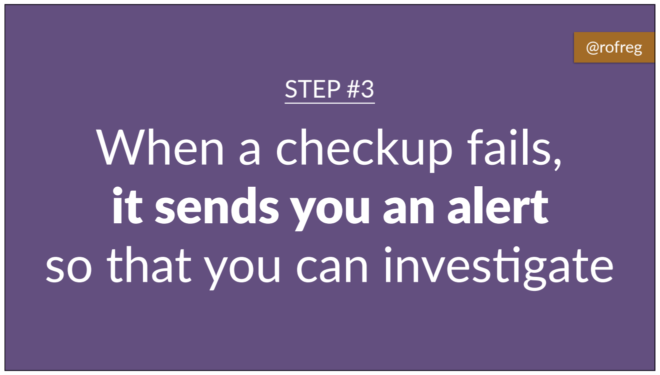
@rofreg

Does every user have a valid email address?

2:00pm ✅
3:00pm ✅
4:00pm ⚠️

…but when I check again a few hours later, that might not be true any more.

Even if I haven't deployed anything new recently, it's possible that a new bug may have bubbled to the surface since my last deploy. A checkup can detect when that happens.

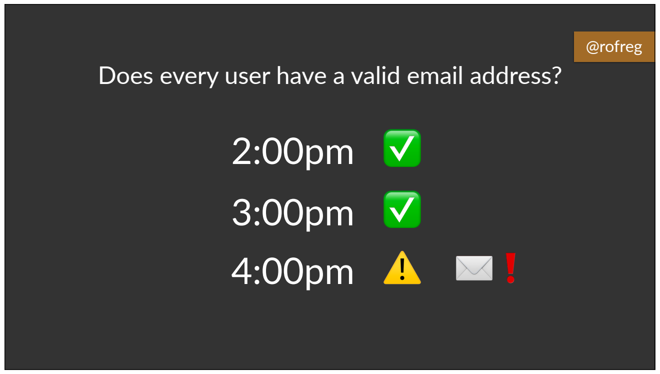


@rofreg

STEP #3

When a checkup fails,
it sends you an alert
so that you can investigate

Finally, if your checkup fails, then you need to be ALERTED so that you can investigate what happened and fix the underlying bug.


@rofreg

Does every user have a valid email address?

2:00pm ✅

3:00pm ✅

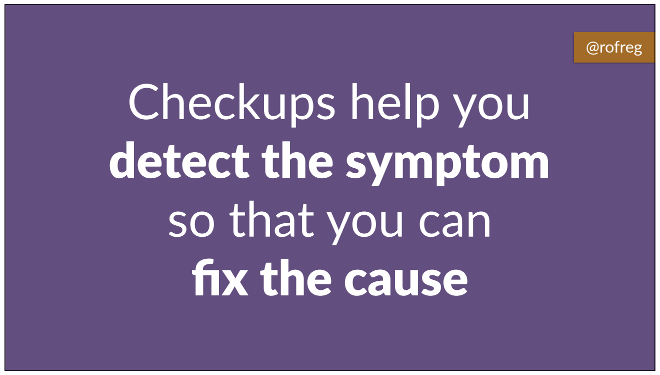
4:00pm ⚠️ ✉️❗

Once you get that alert, you can start to figure out what the problem is.

@rofreg

That's it!

And that's it! That's the whole idea. It's simple, but it's powerful.

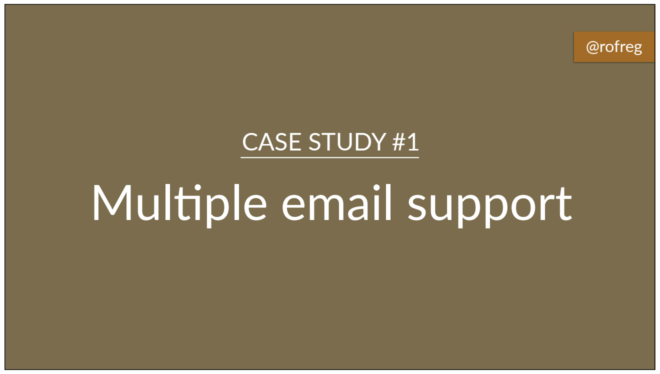

@rofreg

Checkups help you **detect the symptom** so that you can **fix the cause**

Checkups help you detect the SYMPTOM so that you can investigate and fix the CAUSE.

Checkups are the best tool that I know for discovering issues that you didn't even know about.

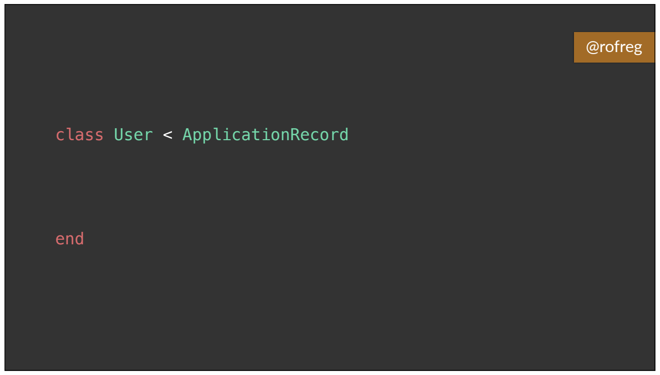
It's just like a checkup with a doctor in real life — if you do it regularly, you can detect problems and fix them before they become bigger issues.


@rofreg

CASE STUDY #1

Multiple email support

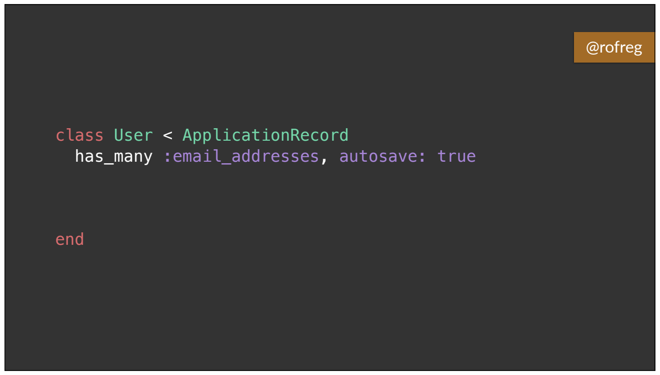
To illustrate, let me give you a simple, real example that we had at Splitwise a few years ago.


@rofreg

```
class User < ApplicationRecord


end
```

At Splitwise, we have a User model.
And for a long time, it was a pretty simple User model. A user had one email address. Not very complicated.


@rofreg

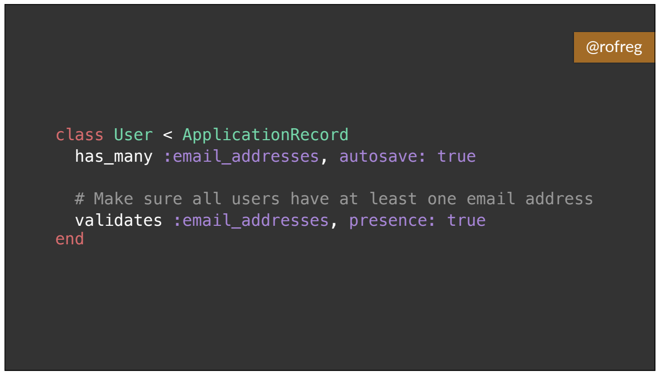
```
class User < ApplicationRecord
  has_many :email_addresses, autosave: true


end
```

And then one day, we decided to add support for multiple email addresses. It seemed like a good, useful feature to add.

So we created a new “EmailAddress” model, and we added a “has_many” relationship so that one User could have many EmailAddresses.

And as we polished up this feature and wrote more tests and such, we realized, oh right, we should make sure that all users have AT LEAST ONE EmailAddress. That's important.



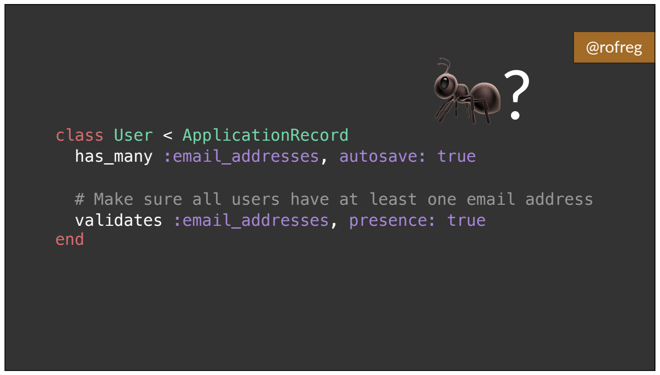
```ruby
class User < ApplicationRecord
  has_many :email_addresses, autosave: true

  # Make sure all users have at least one email address
  validates :email_addresses, presence: true
end
```

So we added a validation, in order to make sure that every user has AT LEAST ONE email address. And it worked! Our tests passed, everything was great. And this is a pretty straightforward-looking bit of code, right? Like, Rails doesn't have a "has_at_least_one" relationship, but this is a pretty clear way to express that idea.

In fact, I actually checked before this talk: if you search Google for "rails has at least one", this is the standard Stack Overflow answer for Rails 4 and up.

And we wrote a whole bunch of tests to make sure that this worked as intended. If you tried to delete a user's last EmailAddress, the validation would not let you continue.

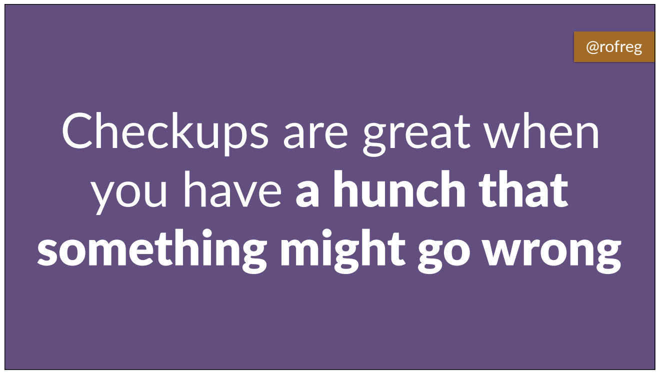


```ruby
class User < ApplicationRecord
  has_many :email_addresses, autosave: true

  # Make sure all users have at least one email address
  validates :email_addresses, presence: true
end
```

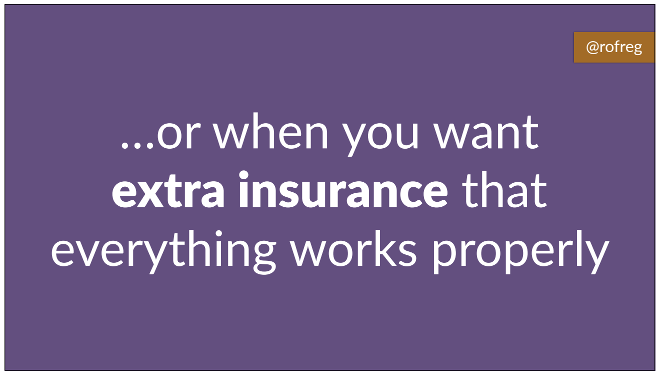
Now, again: we wrote tests for this. We looked at the code, and we thought hard, and we covered all of the edge cases that we could think of.
So I want YOU to look at this code for a few seconds, and I want you to think about what might go wrong.

And let me be specific here:
I am NOT asking you to actually figure out what the specific bug is here.
I'm asking you think about WHAT MIGHT HAPPEN if there IS a bug.
If there IS a bug, HOW will we find out? What is the thing that we will notice?


@rofreg

Checkups are great when you have **a hunch that something might go wrong**

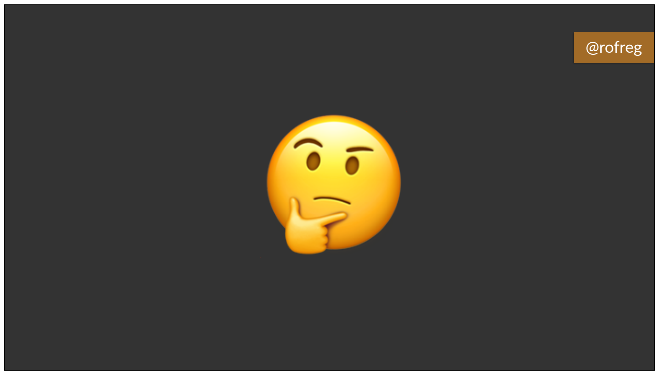
Again, this is where checkups shine. They're great when you think that something might go wrong…


@rofreg

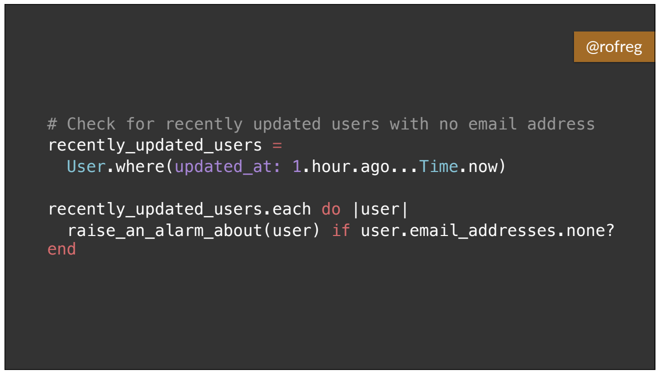
…or when you want **extra insurance** that everything works properly

…or when you just want extra insurance that everything works the way it's supposed to.

This is the same reason that we write tests, right? Like, when I write code, I'm generally pretty confident that it will work properly, but tests help me to have even MORE confidence in my work. Checkups work the same way.



So in this case we thought: hmm, it would be pretty weird if someone ended up with NO email addresses. Maybe we should write a checkup for that! So we wrote this:

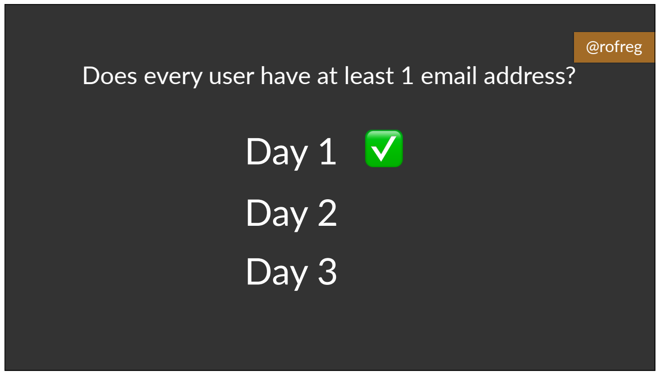

@rofreg

```ruby
# Check for recently updated users with no email address
recently_updated_users =
  User.where(updated_at: 1.hour.ago...Time.now)

recently_updated_users.each do |user|
  raise_an_alarm_about(user) if user.email_addresses.none?
end
```

This is a checkup. It's a very short, very simple little method.
First, we fetch all of the users who have recently updated their accounts.
Then, we iterate through those users and check to see if there are any Users with 0 email addresses.
We run this once per hour.
If we find any Users who DON'T have any email addresses, then this checkup sends an alert to our team so that we can investigate.

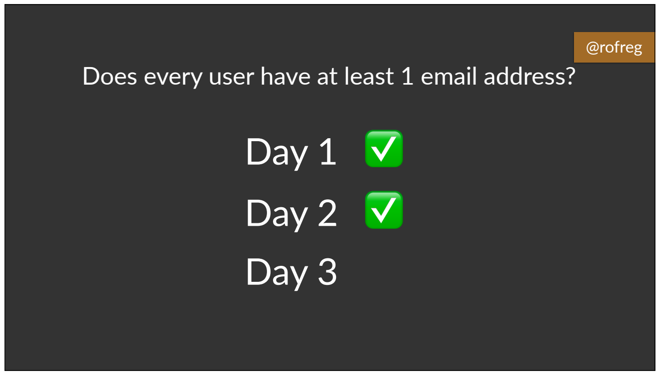
It's 5 lines of code. It's very, very simple.


@rofreg

Does every user have at least 1 email address?

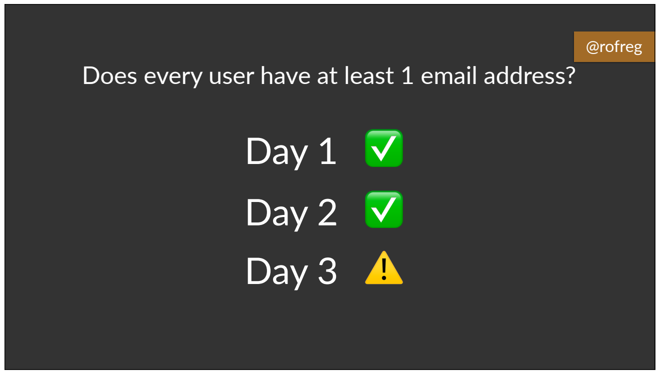
Day 1 ✅
Day 2
Day 3

And so we deployed our new feature, and we included this checkup to make sure that we hadn't missed anything. And for the first day or two, everything was totally great.


@rofreg

Does every user have at least 1 email address?

Day 1 ✅
Day 2 ✅
Day 3



@rofreg

Does every user have at least 1 email address?

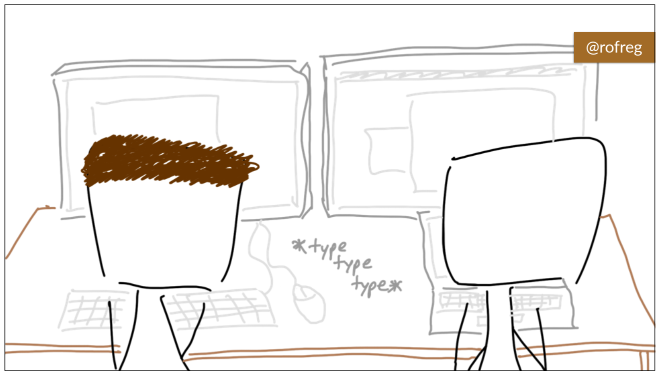
Day 1 ✅

Day 2 ✅

Day 3 ⚠️

But after a few days, sure enough…

…our little checkup sent us an alert. There was a user who somehow ended up with 0 email addresses.

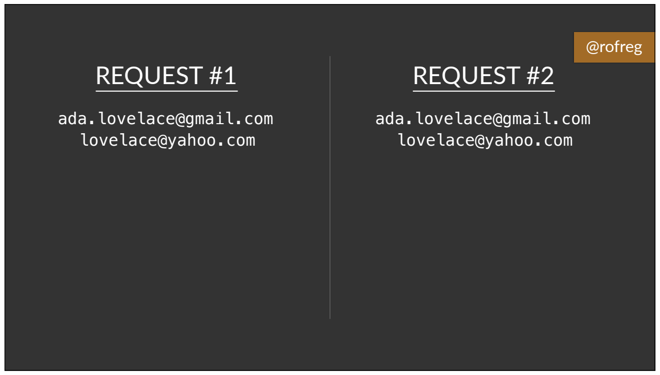


And so we investigated! We looked through our logs for this user, and we realized that they USED to have 2 email addresses, but that they had tried to delete BOTH of those email addresses at the SAME TIME.

@rofreg

Race condition! 🏎️

There was a race condition. One that we hadn't anticipated when we wrote our tests.


@rofreg

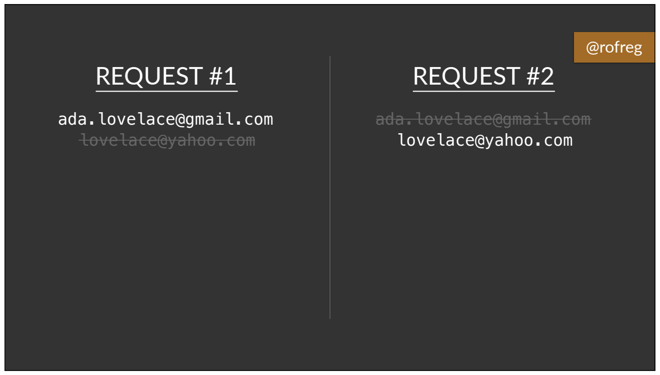
REQUEST #1

ada.lovelace@gmail.com
lovelace@yahoo.com

REQUEST #2

ada.lovelace@gmail.com
lovelace@yahoo.com

See, if you have a user with 2 email addresses…


@rofreg

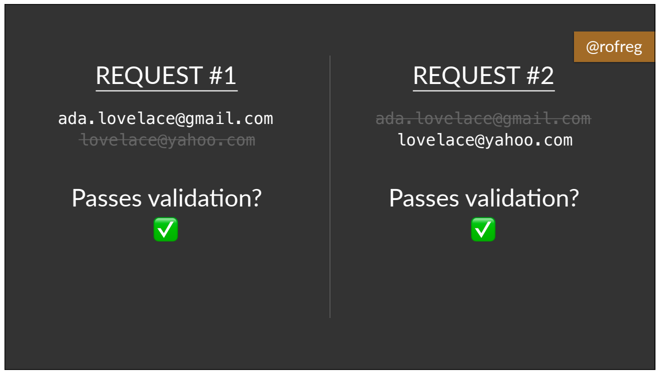
REQUEST #1

ada.lovelace@gmail.com
~~lovelace@yahoo.com~~

REQUEST #2

~~ada.lovelace@gmail.com~~
lovelace@yahoo.com

…and you have TWO different requests that each delete ONE email address…


@rofreg

REQUEST #1

ada.lovelace@gmail.com
~~lovelace@yahoo.com~~

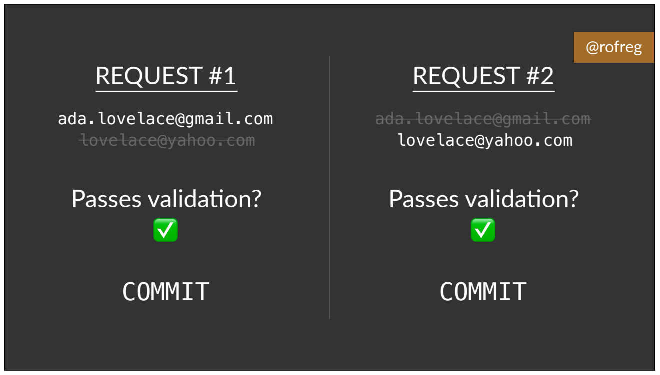
Passes validation?
✅

REQUEST #2

~~ada.lovelace@gmail.com~~
lovelace@yahoo.com

Passes validation?
✅

...then both of those requests will actually pass validation! In request #1, the User still has one email address left, so Rails thinks it's totally valid. The same is true in request #2.


@rofreg

REQUEST #1

ada.lovelace@gmail.com
~~lovelace@yahoo.com~~

Passes validation?
✅

COMMIT

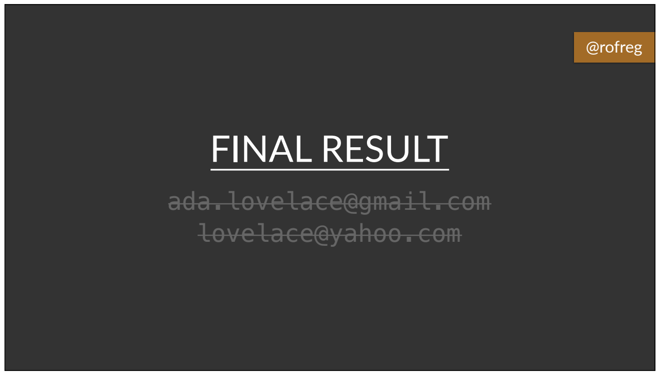
REQUEST #2

~~ada.lovelace@gmail.com~~
lovelace@yahoo.com

Passes validation?
✅

COMMIT

And because it's passed validation, those deleted email addresses then get fully deleted from the database!


@rofreg

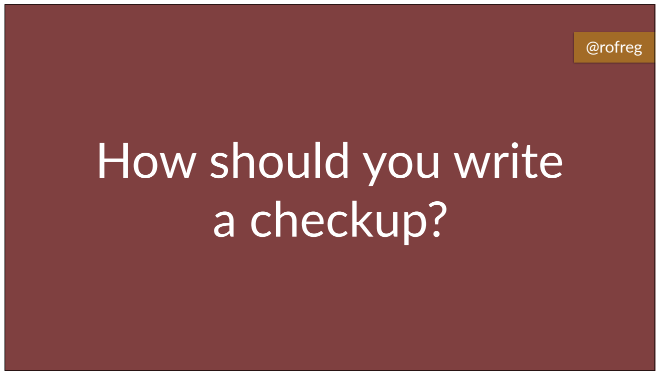
FINAL RESULT

~~ada.lovelace@gmail.com~~
~~lovelace@yahoo.com~~

And you end up with an invalid user with 0 email addresses.

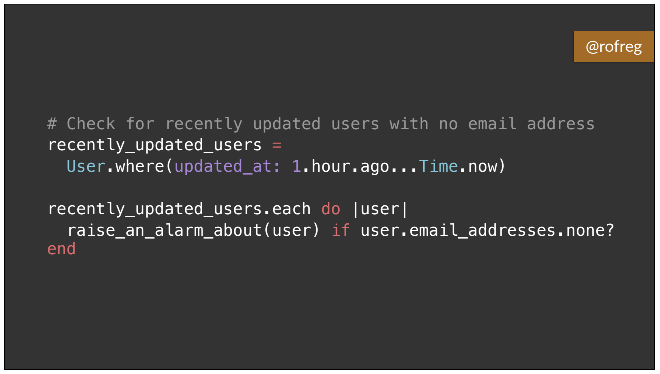
That's obviously a bug! And we had totally missed it. But because we wrote a checkup, that helped us discover this bug as quickly as possible…

…so that we could fix it right away.



How should you write a checkup?

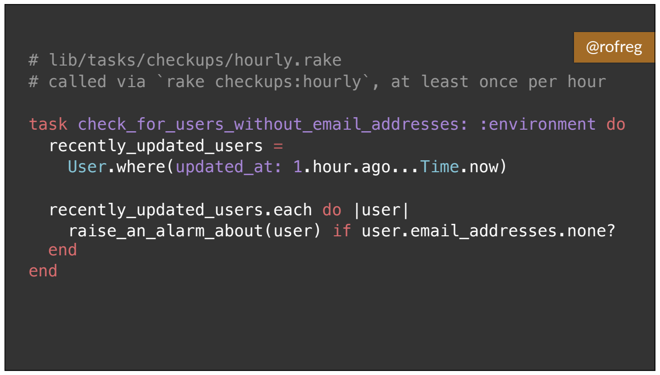
So. How should you write a checkup?


@rofreg

```
# Check for recently updated users with no email address
recently_updated_users =
  User.where(updated_at: 1.hour.ago...Time.now)

recently_updated_users.each do |user|
  raise_an_alarm_about(user) if user.email_addresses.none?
end
```

Well, here's that short little code sample again. And there are a couple of ways that we could finish turning this into a fully-functional checkup.


@rofreg

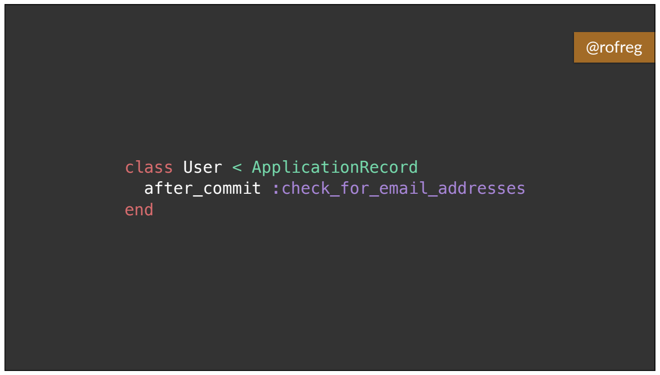
```
# lib/tasks/checkups/hourly.rake
# called via `rake checkups:hourly`, at least once per hour

task check_for_users_without_email_addresses: :environment do
  recently_updated_users =
    User.where(updated_at: 1.hour.ago...Time.now)

  recently_updated_users.each do |user|
    raise_an_alarm_about(user) if user.email_addresses.none?
  end
end
```

One great way is to turn it into a rake task! This is how we write most of our checkups at Splitwise. It's easy to set up a rake task as a recurring cron job, so that it gets called on a regular, repeating basis.

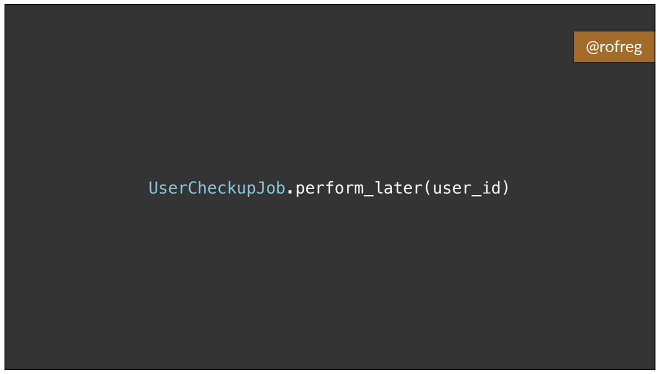
We use Heroku at Splitwise, so we use Heroku Scheduler for this, where it's easy to configure a rake task to get called once per hour, or once per day, or once every 10 minutes.


@rofreg

```
class User < ApplicationRecord
  after_commit :check_for_email_addresses
end
```

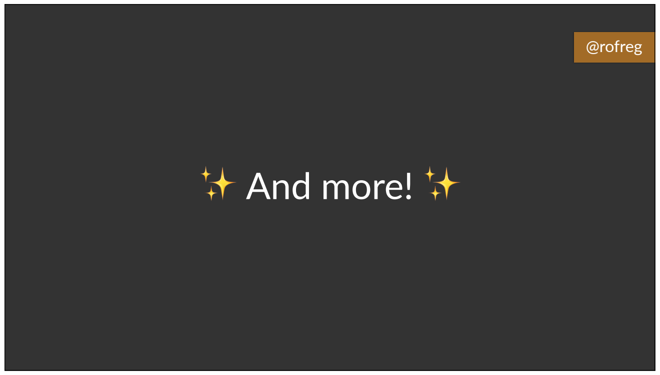
Another good option is as an `after_commit` hook. This is an ActiveRecord callback that executes after your model has been fully written to the database. If you've accidentally written something incorrect to your database, this is an excellent place to catch it.

I should note, this comes at a cost — you're adding overhead to every time you save an ActiveRecord object. That said, it gives you IMMEDIATE feedback about any errors, so it can be a good option if you're writing a checkup about a mission-critical part of your app.


@rofreg

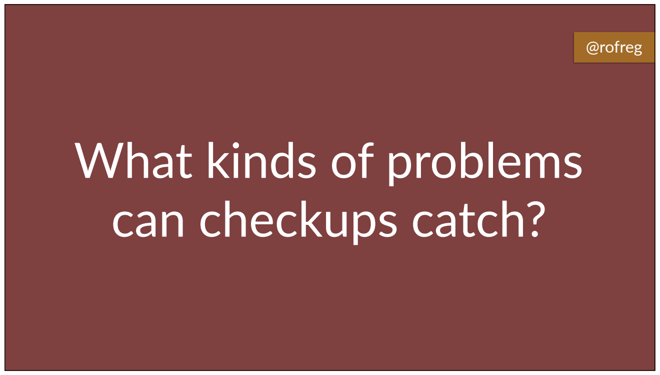
```
UserCheckupJob.perform_later(user_id)
```

You can also kind of split the difference and perform checkups in a background job. This is great way to perform checkups “on demand”, in response to a specific user action, but without slowing down your request too much.


@rofreg

✨ And more! ✨

And honestly, that’s just a start. Checkups are a pretty general idea, and there are a lot of other places that you can use the same concept. For example, I’ve written a few checkups that run inline in controller actions, or in service objects.

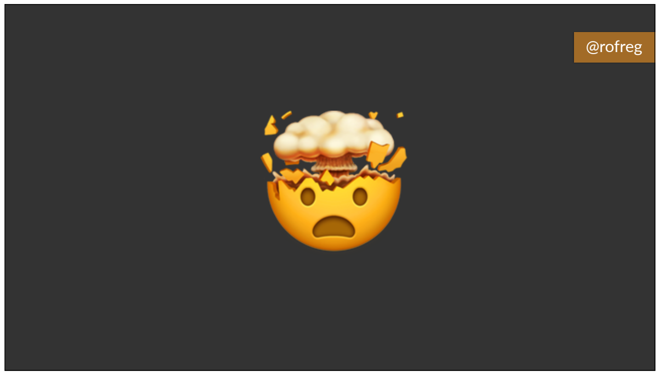

@rofreg

What kinds of problems can checkups catch?

Okay, cool. Different question. When should I write a checkup? What kinds of problems can checkups catch?

Race conditions

Well, as we've already seen, checkups are VERY good at sniffing out race conditions.
I think race conditions are maybe the best example of a problem that is rare in development or testing, but common in production.

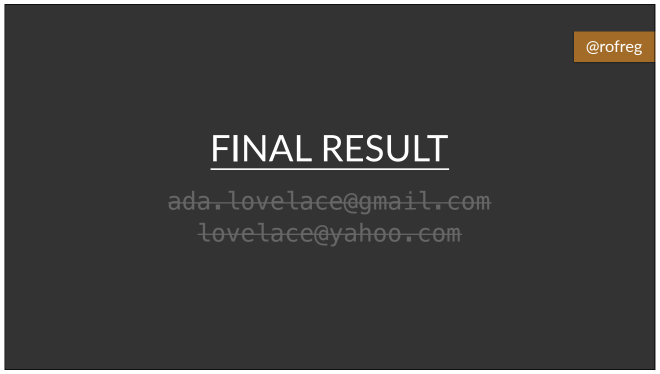


Because if you're like me, you probably find thinking about race conditions really hard! Our brains aren't really built to think in parallel threads. But in production, that's what your app faces all the time.
It's extremely common, not only to see many users trying to use your app at the same time, but to see a SINGLE user trying to use your app from multiple threads at the same time.

Checkups can help you detect when this has caused something weird to happen.

Invalid persisted data

Invalid data is another thing that comes up commonly in production that you really don't see in development. The longer that you run an app in production, the more likely you are to accumulate some weird, malformed, improper records in your data store, whether that's MySQL or Redis or static files in S3.



@rofreg

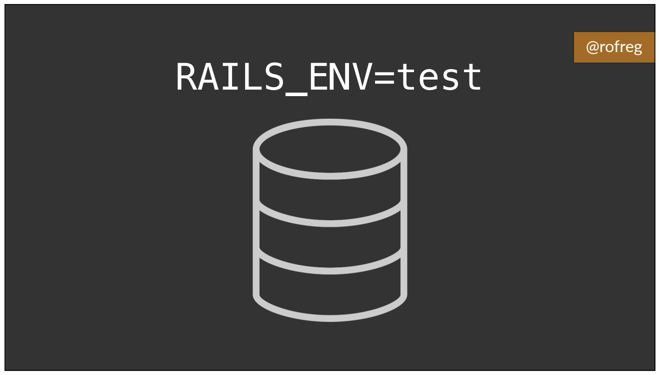
FINAL RESULT

~~ada.lovelace@gmail.com~~
~~lovelace@yahoo.com~~

Here’s a real example. Let’s go back to that “0 email addresses” problem.

So we found this bug, and we wrote some new tests, and then we wrote a fix and we deployed it. The problem was solved.

But! But but but. That only solved the problem going FORWARD. That fix did not solve the EXISTING invalid records that were already living in our production database. There were still several users in our database who didn’t have any email addresses! And we had to hand-fix those records before the issue was fully resolved.



@rofreg

RAILS_ENV=test

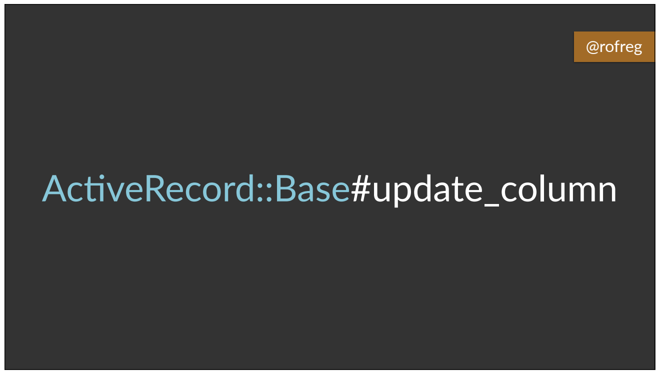
Again, this is the kind of problem that’s super easy to overlook in development and in testing. In those environments, it’s rare to see old or malformed records, because you’re encouraged to clear out your database very regularly.

@rofreg

RAILS_ENV=production

But that’s not true to production. In productions, you might have malformed records that were caused by bugs that happened months ago or even YEARS ago.

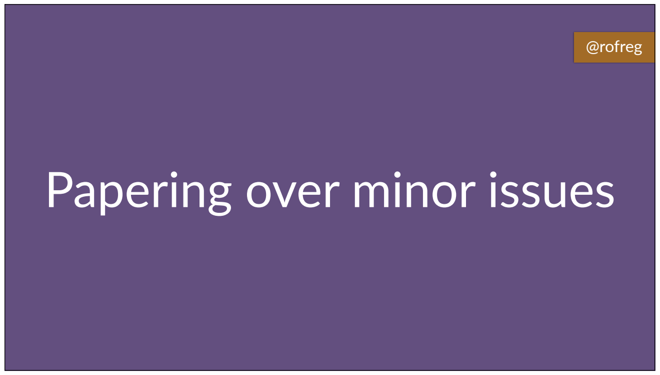
Most of the time, that’s okay — almost every app has a couple of weird bits of data floating around somewhere. But sometimes, that malformed data is REALLY important to catch and to fix, and checkups are a really excellent way to do that.


@rofreg

ActiveRecord::Base#update_column

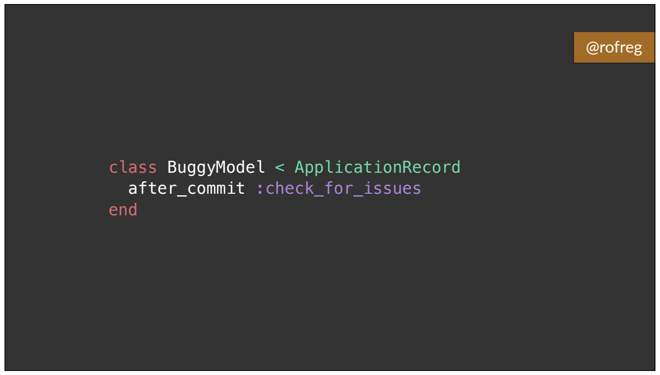
Also, just to check: raise your hand if you've ever used the `update_column` method in ActiveRecord.

Then there might be invalid data in your database! `update_column` skips validations, so there's no guarantee that your data checks out.


@rofreg

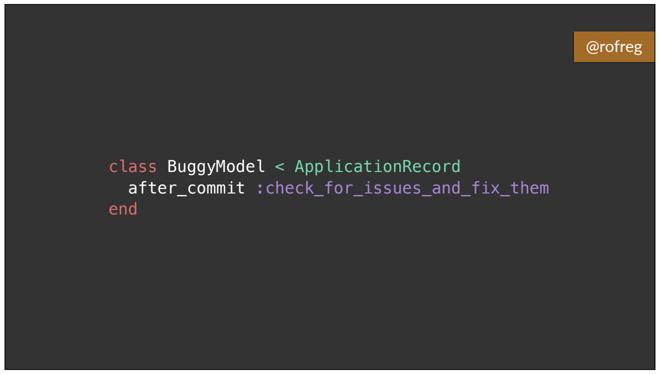
Papering over minor issues

Checkups are also a great tool for when you KNOW there's a bug, but you don't know how to fix it yet.


@rofreg

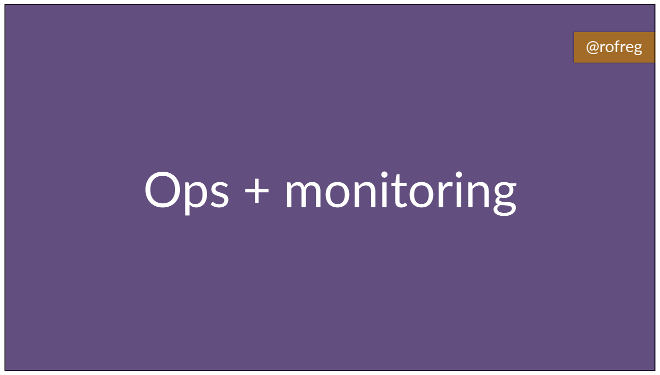
```
class BuggyModel < ApplicationRecord
  after_commit :check_for_issues
end
```

You can use a checkup to gather more diagnostic information about a bug that you don't understand.


@rofreg

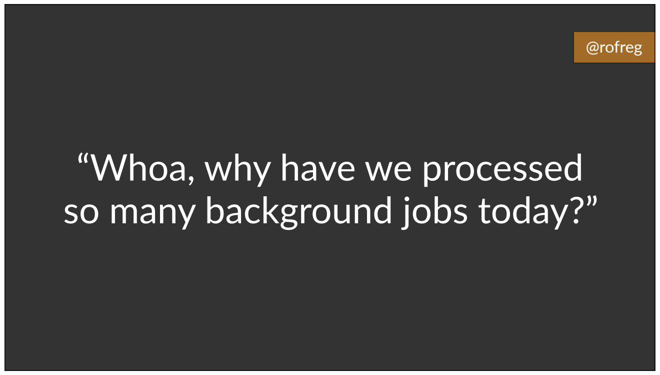
```
class BuggyModel < ApplicationRecord
  after_commit :check_for_issues_and_fix_them
end
```

In some cases, you can even use a checkup to FIX the bug, if there's a programatic way to resolve the issue once its been detected. This can buy you time while you continue to investigate the underlying problem that's causing the bug in the first place.


@rofreg

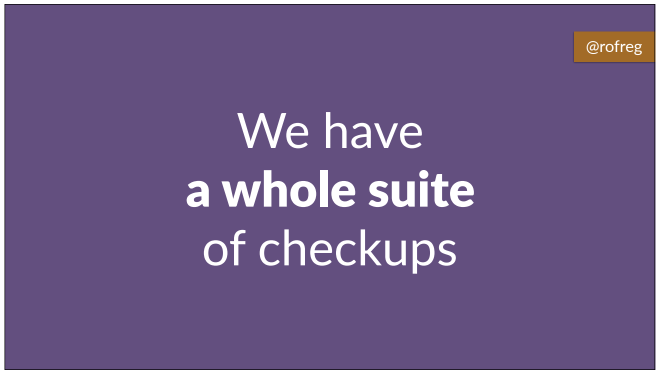
Ops + monitoring

Finally, checkups are really valuable if you're someone who does any kind of ops work in production.
In fact, the whole idea of a "checkup" is basically borrowed from ops. Ops is all about checkups — "Is the site still up?" "Do we have an email backlog?" Checkups are all about evaluating system health RIGHT NOW, and letting you know if something's wrong.


@rofreg

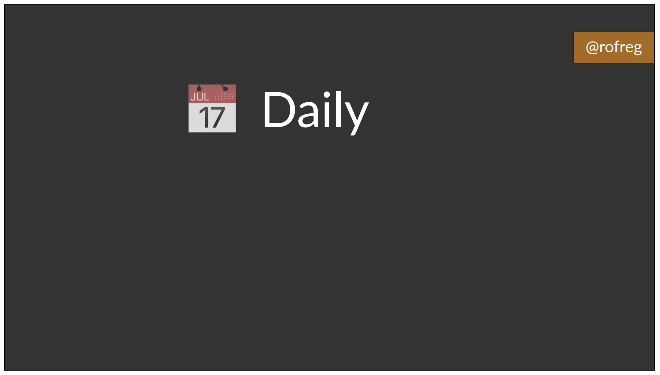
"Whoa, why have we processed so many background jobs today?"

And the thing is, that kind of real-time monitoring can be useful even if you DON'T do ops for your application. Checkups can alert you to unexpected changes in behavior. If your app usually processes 1K background jobs a day, and suddenly it starts processing 100K a day, that COULD be a bug in your code. Maybe there's an infinite loop somewhere that's enqueueing tons of unnecessary jobs by accident.


@rofreg

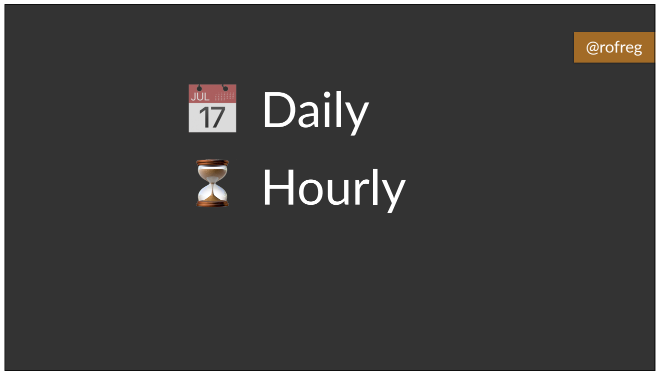
We have
a whole suite
of checkups

At Splitwise, we have a whole suite of checkups like this.


@rofreg

📅 Daily

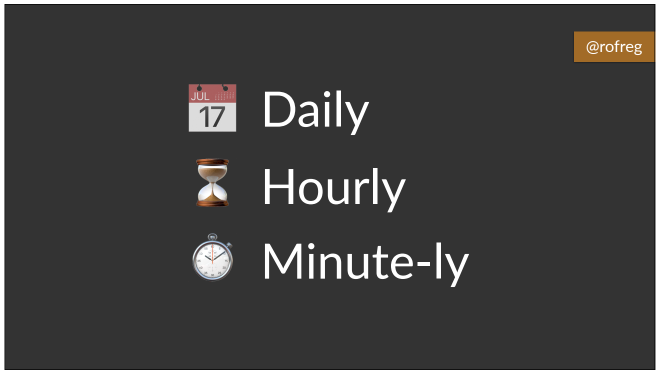
Some of them run daily.


@rofreg

📅 Daily

⏳ Hourly

Some of them run hourly.

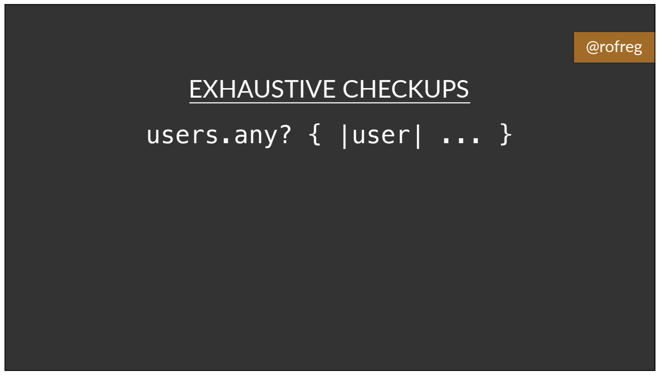

@rofreg

📅 Daily

⏳ Hourly

⏱️ Minute-ly

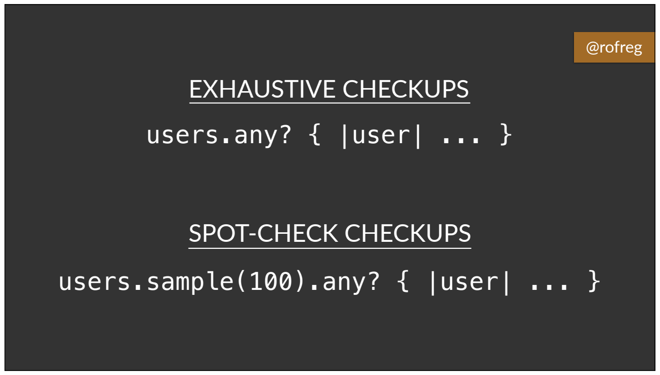
Some run every few minutes.


@rofreg

EXHAUSTIVE CHECKUPS

```
users.any? { |user| ... }
```

Some of our checkups are exhaustive, and check every single record that’s been recently updated, because we don’t want to miss a single problem.


@rofreg

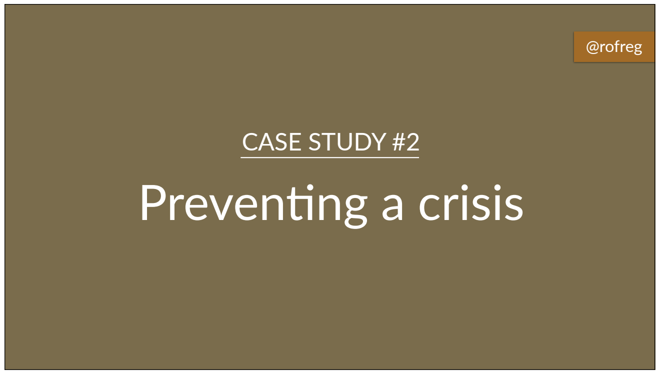
EXHAUSTIVE CHECKUPS

```
users.any? { |user| ... }
```

SPOT-CHECK CHECKUPS

```
users.sample(100).any? { |user| ... }
```

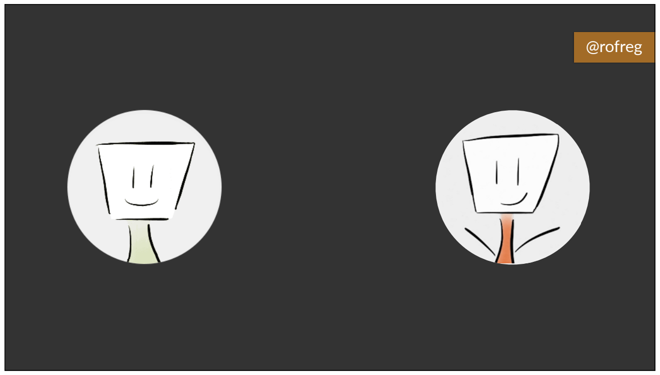
Some of our checkups are just spot-checks — they’re not meant to catch every single error that happens, but they let us know if an error is occurring frequently enough to be a problem.



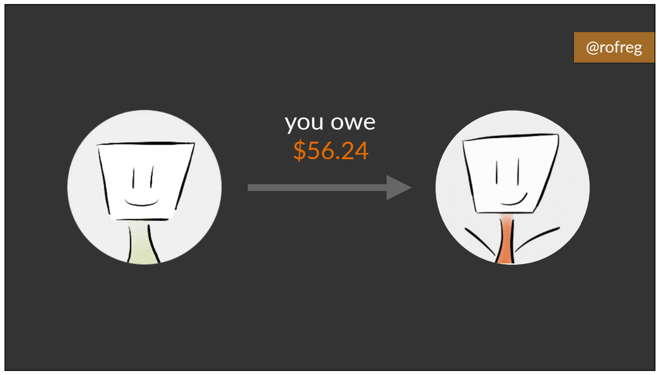
CASE STUDY #2

Preventing a crisis

I want to give you another example where a checkup totally saved my butt in real life, just to drive home how BIG a difference a good checkup can make.

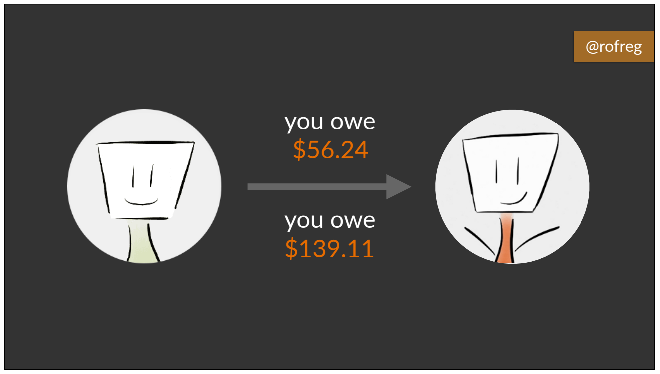


So my company, Splitwise, makes an app that helps people share expenses with each other. And one of the most important things that Splitwise does is calculate your total balance with another person. For example…

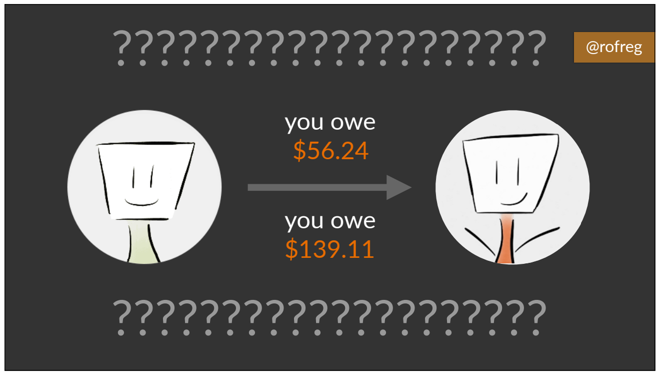


"You owe Ada $56".

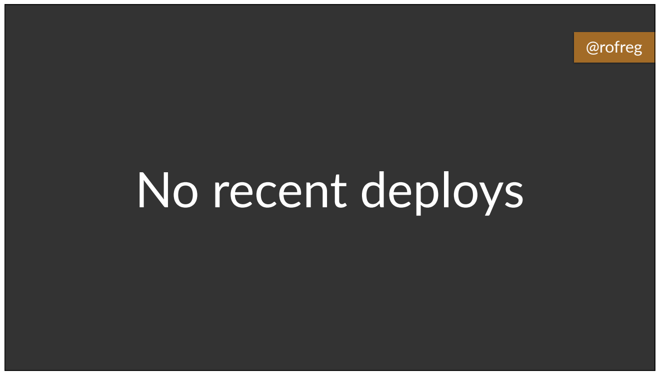
It's really important that we get this calculation right, and we have a bunch of tests to validate that everything adds up correctly.



But one random Tuesday, everything suddenly went wrong.
All of a sudden, our code started returning two different answers for the same calculation.
So when I asked, “How much do I owe Ada?”, our Rails app might reply: “$56”.
But it ALSO might reply: “$139”.
The result was totally random. And I mean random: it was like flipping a coin, where you randomly got one of two possible answers.



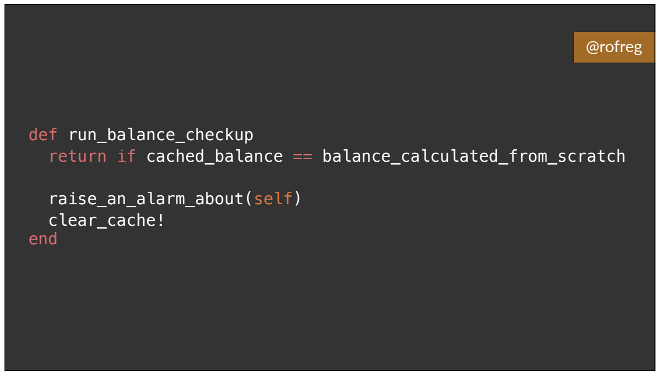
This is obviously a huge, user-facing problem.
It’s massively confusing, and seeing the wrong balance would destroy a user’s trust in our app.
Literally our ONE JOB is to keep track of your expenses for you.
If we can’t do that, then why use Splitwise at all?



No recent deploys

And here’s the kicker: we hadn’t deployed anything new all day.
In fact, we hadn’t touched anything related to this calculation in weeks.
We hadn’t changed ANYTHING. We had no reason to expect that something would go wrong.

But we had a checkup.



```ruby
def run_balance_checkup
  return if cached_balance == balance_calculated_from_scratch

  raise_an_alarm_about(self)
  clear_cache!
end
```

In particular, we had a checkup for our caching layer.

See, we used caching to speed up some of our balance calculations. And this checkup made sure that the cache-dependent version of our `balance` method returned the same result as an alternate implementation that did NOT use the cache. Any time that a person's account was updated, we ran this checkup on their account just a few moments later.

By comparing these two values, we could CONTINUOUSLY VERIFY that our cache-optimized "balance" method was working as expected. And if anything went wrong, we could raise an alarm and clear the cache, getting rid of the incorrect value.

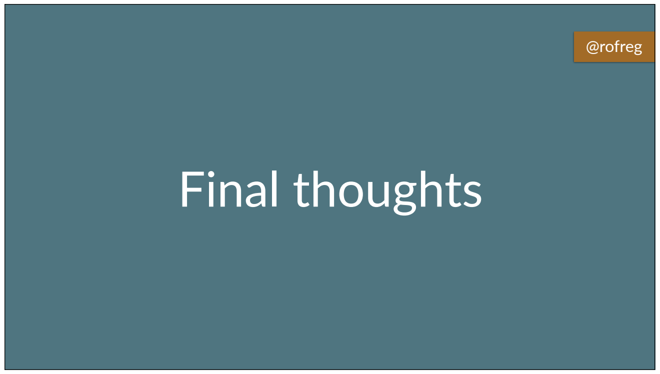
Crisis averted!

Well in this case, that was enough to catch the problem!

Not only did our checkup task ALERT us about the problem immediately, it actually MITIGATED the problem in real time while we figured out the cause and fixed the issue over the next few hours.

In the end, no one even noticed the bug. Instead of thousands of angry users, we had 0 angry users.

(If you're curious, this actually turned out to be a critical infrastructure problem with a third-party caching provider. We detected the problem so fast that we actually alerted THEM about the problem before they had noticed the problem themselves!)



@rofreg

Final thoughts

So. I want to share a few final thoughts about checkups as we wrap up here.

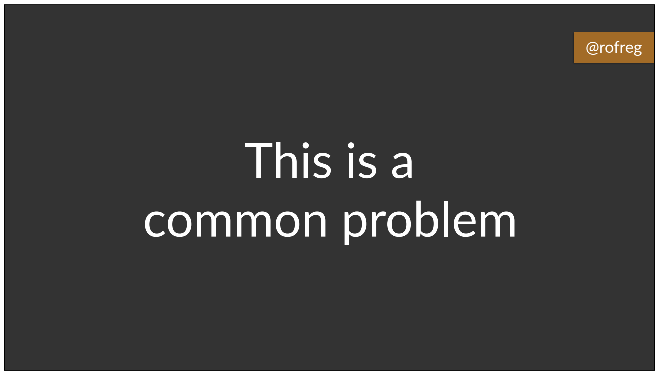
First of all…

@rofreg

This is a work in progress

…this is a work in progress.

Checkups are just an idea that I made up.
As I mentioned at the start, this is my first big public talk.
And this is my first time really trying to spread this idea outside of my own workplace.

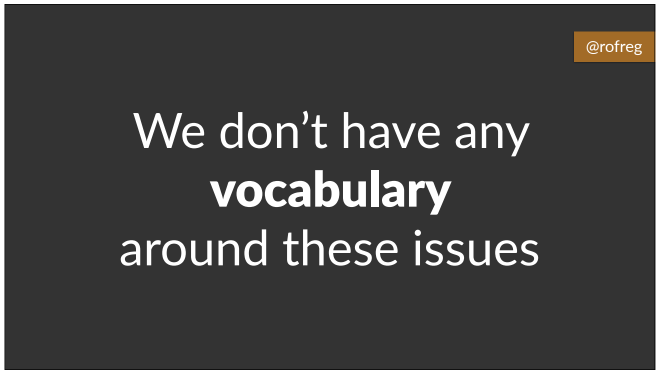


@rofreg

This is a common problem

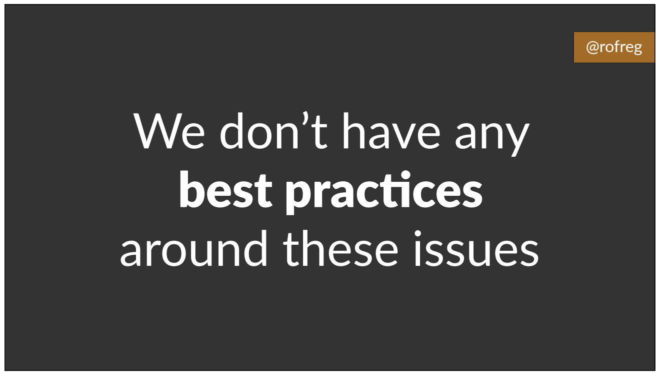
But I know for a fact that this is a common issue. I've talked to friends at a bunch of different companies, and they all have SOMETHING like this — an internal system that double-checks their production environment to make sure certain things haven't exploded.

The problem is, almost no one talks about those systems and those ideas in public. If you're a Rails developer building a new app, they way you learn this stuff is mostly through trial and error. It's not yet part of our standard discussion about how to build an app.


@rofreg

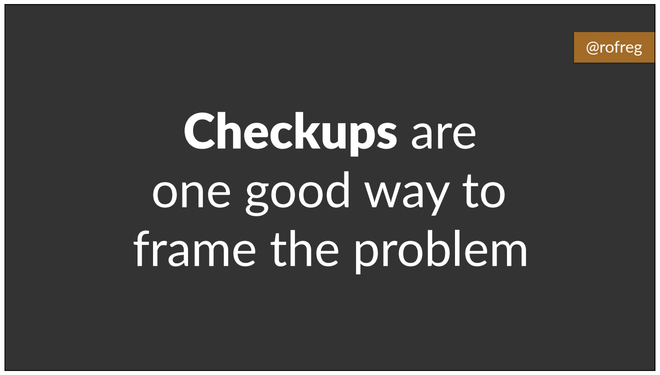
We don't have any **vocabulary** around these issues

And in part, I think that's because we don't have words for it yet. We don't have a pre-existing vocabulary about how to double-check our production systems. And because we don't have a vocabulary…


@rofreg

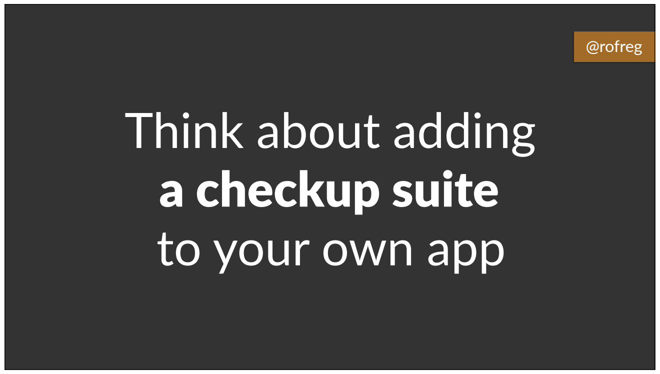
We don't have any **best practices** around these issues

…we don't have best practices yet, either. We're not thinking about this problem in a communal way. We're not learning from each other yet.


@rofreg

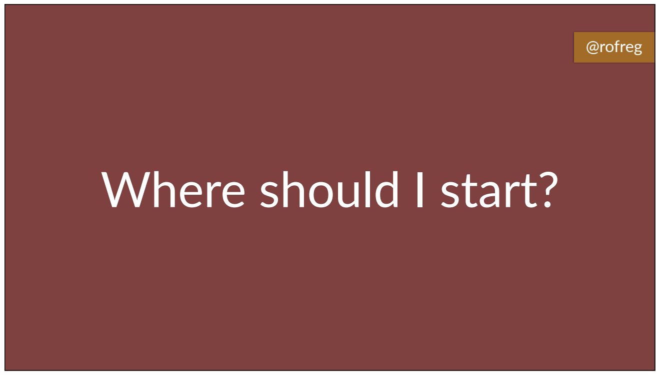
Checkups are one good way to frame the problem

My hope is that the idea of a "checkup" can be somewhere for you to start. I think it's a good, intuitive framing for how to sniff out unexpected bugs in production, and if you think about your own apps through this lens, I think you'll start to see how checkups can help you build something that's more robust and more healthy.


@rofreg

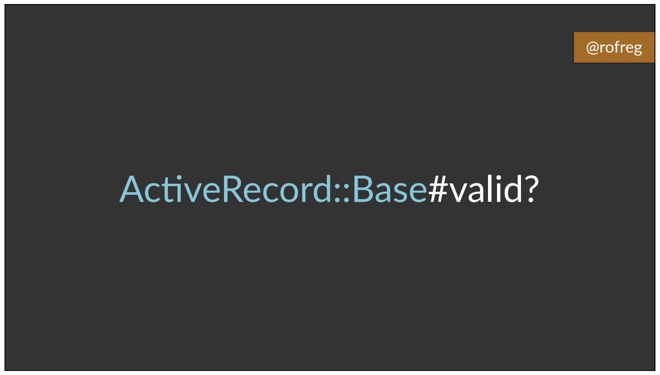
Think about adding **a checkup suite** to your own app

I honestly believe that every app should have a checkup suite. Just like you should have a test suite! Like, you definitely CAN deploy a successful app without tests or without checkups, but if you do, you're leaving yourself blind to a lot of potential problems and headaches.


@rofreg

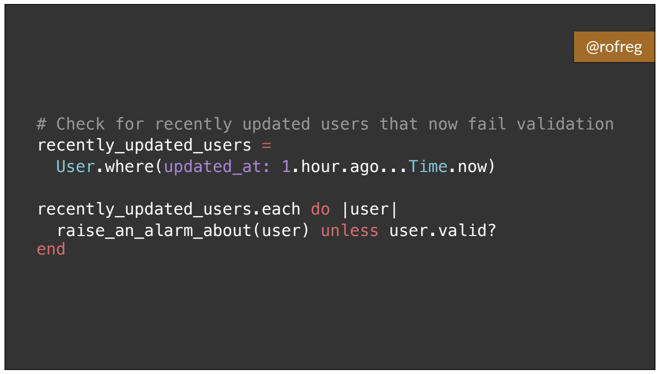
Where should I start?

Now, I realize that building a whole checkup suite may sound pretty intimidating, so here's a suggestion of one very simple place to start.


@rofreg

ActiveRecord::Base#valid?

You might be familiar with ActiveRecord's "valid" method. You can call `.valid?` on an ActiveRecord object, and it will tell you whether that object passes validation or not.

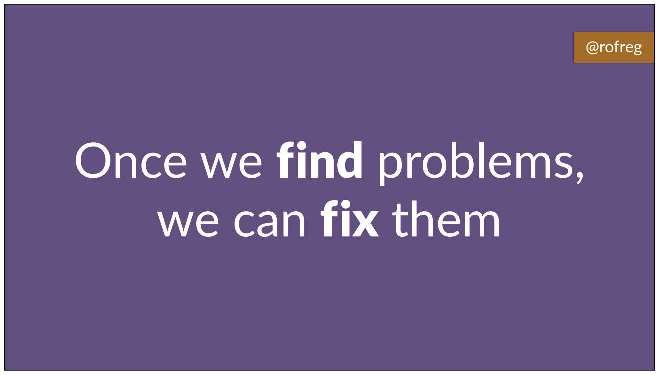

@rofreg

```ruby
# Check for recently updated users that now fail validation
recently_updated_users =
  User.where(updated_at: 1.hour.ago...Time.now)

recently_updated_users.each do |user|
  raise_an_alarm_about(user) unless user.valid?
end
```

Well, start taking advantage of that! In just a few minutes, you can write a checkup that looks at recently updated records in your database, then calls `.valid?` on each record, to make sure that the persisted data still passes validation.

Again, this is about 5 lines of code. It's a pretty easy place to start. And if you run this on all of your ActiveRecord models, I'm confident that you will find some invalid records that managed to weasel their way into your database. You'll be surprised at what you find.


@rofreg

Once we **find** problems, we can **fix** them

And once you find those problems, you can start fixing them.

Ryan Laughlin

@rofreg

http://rofreg.com/talks

Again, my name is Ryan Laughlin.
I'm @rofreg on Twitter, and you can find all these slides at rofreg.com/talks.
I really care about this idea, so I'd love to answer any questions y'all have!