Welcome welcome! My name is Mercedes Bernard. And today we're going to be talking about how to get comfy in a new codebase. So kick back, relax, and make yourself at home.

mercedesbernard.com/speaking/come-on-in

If you are someone who likes to follow along with slides, I've published them on my website.

4 years ago, Sarah Mei gave a great keynote at RailsConf about Livable Code. I encourage you to check it out if you haven't seen it. She talked about how writing applications is no longer like construction or architecture. We're not focused on making buildings.

Now we're focused on making those buildings livable for ourselves and our team. We organize the space and the code so we can do the things we want to do comfortably.

I really liked this metaphor for thinking about a codebase like a house that you decorate and make your own.

I've been a consultant for the last 10 years. This means that I don't really get to claim any codebase as home. I work in up to 20 repos a year. Just last year, I pushed production code in 18 repos that I can remember.

So because I don’t get get to claim any codebase as my home base. It means that I have to find ways to make myself feel at home in a place that isn’t really mine.

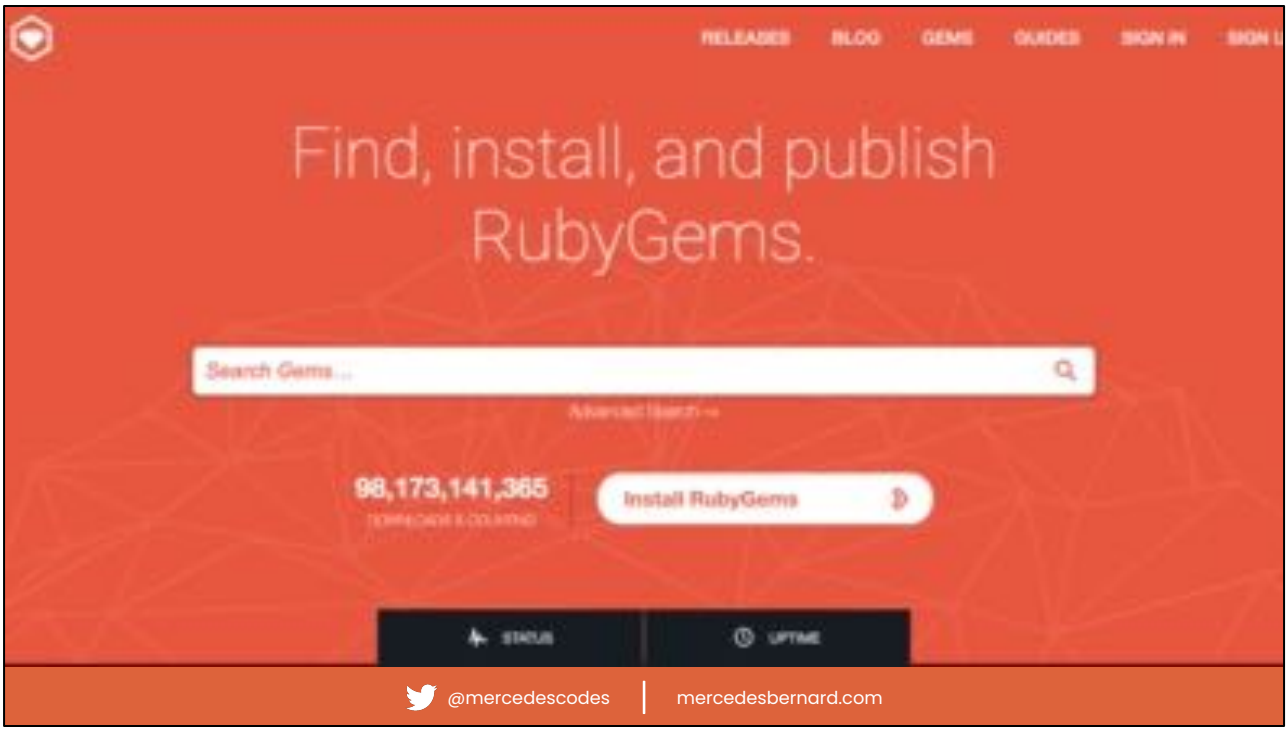
It’s like I’m a reverse digital nomad. I always work out of the same office but in completely different digital spaces. It’s the AirBnb of coding.

And Idk about you but as much as I would love to be a world traveler, living out of a suitcase stresses me out just a bit.

Even though jumping into a new codebase is something I do pretty regularly, it always gives me a bit of anxiety. What if this is the first one that stumps me? What if I can't find my way around? What if the house doesn't have a blow dryer or towels?

This anxiety is super normal. And just like any seasoned traveler will tell you, it gets easier the more you do it. You figure out how to quickly assess a space to make sure it has what you need. You learn what bits are most important and which ones actually aren't or at least not right away.

On your first morning of your visit, you'll need to find the towels and figure out how to work the shower but you won't need to worry about learning how the fancy sound system works or going through the closets to find all the cleaning supplies just yet.



If I'm going to be showing you how to get comfy in a brand new space, it should be a brand new space for me too. So for this talk, I decided to make a contribution to an open source repo that I've never really looked at but we all use all the time, rubygems.org. Shoutout to Ruby Central who maintains RubyGems and who also is hosting this wonderful conference for us :)

Welcome

Gain an understanding

@mercedescodes | mercedesbernard.com

Before you can contribute to a new codebase, you have to get a high-level understanding of what's there and where things are. When you're staying in an AirBnb, the first thing you do is walk through the house right? Or if you're visiting a friend for the first time, they might give you a tour. Or at the very least show you the room you're staying in and the bathroom.

Documentation

@mercedescodes | mercedesbernard.com

Documentation is the place to start your tour. I know not every codebase has the best documentation but there is almost always something to get you started.

Look at the README and follow links you find there. Look for contributing guides, listed dependencies, links to staging and production sites, design diagrams, etc.

RubyGems.org (née Gemcutter)

The Ruby community's gem host.

Purpose

- Provide a better API for dealing with gems
- Create more transparent and accessible project pages
- Enable the community to improve and enhance the site

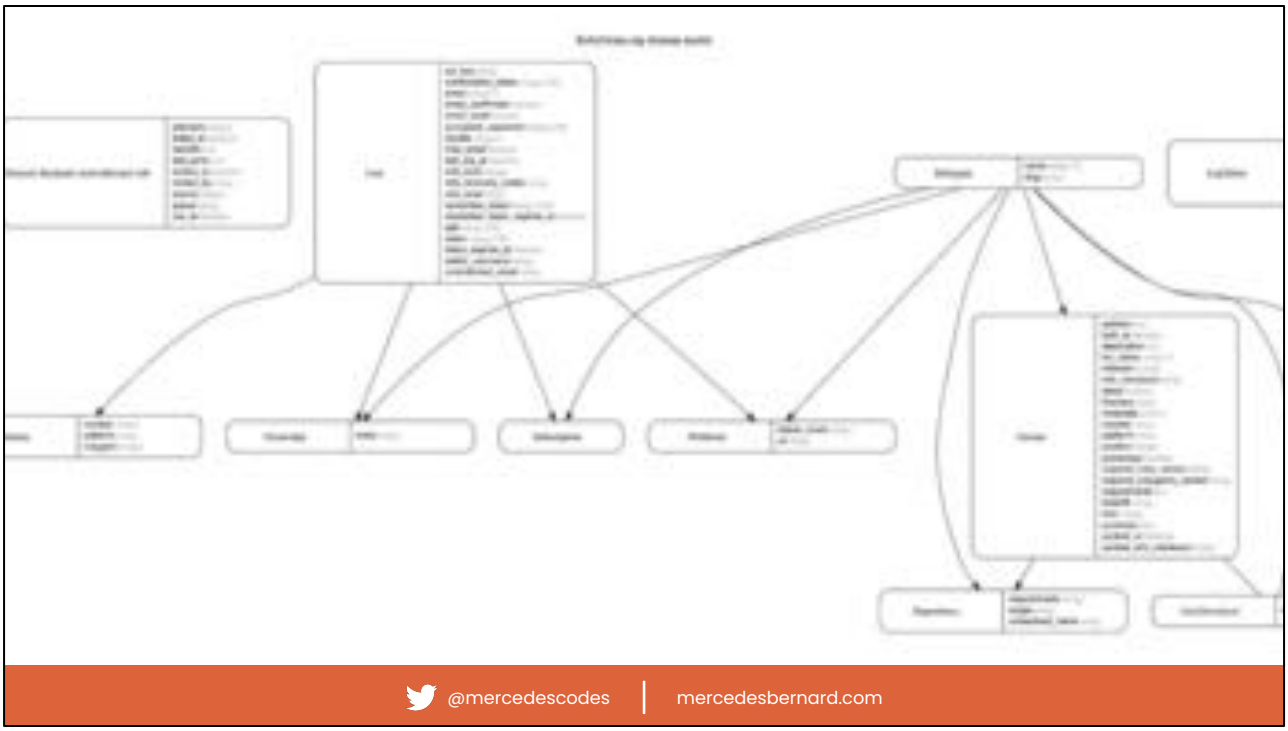
Support

Ruby Central

RubyGems.org is managed by Ruby Central, a community-funded organization supported by conference participation for RailsConf and RubyConf through tickets and sponsorships.

Hosting fees are paid by Ruby Central and CDN fees are generously donated by Fastly.

RubyGems.org README is pretty slim but I learned a lot of information from it. The main helpful parts of the README were the links they included under the Contributions header.

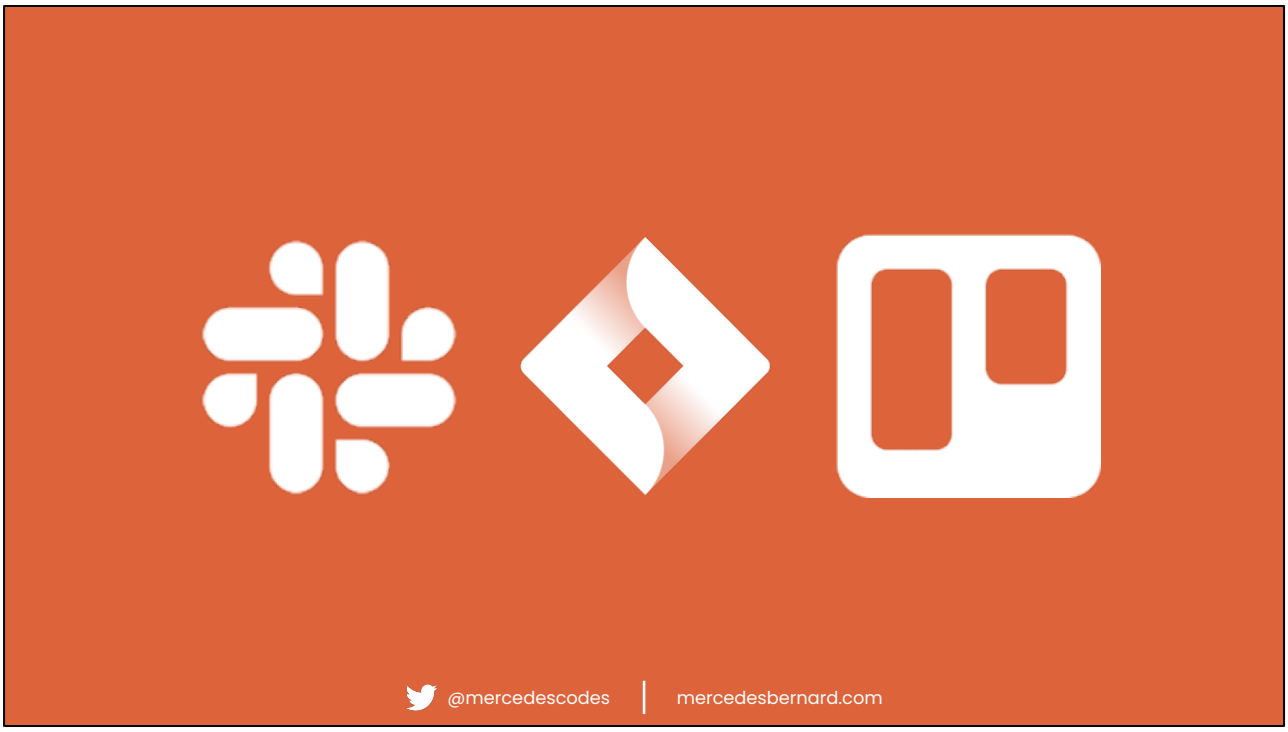


In the linked contribution guidelines, there's this really helpful entity relationship diagram. When I find diagrams in documentation, I get so excited. It's like my host left out a little tray of cookies.

Diagrams are a simple way to show a lot of info all at once. ERDs show you model relationships, sequence diagrams can you show you data flow, and a simple flow chart can show expected app behavior.

In this diagram, I can see that the database is pretty simple. There aren't too many models for me to keep straight and I start to get a sense of what I'll find in the code.

For example here, I can see that Users and Webhooks are related so someone can probably set up webhooks for themselves to be notified when some events occur.



Other places I like to look for useful information when I'm jumping into new code is Slack threads and project management tickets. Both of those could turn into giant black holes of information so it's important to timebox yourself. I typically only look back a sprint or two to see what the team has been focusing their time and energy on lately and where that fits in with the repo. Any longer than that and you're wasting valuable brain space on info that could already be outdated.

Set it up

Rubygems.org's contribution guidelines provided everything I needed to know about how to get the project set up. After I read the README and linked documents, that's where I usually go next. I learn a lot about a project by trying to get all the pieces working.

I think of this a little bit like walking through the house and finding all the things you'll need every day and making sure you know where they are. Silverware? Check. Remotes? Check. Correct Ruby version? Check. Database? Check.

As you're setting the repo, pay attention to different services you'll need.

- What database?
- What cache?
- What web server?
- Are there any other services that this repo is dependent on?
- How is the app deployed?

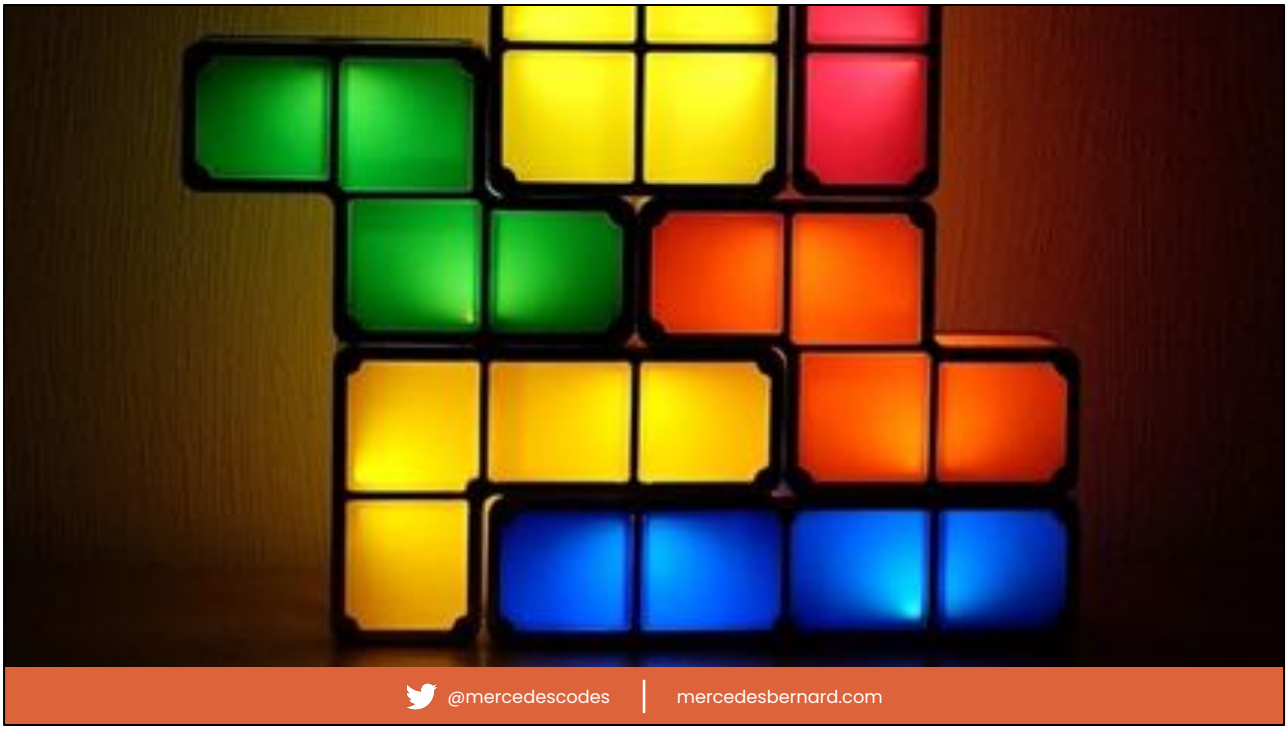
All of this information is helpful for you as you form a mental model of this codebase and how it fits into the wider system.

```yaml
services:
  db:
    image: postgres:11.13
    ports:
      - "5432:5432"
    environment:
      - POSTGRES_HOST_AUTH_METHOD=trust
  cache:
    image: memcached:1.4.24
    ports:
      - "11211:11211"
  search:
    image: elasticsearch:7.10.1
    environment:
      - http.host=0.0.0.0
      - transport.host=127.0.0.1
      - xpack.security.enabled=false
    ports:
      - "9200:9200"
  toxiproxy:
    image: shopify/toxiproxy:2.1.4
    network_mode: "host"
```

@mercedescodes | mercedesbernard.com

If the project is containerized, looking at the container config is a good place to start to find this info. If it's not, the act of setting the project up will probably require you to set up each of these dependencies separately so you'll be aware of them by the time you get the project running.

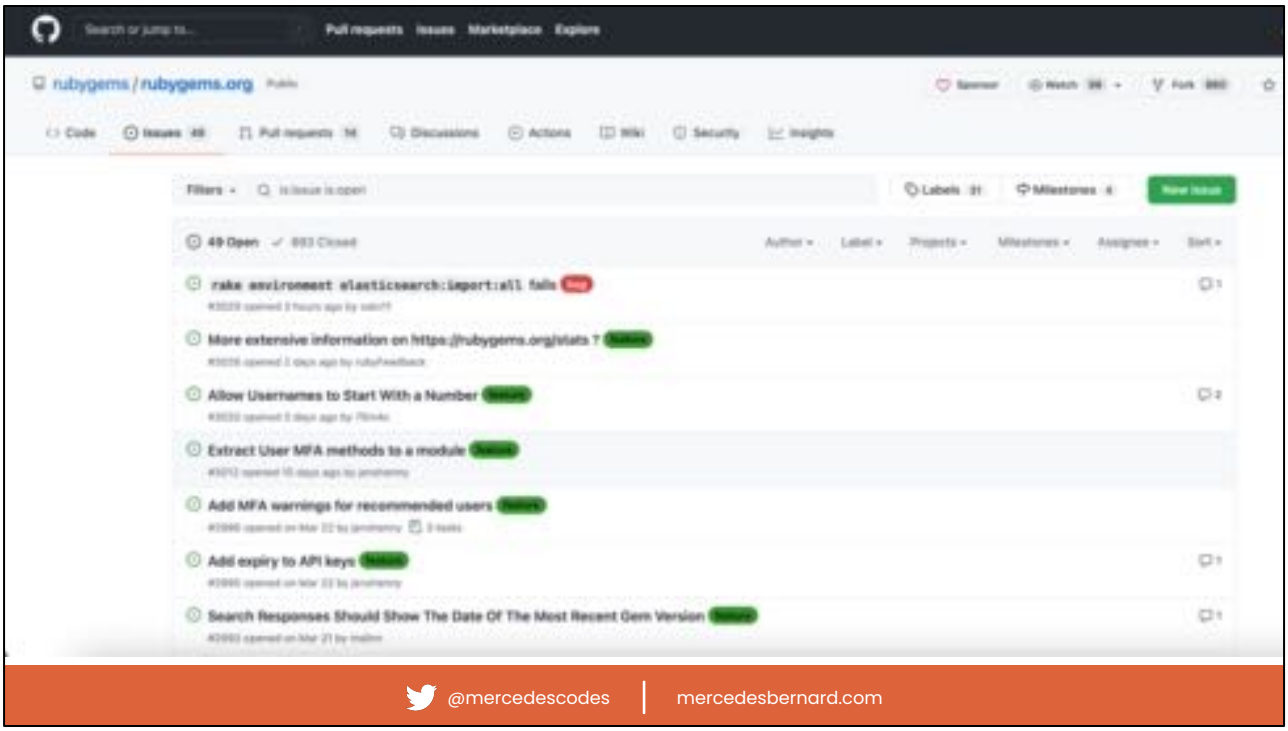
During project set up, you'll also find out how dev-friendly the codebase is. How much help do you need from other members of your team just to get the app running? Can you do it by yourself in an hour or less? If it takes you 3 days and requires 3 team members trying to help you get unstuck, you might prepare for there to a mess waiting for you on the other side of the door.



After getting the project set up, I find it hard to try to absorb much more about the code in the abstract.

If someone is trying to describe a floor plan to you, it can be pretty hard to understand. I had this tiny studio apartment that was shaped like a tetris piece where there was a hallway when you walked in, with a bathroom off to the side and a big closet opposite it. At the end of the hallway was the main living space but then it wrapped around with the kitchen in the other end. And no matter how simple I try to explain it, I feel like I always leave someone confused.

Trying to learn about a codebase by only reading it and its docs is kinda like that.



@mercedescodes | mercedesbernard.com

Once I have the project running, I pick up a ticket or an issue and start working on understanding it and the changes it requires.

This is the main takeaway of this talk. You don’t need to learn the whole codebase to make a valuable contribution. By tackling bite-sized issues ticket by ticket, you’ll learn more and faster than if you feel pressured to understand the whole system before you pick up a ticket.

Learning this way takes advantage of both scaffolding and spaced learning. Scaffolding is breaking up learning into chunks and providing a tool, or structure, with each chunk, in this case a specific bug to resolve or small feature to implement.

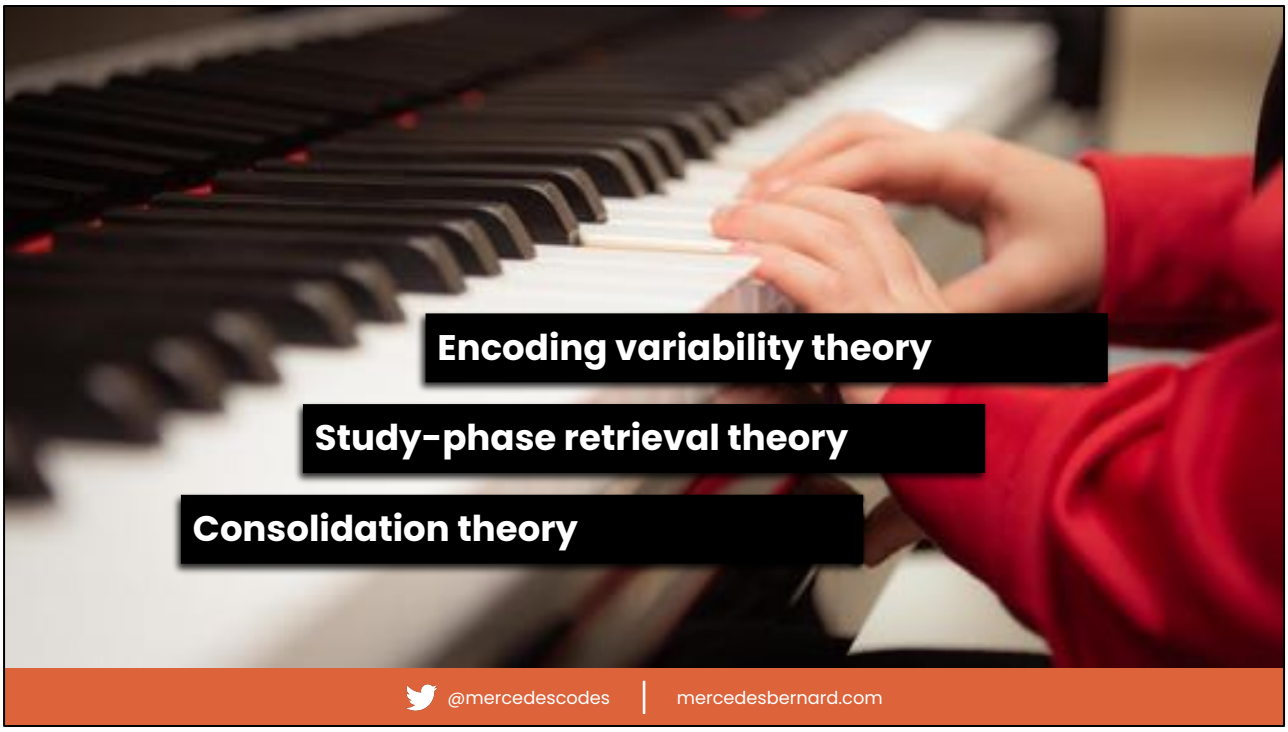
And spaced learning refers to practicing at regular intervals which is more effective than practicing all at once.

And there are multiple psychological theories that back up the efficacy of learning this way.

Encoding variability theory: learning over time is more likely to include a larger variety of contexts (such as weather, wave height, locations and observations)

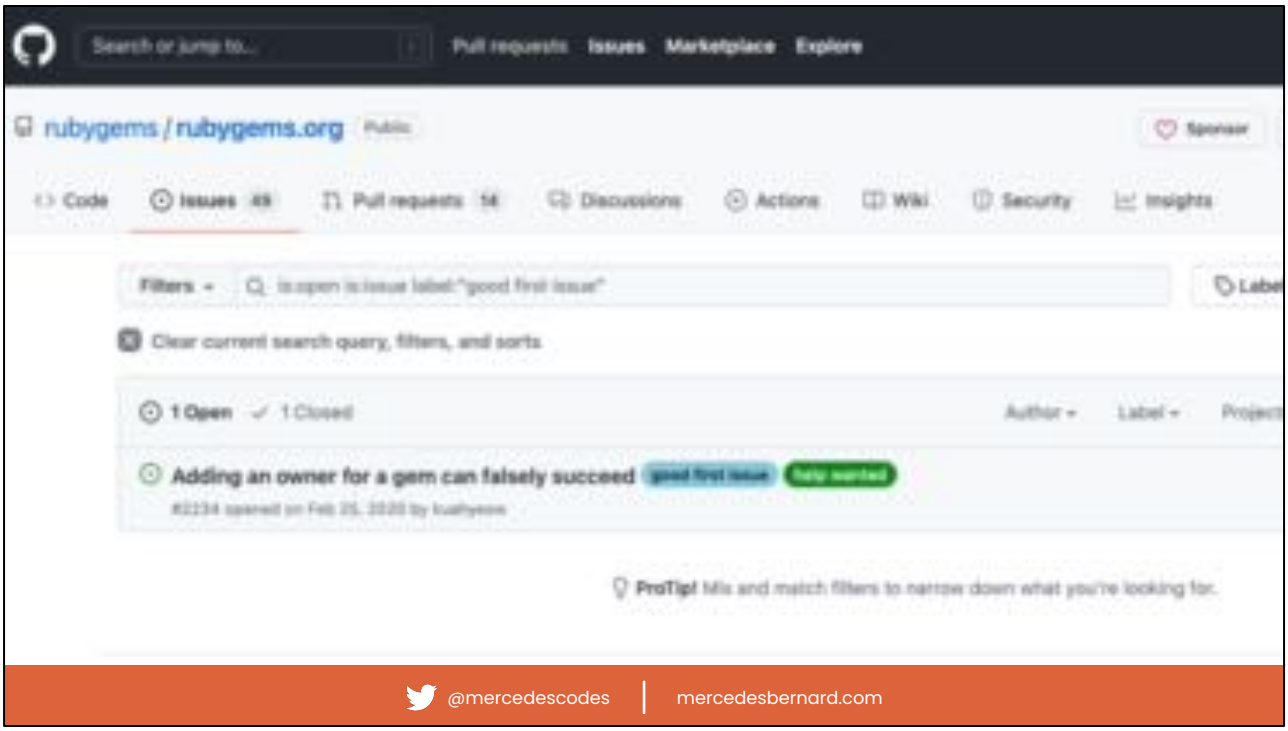
And there are multiple psychological theories that back up the efficacy of learning this way.

Study-phase retrieval theory: memory pathways are improved with each new learning opportunity due to the repeated retrieval of memories from long-term memory



And there are multiple psychological theories that back up the efficacy of learning this way.

Consolidation theory: the more practice (known by the technical term ‘rehearsal’) that occurs, the more likely it is that the learning will be committed to long-term memory



So in true scaffolding fashion, I looked for a “good first issue”. In this case, there was only one. Easy choice!

https://github.com/rubygems/rubygems.org/issues/2234

Adding an owner for a gem can falsely succeed #2234

Open

kuahyeow opened this issue on Feb 25, 2020 · 0 comments · May be fixed by #3030

kuahyeow commented on Feb 25, 2020

It seems possible to have a situation where the owner seems to have been added but in the backend it was not.

This can be seen in https://gitlab.com/gitlab-org/gitlab/issues/207207#note_293730337 where @cablett first tried to add an owner and received the `Owner added successfully.` response. But the new owner was not really added.

On the next day the same command worked again: https://gitlab.com/gitlab-org/gitlab/issues/207207#note_294302339

My theory is that in

rubygems.org/app/controllers/api/v1/owners_controller.rb
Lines 14 to 18 in 1d3d8ba

```
14    def create
15      owner = User.find_by_name(params[:email])
16      if owner
17        @rubygem.ownerships.create(user: owner)
18        render plain: "Owner added successfully."
```

, we are not checking the return value of `@rubygem.ownerships.create(user: owner)`. Does this make sense ? Is it possible to check what happened above ?

If that is the cause, I'm happy to provide a PR

jaqpa mentioned this issue on Mar 3, 2020

Fix cases where #create could silently fail #2249

Closed

Assignees

No one assigned

Labels

good first issue help wanted

Projects

None yet

Milestone

No milestone

Development

Successfully merging a pull request may close this issue.

Prevent remaining possible silent failures for cre...
mercedes/rubygems.org

Notifications Customize

Subscribe

You're not receiving notifications from this thread.

@mercedescodes | mercedesbernard.com

This issue reported was a bug where sometimes when adding an owner to a gem, it would silently fail.

Version control

@mercedescodes | mercedesbernard.com

To start working on a ticket, we get to use one of my favorite strategies for learning more about the relevant code which is digging into version control. It's a little bit of code archaeology.

Version control history, commit messages, and past pull requests tell you a lot about

- what's been tried before
- what has worked
- what hasn't worked
- why changes were made
- and who made them

This is all valuable information when you're trying to work with code you've never seen before. It can save you time from trying the wrong solution. It can help you understand why a part of the code seems to make no sense. Iit can point you toward who to ask questions and if they're still available to you.

Fix cases where #create could silently fail #2249

Closed | jrajat wants to merge 4 commits into rubygems:master from jrajat:fix-silently-failing-create

Conversation 7 | Commits 4 | Checks 0 | Files changed 6

jrajat commented on Mar 3, 2020 · Contributor

Based on #2234 I noticed that there were 3 uses of ActiveRecord's #create that did not handle the failing case. I've changed the calls to #create! which will throw an exception if it fails.

Fix cases where #create could silently fail — de0132

sonalkr132 reviewed on Mar 9, 2020 · View changes

app/models/rubygem.rb · Outdated · Show resolved

app/controllers/api/v1/owners_controller.rb

```
@@ -14,7 +14,7 @@ def show
 def create
   owner = User.find_by_name(params[:email])
   if owner
-    @rubygem.ownerships.create(user: owner)
+    @rubygem.ownerships.create!(user: owner)
```

sonalkr132 on Mar 9, 2020 · Member

Reviewers: maxaria, sonalkr132

Assignees: No one assigned

Labels: needs-more-work

Projects: None yet

Milestone: No milestone

Development: Successfully merging this pull request may close these issues.

@mercedescodes | mercedesbernard.com

In this case, someone had opened a PR for this issue 2 years ago but had closed it without merging. Their solution had been to change the `creates` to `create!` so that if saving failed, it would raise an error.

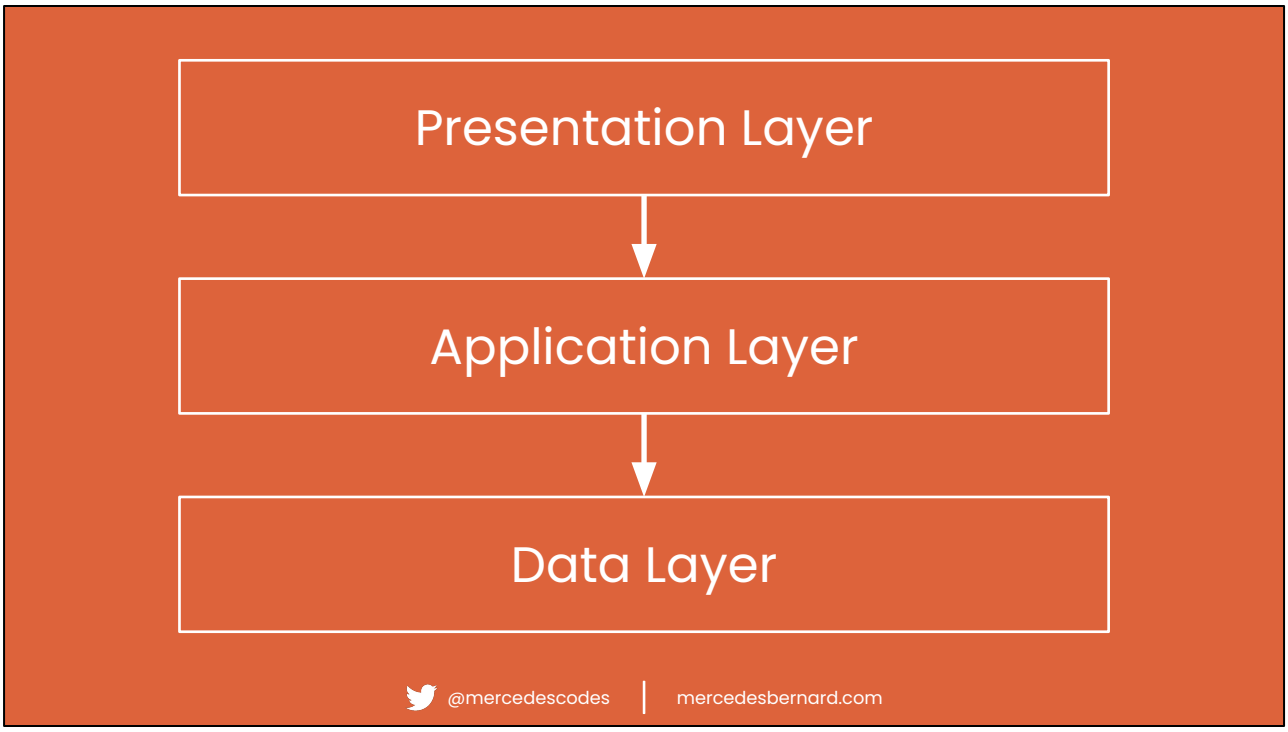
In the conversation, one of the members suggested a slight change to check the return value of the save and display errors. But overall, the solution and more helpfully, the location of the solution, would work. The PR was closed due to lack of activity but now I know that this could be part of what I need.

Trace functionality

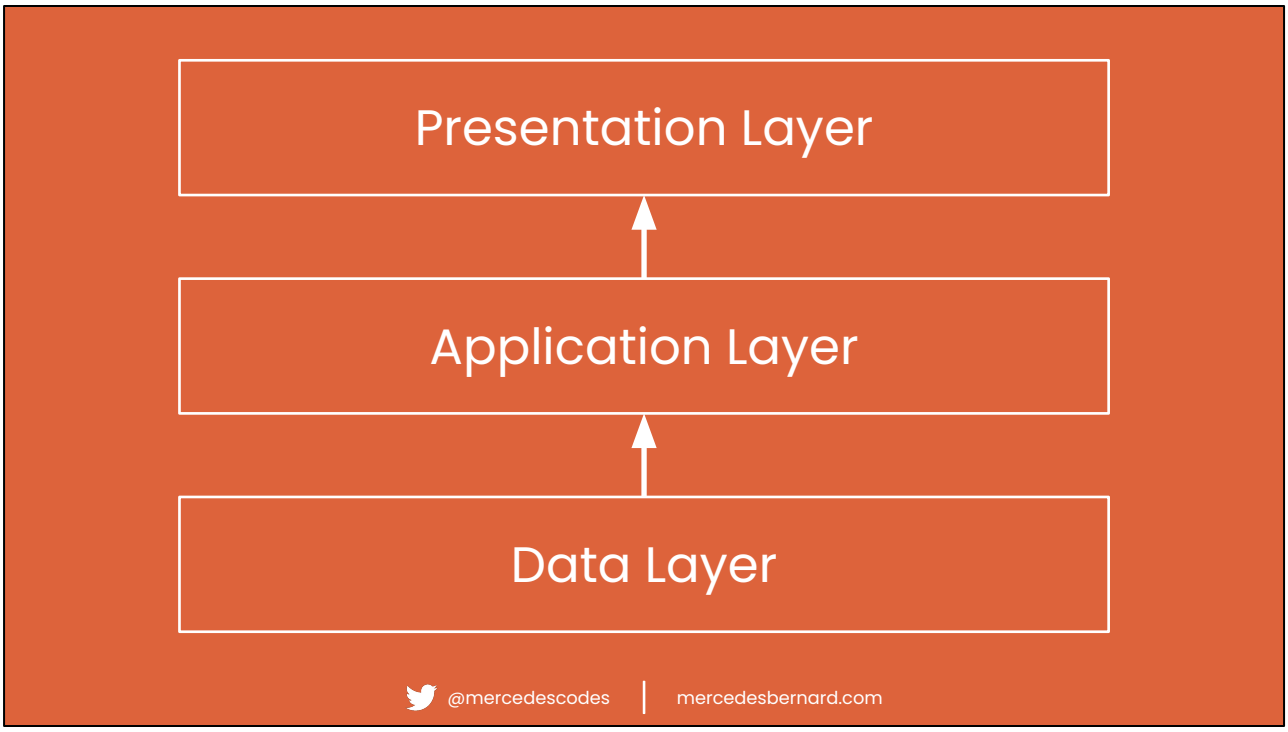
Before I start making changes, I like to make sure I understand what the code is doing. Not all the code, just the bits I'm concerned with for the ticket I'm working on.

I would love to know how the whole codebase works but there's no way I can hold all that context in my head. So I start small. I'm intentionally scaffolding and spacing my learning.

I trace the functionality I'm working with through the call stack to see what methods get invoked and what they do. This seems simple but reading someone else's code can be challenging.



To get started, you need to find your entry point into the control flow. I tend to like to work from the frontend down to the database, finding where a user interacts with a feature and then following the code that direction.



But in some cases starting at the database and learning about the models and associations and climbing up to the frontend might be better.

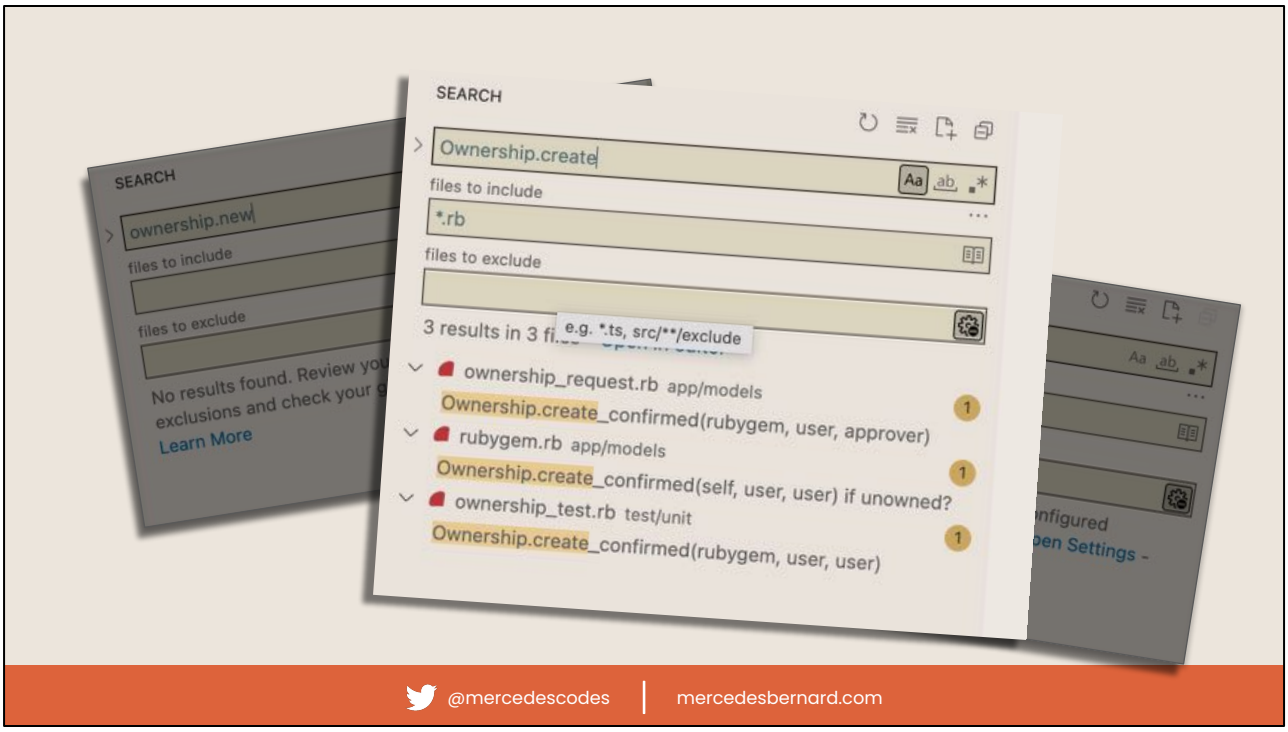
```ruby
class Ownership < ApplicationRecord
 belongs_to :rubygem
 belongs_to :user
 belongs_to :authorizer, class_name: "User"

 validates :user_id, uniqueness: { scope: :rubygem_id }
 # …
end
```


Let's start by looking at our model layer. This is the Ownership model. The important bits right now are the associations and validations. We can ignore any custom class and instance methods until we encounter them while we're tracing the code. It looks like we have a few associations and we can only have one Ownership per user/rubygem combination.

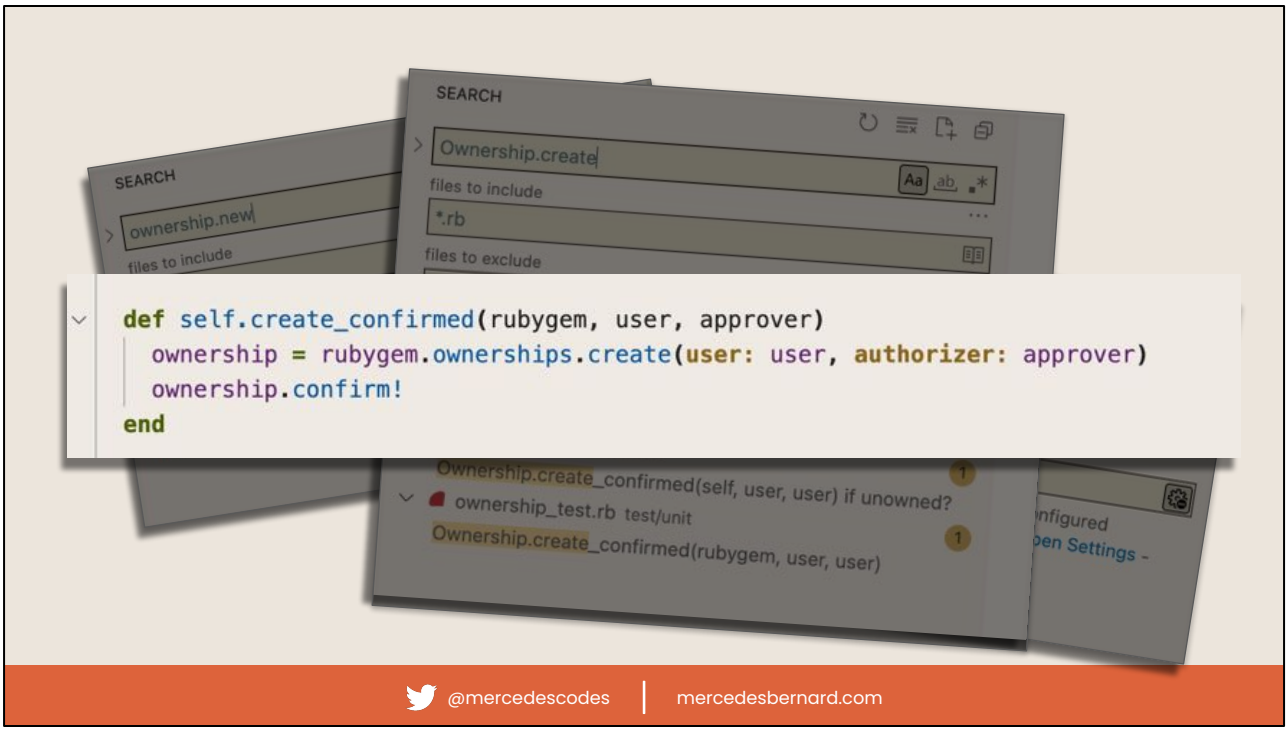
I wasn't familiar with how ownerships get added so to start tracing the code my entrypoint was finding all the instances where we were creating new ownerships since I want to make sure that none would silently fail anymore. I'm not using any fancy IDE extensions, just vanilla code search.

To find where this is happening, I leaned on my ActiveRecord knowledge. Tapping into prior knowledge is a scaffolding technique that helps with retaining information by building connections of what you're learning to what you already know.

First up, where are we initializing new Ownerships. Apparently no where.

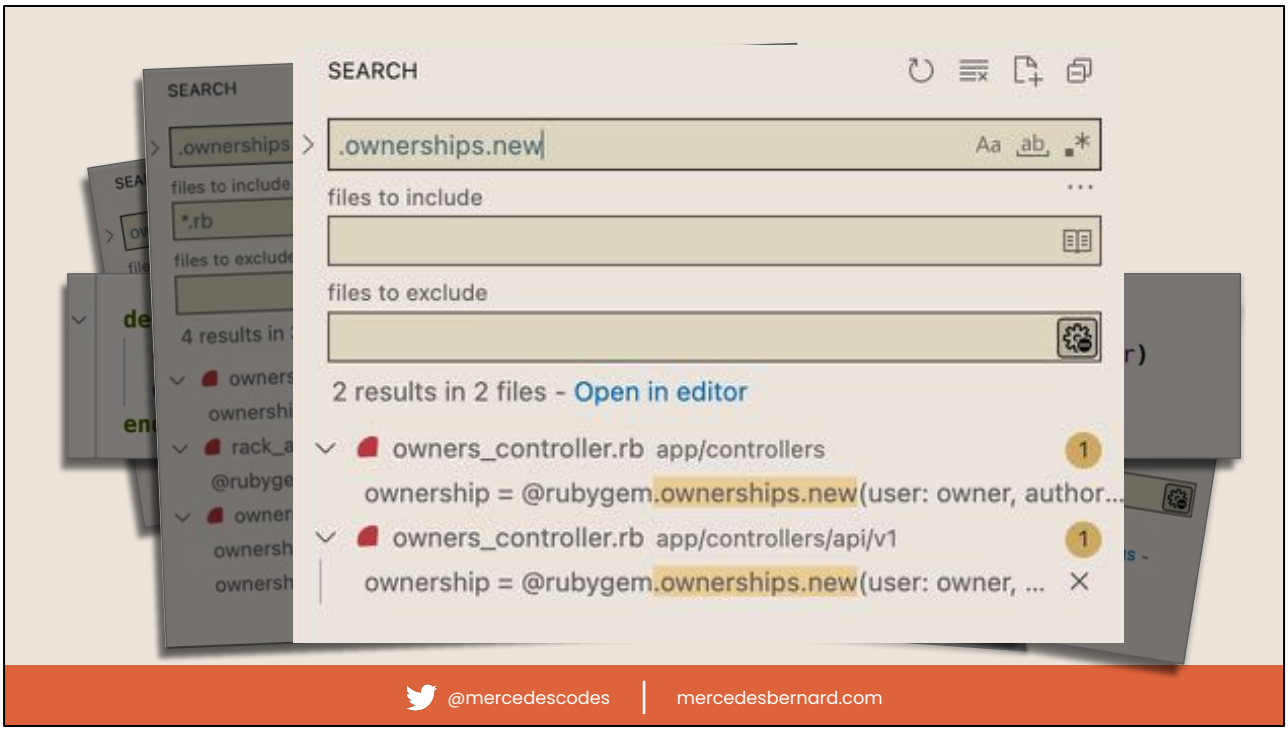


Ok, well what about where we're not initializing but we're creating? This is interesting! I can see that there is a custom class method on the Ownership model that's invoked in 2 places. So I'll note those down as places to come back to.



And in that method, it looks like it's creating ownerships from the rubygem association. That's also interesting! I wonder if anywhere else in the codebase follows that pattern.

Nope, looks like ownerships.create is only in the Ownership class method.



But ownerships.new is used in a couple different controllers. So I'll note those 2 places down as well.

```
models/ownership_request.rb: Ownership.create_confirmed

models/rubygem.rb: Ownership.create_confirmed

app/controllers/owners_controller.rb: rubygem.ownerships.new

app/controllers/api/v1/owners_controller.rb: rubygem.ownerships.new
```


I have 4 places now to look at ownerships being created and saved to the database.

Notice, that I still don't know a ton about these 4 use cases. I couldn't tell you when they get called or from where. And that's ok. I'm going to keep my focus on the small task I'm trying to complete.

```ruby
def create
  return render_api_key_forbidden unless @api_key.can_add_owner?

  owner = User.find_by_name(params[:email])
  if owner
    ownership = @rubygem.ownerships.new(user: owner, authorizer: @api_key.user)
    if ownership.save
      Delayed::Job.enqueue(OwnershipConfirmationMailer.new(ownership.id))
      render plain: "#{owner.display_handle} was added as an unconfirmed owner…"
    else
      render plain: ownership.errors.full_messages.to_sentence, status: :unprocess…
    end
  else
    render plain: "Owner could not be found.", status: :not_found
  end
end
```


So let's take a look at the controllers first. This is where we have the ownerships.new. And this is where the changes in that closed PR were that we looked at earlier.

```ruby
def create
  return render_api_key_forbidden unless @api_key.can_add_owner?

  owner = User.find_by_name(params[:email])
  if owner
    ownership = @rubygem.ownerships.new(user: owner, authorizer: @api_key.user)
    if ownership.save
      Delayed::Job.enqueue(OwnershipConfirmationMailer.new(ownership.id))
      render plain: "#{owner.display_handle} was added as an unconfirmed owner…"
    else
      render plain: ownership.errors.full_messages.to_sentence, status: :unprocess…
    end
  else
    render plain: "Owner could not be found.", status: :not_found
  end
end
```


This code must have been updated since the issue was reported because in both of these controllers, the value of `save` is checked (like the suggestions in the closed PR thread said). If the ownership is invalid and is not saved, then an error is reported to the user. So neither of the controllers seem to still be problems.

Which means we've narrowed down the possible remaining culprits to just be the places where the custom class method Ownership.create_confirmed is called.

Interactive debugging

My absolute, #1 favorite way to explore and understand new code is interactive debugging. I learn more from this than even what the version control history tells me. I'm not a debugging purist and love and use my `puts` statements when they feel right but I find that the control and the 2-way communication that interactive debugging provides helps me learn a lot and very quickly.

You can think of interactive debugging like opening drawers and closets to see what's in there and take an inventory of everything you have available.

```ruby
def self.create_confirmed(rubygem, user, approver)
  ownership = rubygem.ownerships.create(user: user, authorizer: approver)
  binding.pry
  success = ownership.confirm!
  binding.pry
end
```


In this case, I wanted to know what that class method was doing. And how it handled non-valid, non-persisted models. So I threw a couple pry statements in the method, set the return value of confirm! to a variable so I could inspect it…

```
mercedesbernard@Mercedess-MBP rubygems.org % rails console
Loading development environment (Rails 7.0.2.3)
irb(main):001:0> Ownership.count
  Ownership Count (5.6ms)  SELECT COUNT(*) FROM "ownerships"
=> 0
irb(main):002:0> rubygem = Rubygem.first
  Rubygem Load (1.7ms)  SELECT "rubygems".* FROM "rubygems" ORDER BY "rubygems"."id" ASC LIMIT $1  [["LIMIT", 1]]
=>
#<Rubygem:0x000000010e01c228
...
irb(main):003:0> user = User.first
  User Load (2.5ms)  SELECT "users".* FROM "users" ORDER BY "users"."id" ASC LIMIT $1  [["LIMIT", 1]]
=>
#<User:0x000000010cf885b0
...
irb(main):004:0> approver = User.last
  User Load (2.3ms)  SELECT "users".* FROM "users" ORDER BY "users"."id" DESC LIMIT $1  [["LIMIT", 1]]
=>
#<User:0x000000010d9bf178
...
irb(main):005:0>
```


and then cracked open the rails console

```
mercedesbernard@Mercedess-MBP rubygems.org % rails console
Loading development environment (Rails 7.0.2.3)
irb(main):001:0> Ownership.count
  Ownership Count (5.6ms)  SELECT COUNT(*) FROM "ownerships"
=> 0
irb(main):002:0> rubygem = Rubygem.first
  Rubygem Load (1.7ms)  SELECT "rubygems".* FROM "rubygems" ORDER BY "rubygems"."id" ASC LIMIT $1  [["LIMIT", 1]]
=>
#<Rubygem:0x000000010e01c228
...
irb(main):003:0> user = User.first
  User Load (2.5ms)  SELECT "users".* FROM "users" ORDER BY "users"."id" ASC LIMIT $1  [["LIMIT", 1]]
=>
#<User:0x000000010cf885b0
...
irb(main):004:0> approver = User.last
  User Load (2.3ms)  SELECT "users".* FROM "users" ORDER BY "users"."id" DESC LIMIT $1  [["LIMIT", 1]]
=>
#<User:0x000000010d9bf178
...
irb(main):005:0>
```


Here, you'll see that there are no Ownerships in the database

```
mercedesbernard@Mercedess-MBP rubygems.org % rails console
Loading development environment (Rails 7.0.2.3)
irb(main):001:0> Ownership.count
  Ownership Count (5.6ms)  SELECT COUNT(*) FROM "ownerships"
=> 0
irb(main):002:0> rubygem = Rubygem.first
  Rubygem Load (1.7ms)  SELECT "rubygems".* FROM "rubygems" ORDER BY "rubygems"."id" ASC LIMIT $1  [["LIMIT", 1]]
=>
#<Rubygem:0x000000010e01c228
...
irb(main):003:0> user = User.first
  User Load (2.5ms)  SELECT "users".* FROM "users" ORDER BY "users"."id" ASC LIMIT $1  [["LIMIT", 1]]
=>
#<User:0x000000010cf885b0
...
irb(main):004:0> approver = User.last
  User Load (2.3ms)  SELECT "users".* FROM "users" ORDER BY "users"."id" DESC LIMIT $1  [["LIMIT", 1]]
=>
#<User:0x000000010d9bf178
...
irb(main):005:0>
```


And I initialized the variables I'll need to pass as parameters to that create_confirmed method.

```
irb(main):006:0> Ownership.create_confirmed(rubygem, user, approver)
  TRANSACTION (1.5ms)  BEGIN
  Ownership Exists? (2.5ms)  SELECT 1 AS one FROM "ownerships" WHERE "ownerships"."user_id" = $1 AND "ownerships"."rubygem_id" = $2 L
"rubygem_id", 2], ["LIMIT", 1]]
  Ownership Create (5.1ms)  INSERT INTO "ownerships" ("rubygem_id", "user_id", "token", "created_at", "updated_at", "token_expires_a
($1, $2, $3, $4, $5, $6, $7) RETURNING "id"  [["rubygem_id", 2], ["user_id", 3], ["token", "4531327361cb2dc948fc424fbb3f5eeca1e83ae6"
26 19:36:55.343399"], ["updated_at", "2022-04-26 19:36:55.343399"], ["token_expires_at", "2022-04-28 19:36:55.344124"], ["authorizer_
  TRANSACTION (40.1ms)  COMMIT

From: /Users/mercedesbernard/Git/Personal/rubygems.org/app/models/ownership.rb:31 Ownership.create_confirmed:

    28: def self.create_confirmed(rubygem, user, approver)
    29:   ownership = rubygem.ownerships.create(user: user, authorizer: approver)
    30:   binding.pry
 => 31:   success = ownership.confirm!
    32:   binding.pry
    33: end

[1] pry(Ownership)> ownership
=> #<Ownership:0x0000000111cd1448
 id: 6,
 rubygem_id: 2,
 user_id: 3,
 token: "4531327361cb2dc948fc424fbb3f5eeca1e83ae6",
 created_at: Tue, 26 Apr 2022 19:36:55.343399000 UTC +00:00,
 updated_at: Tue, 26 Apr 2022 19:36:55.343399000 UTC +00:00,
 push_notifier: true,
 confirmed_at: nil,
 token_expires_at: Thu, 28 Apr 2022 19:36:55.344124000 UTC +00:00,
 owner_notifier: true,
 authorizer_id: 4,
 ownership_request_notifier: true>
[2] pry(Ownership)>
```


Once I run create_confirmed…

```
irb(main):006:0> Ownership.create_confirmed(rubygem, user, approver)
  TRANSACTION (1.9ms)  BEGIN
  Ownership Exists? (2.5ms)  SELECT 1 AS one FROM "ownerships" WHERE "ownerships"."user_id" = $1 AND "ownerships"."rubygem_id" = $2 L
"rubygem_id", 2], ["LIMIT", 1]]
  Ownership Create (5.1ms)  INSERT INTO "ownerships" ("rubygem_id", "user_id", "token", "created_at", "updated_at", "token_expires_a
($1, $2, $3, $4, $5, $6, $7) RETURNING "id"  [["rubygem_id", 2], ["user_id", 3], ["token", "4531327361cb2dc948fc424fbb3f5eeca1e83ae6"
26 19:36:55.343399"], ["updated_at", "2022-04-26 19:36:55.343399"], ["token_expires_at", "2022-04-28 19:36:55.344124"], ["authorizer_
  TRANSACTION (40.1ms)  COMMIT

From: /Users/mercedesbernard/Git/Personal/rubygems.org/app/models/ownership.rb:31 Ownership.create_confirmed:

    28: def self.create_confirmed(rubygem, user, approver)
    29:   ownership = rubygem.ownerships.create(user: user, authorizer: approver)
    30:   binding.pry
 => 31:   success = ownership.confirm!
    32:   binding.pry
    33: end

[1] pry(Ownership)> ownership
=> #<Ownership:0x0000000111cd1448
 id: 6,
 rubygem_id: 2,
 user_id: 3,
 token: "4531327361cb2dc948fc424fbb3f5eeca1e83ae6",
 created_at: Tue, 26 Apr 2022 19:36:55.343399000 UTC +00:00,
 updated_at: Tue, 26 Apr 2022 19:36:55.343399000 UTC +00:00,
 push_notifier: true,
 confirmed_at: nil,
 token_expires_at: Thu, 28 Apr 2022 19:36:55.344124000 UTC +00:00,
 owner_notifier: true,
 authorizer_id: 4,
 ownership_request_notifier: true>
[2] pry(Ownership)>
```

@mercedescodes | mercedesbernard.com

the code will pause execution at my first breakpoint.

```
irb(main):006:0> Ownership.create_confirmed(rubygem, user, approver)
  TRANSACTION (1.9ms)  BEGIN
  Ownership Exists? (2.5ms)  SELECT 1 AS one FROM "ownerships" WHERE "ownerships"."user_id" = $1 AND "ownerships"."rubygem_id" = $2 L
"rubygem_id", 2], ["LIMIT", 1]]
  Ownership Create (5.1ms)  INSERT INTO "ownerships" ("rubygem_id", "user_id", "token", "created_at", "updated_at", "token_expires_a
($1, $2, $3, $4, $5, $6, $7) RETURNING "id"  [["rubygem_id", 2], ["user_id", 3], ["token", "4531327361cb2dc948fc424fbb3f5eeca1e83ae6"
26 19:36:55.343399"], ["updated_at", "2022-04-26 19:36:55.343399"], ["token_expires_at", "2022-04-28 19:36:55.344124"], ["authorizer_
  TRANSACTION (40.1ms)  COMMIT

From: /Users/mercedesbernard/Git/Personal/rubygems.org/app/models/ownership.rb:31 Ownership.create_confirmed:

    28: def self.create_confirmed(rubygem, user, approver)
    29:   ownership = rubygem.ownerships.create(user: user, authorizer: approver)
    30:   binding.pry
 => 31:   success = ownership.confirm!
    32:   binding.pry
    33: end

[1] pry(Ownership)> ownership
=> #<Ownership:0x0000000111cd1448
 id: 6,
 rubygem_id: 2,
 user_id: 3,
 token: "4531327361cb2dc948fc424fbb3f5eeca1e83ae6",
 created_at: Tue, 26 Apr 2022 19:36:55.343399000 UTC +00:00,
 updated_at: Tue, 26 Apr 2022 19:36:55.343399000 UTC +00:00,
 push_notifier: true,
 confirmed_at: nil,
 token_expires_at: Thu, 28 Apr 2022 19:36:55.344124000 UTC +00:00,
 owner_notifier: true,
 authorizer_id: 4,
 ownership_request_notifier: true>
[2] pry(Ownership)>
```


Here, you'll see that we've created an ownership. And when I look at the value of it, it looks good to me. It has an id so we know it's persisted and I can see all its properties.

```
From: /Users/mercedesbernard/Git/Personal/rubygems.org/app/models/ownership.rb:31 Ownership.create_confirmed:

    28: def self.create_confirmed(rubygem, user, approver)
    29:   ownership = rubygem.ownerships.create(user: user, authorizer: approver)
    30:   binding.pry
 => 31:   success = ownership.confirm!
    32:   binding.pry
    33: end

[3] pry(Ownership)> step

From: /Users/mercedesbernard/Git/Personal/rubygems.org/app/models/ownership.rb:62 Ownership#confirm!:

    61: def confirm!
 => 62:   update(confirmed_at: Time.current, token: nil) if unconfirmed?
    63: end

[3] pry(#<Ownership>)> unconfirmed?
=> true
[4] pry(#<Ownership>)>
```

@mercedescodes | mercedesbernard.com

So now I want to step into that `confirm!` method and see what's happening.

```
From: /Users/mercedesbernard/Git/Personal/rubygems.org/app/models/ownership.rb:31 Ownership.create_confirmed:

    28: def self.create_confirmed(rubygem, user, approver)
    29:   ownership = rubygem.ownerships.create(user: user, authorizer: approver)
    30:   binding.pry
 => 31:   success = ownership.confirm!
    32:   binding.pry
    33: end

[3] pry(Ownership)> step

From: /Users/mercedesbernard/Git/Personal/rubygems.org/app/models/ownership.rb:62 Ownership#confirm!:

    61 def confirm!
 => 62   update(confirmed_at: Time.current, token: nil) if unconfirmed?
    63 end

[3] pry(#<Ownership>)> unconfirmed?
=> true
[4] pry(#<Ownership>)>
```


We can see it just calls the vanilla update method if the model hasn't been confirmed yet and returns the value of the update. I'm glad I stepped into this method because I would assume that it ran update! to raise an error if the model was invalid but it doesn't, it would just return false.

```
[4] pry(#<Ownership>)> next
  TRANSACTION (3.7ms)  BEGIN
  Ownership Exists? (3.8ms)  SELECT 1 AS one FROM "ownerships" WHERE "ownerships"."user_id" = $1 AND "owners
LIMIT $4  [["user_id", 3], ["id", 6], ["rubygem_id", 2], ["LIMIT", 1]]
  Ownership Update (6.9ms)  UPDATE "ownerships" SET "token" = $1, "updated_at" = $2, "confirmed_at" = $3 WHE
d_at", "2022-04-26 19:41:48.206995"], ["confirmed_at", "2022-04-26 19:41:48.178331"], ["id", 6]]
  TRANSACTION (7.6ms)  COMMIT

From: /Users/mercedesbernard/Git/Personal/rubygems.org/app/models/ownership.rb:32 Ownership.create_confirmed

    28: def self.create_confirmed(rubygem, user, approver)
    29:   ownership = rubygem.ownerships.create(user: user, authorizer: approver)
    30:   binding.pry
    31:   success = ownership.confirm!
 => 32:   binding.pry
    33: end

[4] pry(Ownership)> success
=> true
[5] pry(Ownership)>
```


So when I go to the next line, you see I come back up into the create_confirmed method

```
[4] pry(#<Ownership>)> next
  TRANSACTION (3.7ms)  BEGIN
  Ownership Exists? (3.8ms)  SELECT 1 AS one FROM "ownerships" WHERE "ownerships"."user_id" = $1 AND "owners
LIMIT $4  [["user_id", 3], ["id", 6], ["rubygem_id", 2], ["LIMIT", 1]]
  Ownership Update (6.9ms)  UPDATE "ownerships" SET "token" = $1, "updated_at" = $2, "confirmed_at" = $3 WHE
d_at", "2022-04-26 19:41:48.206995"], ["confirmed_at", "2022-04-26 19:41:48.178331"], ["id", 6]]
  TRANSACTION (7.6ms)  COMMIT

From: /Users/mercedesbernard/Git/Personal/rubygems.org/app/models/ownership.rb:32 Ownership.create_confirmed

    28: def self.create_confirmed(rubygem, user, approver)
    29:   ownership = rubygem.ownerships.create(user: user, authorizer: approver)
    30:   binding.pry
    31:   success = ownership.confirm!
 => 32:   binding.pry
    33: end

[4] pry(Ownership)> success
=> true
[5] pry(Ownership)>
```


and I can check the value of success which is true. And that's what would be returned in a successful case.

```
[5] pry(Ownership)> continue

From: /Users/mercedesbernard/Git/Personal/rubygems.org/app/models/ownership.rb:33 Ownership.create_confirmed:

    28: def self.create_confirmed(rubygem, user, approver)
    29:   ownership = rubygem.ownerships.create(user: user, authorizer: approver)
    30:   binding.pry
    31:   success = ownership.confirm!
    32:   binding.pry
 => 33: end

[1] pry(Ownership)> continue
=> nil
irb(main):007:0> Ownership.count
  Ownership Count (2.8ms)  SELECT COUNT(*) FROM "ownerships"
=> 1
irb(main):008:0>
```

@mercedescodes | mercedesbernard.com

And you can see that we now have one Ownership in the database.

```
irb(main):004:0> Ownership.create_confirmed(rubygem, user, approver)
  TRANSACTION (1.8ms)  BEGIN
  Ownership Exists? (2.3ms)  SELECT 1 AS one FROM "ownerships" WHERE "ownerships"."user_id" = $1 AND "ownerships"."rubyg
"rubygem_id", 2], ["LIMIT", 1]]
  TRANSACTION (3.0ms)  ROLLBACK

From: /Users/mercedesbernard/Git/Personal/rubygems.org/app/models/ownership.rb:31 Ownership.create_confirmed:

    28: def self.create_confirmed(rubygem, user, approver)
    29:   ownership = rubygem.ownerships.create(user: user, authorizer: approver)
    30:   binding.pry
 => 31:   success = ownership.confirm!
    32:   binding.pry
    33: end

[1] pry(Ownership)> ownership
=> #<Ownership:0x00000001126355c0
 id: nil,
 rubygem_id: 2,
 user_id: 3,
 token: nil,
 created_at: nil,
 updated_at: nil,
 push_notifier: true,
 confirmed_at: nil,
 token_expires_at: nil,
 owner_notifier: true,
 authorizer_id: 4,
 ownership_request_notifier: true>
[2] pry(Ownership)>
```

@mercedescodes | mercedesbernard.com

Now I want to look at the behavior of an unsuccessful case. So I'm going to do the exact same thing and try to create a duplicate confirmed ownership…

```
irb(main):004:0> Ownership.create_confirmed(rubygem, user, approver)
  TRANSACTION (1.0ms)  BEGIN
  Ownership Exists? (2.3ms)  SELECT 1 AS one FROM "ownerships" WHERE "ownerships"."user_id" = $1 AND "ownerships"."rubyg
"rubygem_id", 2], ["LIMIT", 1]]
  TRANSACTION (3.0ms)  ROLLBACK

From: /Users/mercedesbernard/Git/Personal/rubygems.org/app/models/ownership.rb:31 Ownership.create_confirmed:

    28: def self.create_confirmed(rubygem, user, approver)
    29:   ownership = rubygem.ownerships.create(user: user, authorizer: approver)
    30:   binding.pry
 => 31:   success = ownership.confirm!
    32:   binding.pry
    33: end

[1] pry(Ownership)> ownership
=> #<Ownership:0x00000001126355c0
 id: nil,
 rubygem_id: 2,
 user_id: 3,
 token: nil,
 created_at: nil,
 updated_at: nil,
 push_notifier: true,
 confirmed_at: nil,
 token_expires_at: nil,
 owner_notifier: true,
 authorizer_id: 4,
 ownership_request_notifier: true>
[2] pry(Ownership)>
```

@mercedescodes | mercedesbernard.com

for the same user and rubygem.

```
irb(main):004:0> Ownership.create_confirmed(rubygem, user, approver)
  TRANSACTION (1.8ms)  BEGIN
  Ownership Exists? (2.3ms)  SELECT 1 AS one FROM "ownerships" WHERE "ownerships"."user_id" = $1 AND "ownerships"."rubyg
"rubygem_id", 2], ["LIMIT", 1]]
  TRANSACTION (3.0ms)  ROLLBACK

From: /Users/mercedesbernard/Git/Personal/rubygems.org/app/models/ownership.rb:31 Ownership.create_confirmed:

    28: def self.create_confirmed(rubygem, user, approver)
    29:   ownership = rubygem.ownerships.create(user: user, authorizer: approver)
    30:   binding.pry
 => 31:   success = ownership.confirm!
    32:   binding.pry
    33: end

[1] pry(Ownership)> ownership
=> #<Ownership:0x00000001126355c0
 id: nil,
 rubygem_id: 2,
 user_id: 3,
 token: nil,
 created_at: nil,
 updated_at: nil,
 push_notifier: true,
 confirmed_at: nil,
 token_expires_at: nil,
 owner_notifier: true,
 authorizer_id: 4,
 ownership_request_notifier: true>
[2] pry(Ownership)>
```

@mercedescodes | mercedesbernard.com

When we look at the value of the created ownership, you'll see that it doesn't have an id so it wasn't persisted to the database.

```
[4] pry(#<Ownership>)> next
  TRANSACTION (5.4ms)  BEGIN
  Ownership Exists? (5.9ms)  SELECT 1 AS one FROM "ownerships" WHERE "ownerships"."user_id" = $1 AND "ownerships"."rubyg
"rubygem_id", 2], ["LIMIT", 1]]
  TRANSACTION (5.4ms)  ROLLBACK

From: /Users/mercedesbernard/Git/Personal/rubygems.org/app/models/ownership.rb:32 Ownership.create_confirmed:

    28: def self.create_confirmed(rubygem, user, approver)
    29:   ownership = rubygem.ownerships.create(user: user, authorizer: approver)
    30:   binding.pry
    31:   success = ownership.confirm!
 => 32:   binding.pry
    33: end

[4] pry(Ownership)> success
=> false
[5] pry(Ownership)> ownership.valid?
  Ownership Exists? (2.6ms)  SELECT 1 AS one FROM "ownerships" WHERE "ownerships"."user_id" = $1 AND "ownerships"."rubyg
"rubygem_id", 2], ["LIMIT", 1]]
=> false
[6] pry(Ownership)> ownership.errors.full_messages
=> ["User has already been taken"]
[7] pry(Ownership)>
```


Because I already know what the confirm method does, I'm just going to next over it…

```
[4] pry(#<Ownership>)> next
  TRANSACTION (3.4ms)  BEGIN
  Ownership Exists? (5.9ms)  SELECT 1 AS one FROM "ownerships" WHERE "ownerships"."user_id" = $1 AND "ownerships"."rubyg
"rubygem_id", 2], ["LIMIT", 1]]
  TRANSACTION (5.4ms)  ROLLBACK

From: /Users/mercedesbernard/Git/Personal/rubygems.org/app/models/ownership.rb:32 Ownership.create_confirmed:

    28: def self.create_confirmed(rubygem, user, approver)
    29:   ownership = rubygem.ownerships.create(user: user, authorizer: approver)
    30:   binding.pry
    31:   success = ownership.confirm!
 => 32:   binding.pry
    33: end

[4] pry(Ownership)> success
=> false
[5] pry(Ownership)> ownership.valid?
  Ownership Exists? (2.6ms)  SELECT 1 AS one FROM "ownerships" WHERE "ownerships"."user_id" = $1 AND "ownerships"."rubyg
"rubygem_id", 2], ["LIMIT", 1]]
=> false
[6] pry(Ownership)> ownership.errors.full_messages
=> ["User has already been taken"]
[7] pry(Ownership)>
```


…and verify that it returned false like I expected. Which it did.

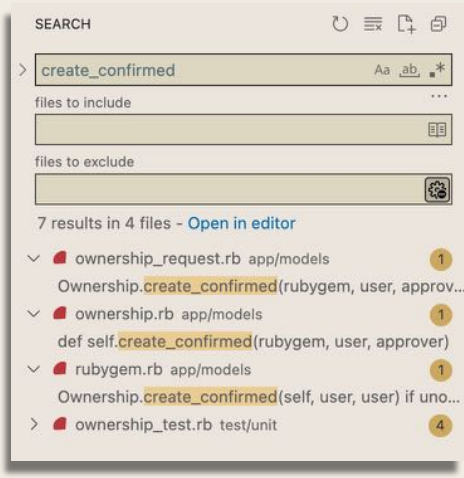
```
[4] pry(#<Ownership>)> next
  TRANSACTION (3.4ms)  BEGIN
  Ownership Exists? (5.9ms)  SELECT 1 AS one FROM "ownerships" WHERE "ownerships"."user_id" = $1 AND "ownerships"."rubyg
"rubygem_id", 2], ["LIMIT", 1]]
  TRANSACTION (5.4ms)  ROLLBACK

From: /Users/mercedesbernard/Git/Personal/rubygems.org/app/models/ownership.rb:32 Ownership.create_confirmed:

    28: def self.create_confirmed(rubygem, user, approver)
    29:   ownership = rubygem.ownerships.create(user: user, authorizer: approver)
    30:   binding.pry
    31:   success = ownership.confirm!
 => 32:   binding.pry
    33: end

[4] pry(Ownership)> success
=> false
[5] pry(Ownership)> ownership.valid?
  Ownership Exists? (2.6ms)  SELECT 1 AS one FROM "ownerships" WHERE "ownerships"."user_id" = $1 AND "ownerships"."rubyg
"rubygem_id", 2], ["LIMIT", 1]]
=> false
[6] pry(Ownership)> ownership.errors.full_messages
=> ["User has already been taken"]
[7] pry(Ownership)>
```

@mercedescodes | mercedesbernard.com

I'm also going to verify that the ownership is invalid.

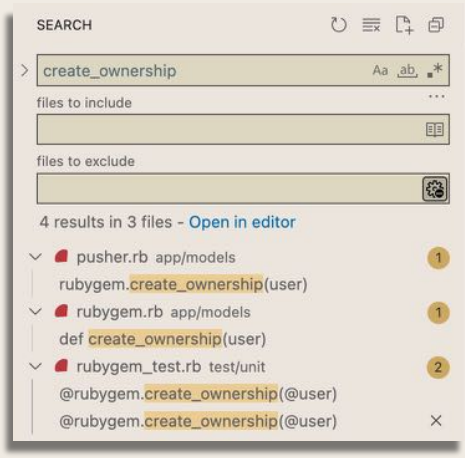
```
[4] pry(#<Ownership>)> next
  TRANSACTION (3.4ms)  BEGIN
  Ownership Exists? (5.9ms)  SELECT 1 AS one FROM "ownerships" WHERE "ownerships"."user_id" = $1 AND "ownerships"."rubyg
"rubygem_id", 2], ["LIMIT", 1]]
  TRANSACTION (5.4ms)  ROLLBACK

From: /Users/mercedesbernard/Git/Personal/rubygems.org/app/models/ownership.rb:32 Ownership.create_confirmed:

    28: def self.create_confirmed(rubygem, user, approver)
    29:   ownership = rubygem.ownerships.create(user: user, authorizer: approver)
    30:   binding.pry
    31:   success = ownership.confirm!
 => 32:   binding.pry
    33: end

[4] pry(Ownership)> success
=> false
[5] pry(Ownership)> ownership.valid?
  Ownership Exists? (2.6ms)  SELECT 1 AS one FROM "ownerships" WHERE "ownerships"."user_id" = $1 AND "ownerships"."rubyg
"rubygem_id", 2], ["LIMIT", 1]]
=> false
[6] pry(Ownership)> ownership.errors.full_messages
=> ["User has already been taken"]
[7] pry(Ownership)>
```


And I'm going to take a look at its errors which match what we'd expect. Because this create_confirmed method returns the value from the confirm! method, when an ownership can't be created, it returns false.




```ruby
def approve(approver)
  return false unless rubygem.owned_by?(approver)
  return false unless update(status: :approved, appr…)

  Ownership.create_confirmed(rubygem, user, approver)
end
```


So where are we using this create_confirmed method? We're using it in the ownership_request model when we're approving an ownership request.




```
def create_ownership(user)
  Ownership.create_confirmed(self, user, user) if uno…
end
```

@mercedescodes | mercedesbernard.com

And we're using it in another small method in the rubygem model. This tiny method is only called from 1 place that we'll see in a minute.

All of the code that we just looked at is pretty straightforward. It's not doing anything complex but it did take us a minute to get through the couple layers of methods. And we found that interesting convention violation where the confirm! method called the regular update and not the update! as we expected. No matter how senior we get, whenever we move into a new codebase we always have to start tracing the code somewhere.

And tracing the code you need one bug or feature at a time, is a great way to learn the whole codebase. Over time, you'll get an awareness of all the important pieces of code and the pieces that change frequently. By learning as-needed, you won't waste any of your time on parts that don't matter or that don't change often.

Settling in

Make changes

@mercedescodes | mercedesbernard.com

I'm feeling sufficiently welcome in the codebase. I have an understanding about the bits of the code that I'll be working in to resolve this first issue. So now I get to settle in and start to make the place a little bit my own.

```
models/ownership_request.rb: Ownership.create_confirmed

models/rubygem.rb: Ownership.create_confirmed

app/controllers/owners_controller.rb: rubygem.ownerships.new

app/controllers/api/v1/owners_controller.rb: rubygem.ownerships.new
```


At this point I know where ownerships are getting created.

```
models/ownership_request.rb: Ownership.create_confirmed

models/rubygem.rb: Ownership.create_confirmed
```

~~app/controllers/owners_controller.rb: rubygem.ownerships.new~~

~~app/controllers/api/v1/owners_controller.rb: rubygem.ownerships.new~~

We've ruled out silent failures in the 2 controllers.

~~models/ownership_request.rb: Ownership.create_confirmed~~

models/rubygem.rb: Ownership.create_confirmed

~~app/controllers/owners_controller.rb: rubygem.ownerships.new~~

~~app/controllers/api/v1/owners_controller.rb: rubygem.ownerships.new~~

And we verified in our debugging, trying to update and confirm an invalid ownership will return false. So it seems like approving an invalid ownership from the ownership request model will behave as expected.

~~models/ownership_request.rb: Ownership.create_confirmed~~

models/rubygem.rb: Ownership.create_confirmed

~~app/controllers/owners_controller.rb: rubygem.ownerships.new~~

~~app/controllers/api/v1/owners_controller.rb: rubygem.ownerships.new~~

It looks like there's only one place left where create could silently fail. So let's focus there and try to fix that.

Mimic

When making changes in a new codebase, try to mimic existing examples.

```
def update
  rubygem.disown if rubygem.versions.indexed.count.zero?
  rubygem.update_attributes_from_gem_specification!(version, spec)
  rubygem.create_ownership(user)
  set_info_checksum

  true
rescue ActiveRecord::RecordInvalid, ActiveRecord::Rollback, Active…
  false
end
```


This code comes from the Pusher class. We can see that it invokes rubygem.create_ownership which calls the custom class method we debugged earlier.

This code is doing a few other things and it rescues some exceptions. I want to be careful with my code changes so that if creating an ownership fails, I don't leave open the possibility to save a broken state. Using an ActiveRecord transaction feels like a good idea. But I also want to make sure that I match current codebase conventions.

I found a few examples in the code where they open transactions from the relevant ActiveRecord model. So I feel like I can go ahead and use this strategy for wrapping the ownership create.

But when I was working on this, I couldn't quite remember how ActiveRecord transactions handle different error types.

Documentation

@mercedescodes | mercedesbernard.com

Which brings me to the next strategy for settling in. Documentation! Don't be afraid to look up documentation for ActiveRecord classes, other dependencies, external libraries, browser behavior, database specs. Anything you could possibly need. Even when we've been coding for decades, we will never remember everything and answers are only a Google search away.

For this issue, I looked up how ActiveRecord transactions handled exceptions.

`Exception` handling and rolling back

Also have in mind that exceptions thrown within a transaction block will be propagated (after triggering the ROLLBACK), so you should be ready to catch those in your application code.

One exception is the `ActiveRecord::Rollback` exception, which will trigger a ROLLBACK when raised, but not be re-raised by the transaction block.

Da da da laaaaaa, all exceptions are re-raised except ActiveRecord::Rollback exceptions.

```
def update
  Rubygem.transaction do
    rubygem.disown if rubygem.versions.indexed.count.zero?
    rubygem.update_attributes_from_gem_specification!(version, spec)
    raise ActiveRecord::Rollback unless rubygem.create_ownership(user)
    set_info_checksum
    true
  rescue ActiveRecord::RecordInvalid, ActiveRecord::Rollback, Active…
    false
  end
end
```


So I'm going to go ahead and wrap the code in a transaction…

```
def update
  Rubygem.transaction do
    rubygem.disown if rubygem.versions.indexed.count.zero?
    rubygem.update_attributes_from_gem_specification!(version, spec)
    raise ActiveRecord::Rollback unless rubygem.create_ownership(user)
    set_info_checksum
    true
  rescue ActiveRecord::RecordInvalid, ActiveRecord::Rollback, Active…
    false
  end
end
```


keep the rescue block…

```ruby
def update
  Rubygem.transaction do
    rubygem.disown if rubygem.versions.indexed.count.zero?
    rubygem.update_attributes_from_gem_specification!(version, spec)
    raise ActiveRecord::Rollback unless rubygem.create_ownership(user)
    set_info_checksum
    true
  rescue ActiveRecord::RecordInvalid, ActiveRecord::Rollback, Active…
    false
  end
end
```


And raise an exception if creating an ownership failed.

```ruby
def update
   Rubygem.transaction do
     rubygem.disown if rubygem.versions.indexed.count.zero?
     rubygem.update_attributes_from_gem_specification!(version, spec)
     raise ActiveRecord::Rollback unless rubygem.create_ownership(user)
     set_info_checksum
     true
   rescue ActiveRecord::RecordInvalid, ActiveRecord::Rollback, Active…
     false
   end
 end
```


Now this code will rollback any database operations in the event of an exception. And it will still return false as expected. We just added an extra false case so that if an ownership is not created, it will also return false.

I chose to explicitly raise an exception if the create_ownership call fails there so that it would be clear to future contributors what this code does but I could've also chosen to refactor this method to throw the exception inside of it.

Error messages

@mercedescodes | mercedesbernard.com

While you're making a change, if you get any errors while running your code be sure to slow down and read what your errors are telling you. This seems so obvious but is also one of the tips I give to new pairs most frequently regardless of how many years of experience they have.

We have a tendency to want to go fast and be productive, especially when we're trying to prove ourselves in unfamiliar code. We make a lot of assumptions about why our code is broken. We're really quick to assume that we wrote everything wrong and we forget to read what the error message is actually telling us.

If you want to speed up your coding skills, this one small strategy to slow down will make you go much faster in the long run.

Being a good host

Make your codebase welcoming for future guests

@mercedescodes | mercedesbernard.com

I've made my change, left my mark on the place. Now I share in the responsibility to make this codebase welcoming to future guests. It's my turn to clean up and maybe leave out some treats to make the next guest feel comfy.

Commit messages & PR descriptions

Commit messages and PR descriptions will help future guests and contributors when they need to do their own code archaeology. Be sure to leave enough information in the history that someone following you 6 months from now understands what you did and why.

Enable remaining rails 6.1 defaults ···

```
We don't use action_mailbox so it doesn't affect us. We don't use
callbacks in active job either.
```

sonalkr132 committed on Jan 31 ✓

In a commit message, you don't have to limit yourself to the 50 character summary. You can also include a description with more in-depth information about what you did.

This is helpful if you need more info than what you can describe in 50 chars. For instance, when you're squashing commits. Or if you want to include a rationale for future contributors.

Description

Ownerships are created in the following places

- The owners API controller (the original location of the reported issue)

  rubygems.org/app/controllers/api/v1/owners_controller.rb
  Lines 18 to 21 in fc7cc3a

- The owners controller

  rubygems.org/app/controllers/owners_controller.rb
  Lines 34 to 36 in fc7cc3a

- The ownership request model

  rubygems.org/app/models/ownership_request.rb
  Lines 18 to 21 in fc7cc3a

- The pusher

  rubygems.org/app/models/pusher.rb
  Lines 127 to 136 in fc7cc3a

The only place where a silent failure could occur is in the last example.

In the first two, the original `.create` has already been replaced by a `.save` which has its return value checked. And in the third, the method will return false if trying to confirm an invalid ownership.

In the last one, the user for the ownership is already persisted and an ownership will only fail validation if a duplicate ownership exists. This should be a rare occurrence. However, I went ahead and added some extra safety just in case.

And in your PR descriptions, instead of just describing *what* code changes you made be sure to include the context or *why* you made the changes. Don't be afraid to link to docs, Github issues, or StackOverflow threads that helped you find the answer. You never know when that will help a future guest.

Valuable tests

Write valuable tests that describe the expected behavior. Rather than testing that your code didn’t raise an error or only testing happy paths, be sure to describe and test edge cases.

Rails tests (Ruby 3.1.2, RubyGems 3.3.11)

failed 4 days ago in 5m 4s

Search logs

```
56  Failure:
57  Api::V1::RubygemsControllerTest#test_: with index and push rubygem api key scope On POST to create
    for existing gem with confirmed ownership should respond_with success.
    [/home/runner/work/rubygems.org/rubygems.org/test/functional/api/v1/rubygems_controller_test.rb:300]:
58  Expected response to be a <2XX: success>, but was a <403: Forbidden>
59  Response body: There was a problem saving your gem:
60
61  rails test test/functional/api/v1/rubygems_controller_test.rb:298
62
63  .......F
```


When I opened a PR for the change to prevent silent create failures, I got a bunch of failing tests like this one. It’s a controller test to create a new version of an existing rubygem.

Rails tests (Ruby 3.1.2, RubyGems 3.3.11)

failed 4 days ago in 5m 4s

Search logs

```
56  Failure:
57  Api::V1::RubygemsControllerTest#test_: with index and push rubygem api key scope On POST to create
    for existing gem with confirmed ownership should respond_with success.
    [/home/runner/work/rubygems.org/rubygems.org/test/functional/api/v1/rubygems_controller_test.rb:300]:
58  Expected response to be a <2XX: success>, but was a <403: Forbidden>
59  Response body: There was a problem saving your gem:
60
61  rails test test/functional/api/v1/rubygems_controller_test.rb:298
62
63  .......F
```


And the key part of the description is “with confirmed ownership should respond_with success”. The only time creating an ownership is invalid is if one already exists. This test was added a year and a half ago (after the original issue was opened) to test that in this case, the code actually DOES silently fail. If I had looked at these tests more before I started coding, I might have caught this. Valuable tests will not only help prevent future guests (like me!) from breaking expected functionality, they can also serve as documentation for contributors to refer to.

mercedesb commented 2 days ago Contributor Author

After looking at the CI test failures, I think it makes sense to close this PR. The only remaining place where a create owner could silently fail is when it's expected to (as evidenced by the documentation the tests provide 🙌 🙌)

rubygems.org/test/functional/api/v1/rubygems_controller_test.rb
Lines 290 to 301 in ae28df5

```
290     context "On POST to create for existing gem" do
291       context "with confirmed ownership" do
292         setup do
293           create(:global_web_hook, user: @user, url: "http://example.org")
294           rubygem = create(:rubygem, name: "test")
295           create(:ownership, rubygem: rubygem, user: @user)
296           create(:version, rubygem: rubygem, number: "0.0.0", updated_at: 1.year.ago, created_at: 1.year.
297         end
298         should "respond_with success" do
299           post :create, body: gem_file("test-1.0.0.gem").read
300           assert_response :success
301         end
```

It looks like the original issue has been resolved since it was opened and could be closed.

mercedesb closed this 2 days ago

@mercedescodes | mercedesbernard.com

I ended up closing my PR and letting the maintainers know that the original issue we looked at was outdated and no longer needed. Even though my work for this issue didn't get merged, I'm not bummed. I still helped close an issue and as I got comfy in the code, I opened a couple other PRs with small things I noticed.

Documentation

Finally, good hosts create documentation and keep it up to date. I hope you're noticing a pattern that documentation is helpful. This includes tests, code comments, READMEs, diagrams, etc. It doesn't have to be long form, written docs but make sure that if you notice something is out-of-date, you do your part and update it.

@mercedescodes | mercedesbernard.com

When I was working on this issue, I noticed that the ERD we talked about earlier was out of date and missing some key models (like that ownership_request we looked at). So I opened a PR to update the ERD.

Commits on Dec 25, 2018

Update ERD to reflect current DB structure ···

Someone mentioned our ERD was 2 years out of date...

Generated with `$ rake gen_erd`

kerrizor committed on Dec 25, 2018 ✓

The diagram is a doc inside the repo. I didn't know how to update it, so I looked at the git history for the file. Big shoutout to Kerri who left the perfect commit message so that I would know how to update it.

voormedia / rails-erd Public

Edit Pins

Code | Issues 74 | Pull requests 11 | Actions | Projects | Security | Insights

add optional configuration to set which fonts to use #378

Open mercedesb wants to merge 1 commit into `voormedia:master` from `mercedesb:optional-fonts-config`

Conversation 0 | Commits 1 | Checks 0 | Files changed 7

mercedesb commented 18 days ago

This PR adds the ability to optionally configure font faces to use in the generated diagram.

While working on a PR in the rubygems.org repo to check the ERD in a Github action whenever the database schema changes (to ensure it's kept up to date), we realized that some of the diffs are dependent on what fonts the user has installed on their local machine.

By having optional configuration for fonts, the user can explicitly choose to bypass OS-dependent fonts for more consistent generated files.

Note: When I tried to run the tests, the output was super noisy (even without my changes) so I must have missed a set up step? Apologies if these tests are broken. I'll fix them if they are.

And after opening the PR, a contributor gave me a great suggestion to turn it into a Github Action and make checking that the ERD is up-to-date part of the CI since it was so easy for someone to miss this step.

Which kicked off this process all over again to make a contribution to the rails-erd repo so that we could have consistent output from gem in order to maintain the diagram as part of CI :)

Whether you're visiting a codebase for a short period of time or moving in for a bit longer, you don't need to know it inside and out to make valuable contributions. By making small contributions right away, you'll get comfy bit by bit and end up learning how all the parts of the system fit together faster than trying to learn it all at once. You don't need to where all the pots and pans are, how the stove works, and go buy groceries on your first day. You can microwave some soup while you look around the kitchen and get your bearings.

Thank you

@mercedescodes | mercedesbernard.com